Inhaltsverzeichnis

1 Einführung

1.1 Was ist ein Betriebssystem?

Die Aufgabe eines Betriebssystems ist nicht ganz so einfach geschlossen in einem kurzen Satz zu beschreiben wie das manchmal bei anderen Softwaresystemen möglich ist. Meist stellt man es sich als ein grundlegendes Programmsystem vor, das die sinnvolle Benutzung eines Rechners überhaupt erst ermöglicht.

In der DIN-Norm 44300 ist folgendes „definiert“:

> **Betriebssystem** (operating system): *Die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften der Rechenanlage die Grundlage der möglichen Betriebsarten des digitalen Rechensystems bilden und insbesondere die Abwicklung von Programmen steuern und überwachen.*

1.1.1 Top-Down vs. Bottom-Up

Die Sichtweise, eine einheitliche Aufgabe definieren zu wollen, ist genau die, die dem „**Top-Down**“-Entwurf bei der Entwicklung der Software entspricht. Dabei wird „die“ Aufgabe des geforderten Systems festgelegt und formuliert und dann, um eine sichere und einfache Entwicklung zu ermöglichen, in Teilaufgaben zerlegt.

Bei einem Betriebssystem liegen die Dinge aber eher so, dass das System nichts *in sich selbst Sinnvolles* tut – es schafft dagegen die Bedingungen dafür, dass *andere Programme* (Anwenderprogramme) sinnvolle Dinge tun können. Da diese Programme von sehr vielfältiger Gestalt sein können, sind auch die Aufgaben des Betriebssystems entsprechend vielfältig und nicht gut geschlossen zu beschreiben.

Eine bessere Sichtweise wäre die „**Bottom-Up**“-Sichtweise. Es liegt ein geordnetes System von in sich geschlossenen Bauelementen vor, die jeweils unterschiedliche Aspekte der Benutzeranforderungen bedienen.

Nicht umsonst ist die Bezeichnung „Betriebssystem“ und nicht „Betriebsprogramm“:

> **System:** *Sammlung oder Kombination zusammengehörender Teile zu einem Ganzen; aus dem Griechischen σύστημα; συν- = zusammen, ἵσταμαι = stellen, sich aufrichten, −μα = Resultat einer Aktion.*

Wir sehen also das Betriebssystem an als eine Art „Schnittstelle“: eine Sammlung von „Programmen“ (in welcher Form genauer auch immer), die zwischen den Benutzern und der Hardware eines Rechnersystems vermitteln.

1.1.2 Mehrbenutzer-Systeme

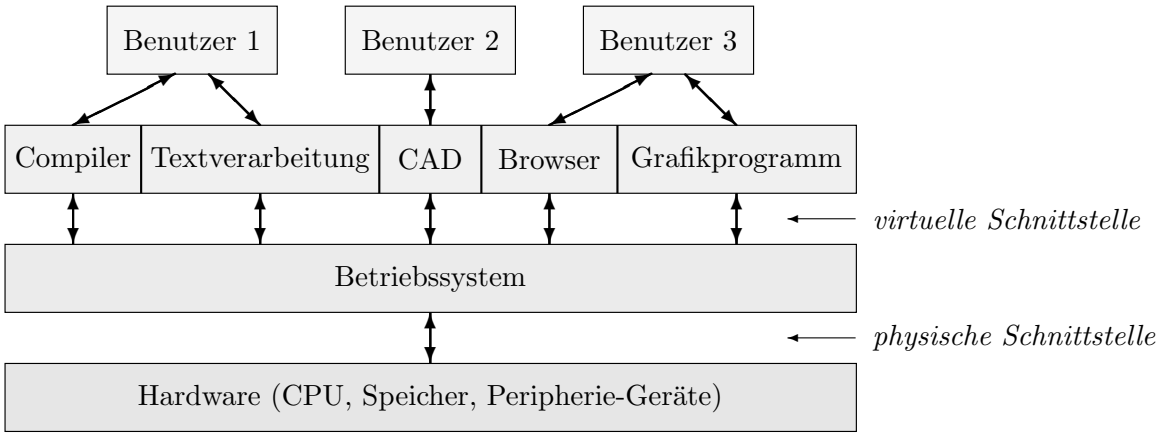
Die meisten Betriebssysteme ermöglichen es, dass nicht nur mehrere Programme „gleichzeitig“ ablaufen können („Prozesse“), sondern auch, dass mehrere Benutzer gleichzeitig Zugriff haben.

Das Arbeiten mit dem System sollte für den Menschen möglichst angenehm sein. Benutzer sollen insbesondere den Rechner nicht nur nacheinander benutzen können, also jeweils auf

ihre erlaubte Arbeitszeit warten müssen. Jeder Benutzer soll in seinen Aktionen aber nicht eingeschränkt sein – er soll also so arbeiten können, als wäre er allein. Es ist klar, dass das nicht ganz ohne Effizienzeinbußen möglich ist; diese Einbußen sollen aber so gering wie in diesem Rahmen möglich gehalten werden.

Die „*Ressourcen*“ (Betriebsmittel) wie Speicher, Peripherie, etc. sollen gerecht und effizient auf die Benutzer verteilt werden. Die Benutzer sollen im Normalfall völlig voneinander abgeschirmt sein (Speicher- und Datei-Schutz), es soll aber kontrollierte Kommunikation untereinander unterstützt werden.

Diese Sicht von Betriebssystem ist im folgenden Bild dargestellt:



Die Ebenen können eventuell weiter unterteilt werden, beispielsweise

- das Betriebssystem in anwendernähere und maschinennähere Teile,
- der CPU-Teil in Maschinensprachen-Ebene/Mikro-/Nano-Programmierung,
- der Hardware-Teil in die I/O-Schnittstelle, Controller und eigentliche Peripherie-Geräte, etc.

Das Betriebssystem „multiplext“ die tatsächlich vorhandenen Ressourcen. Durch Platz-Aufteilung (bei Speicher, Platte, etc.) bzw. Zeitscheiben-Zuteilung (beim Prozessor) wird die Illusion von mehr und größeren Ressourcen erzeugt, als tatsächlich vorhanden sind.

> **Betriebssystem als Jongleur**: Jeder Benutzer erhält die Illusion, den Prozessor ganz für sich zu haben, „unbeschränkt“ viel Speicher zur Verfügung zu haben, etc.

Wenn die tatsächlichen Ressourcen dem Anwender in Form von „logischen“ Ressourcen dargeboten werden, wobei das Betriebssystem eine entsprechende Abbildung vornimmt, spricht man von **Virtualisierung** der Ressourcen (virtueller Speicher, virtuelle Maschine, etc.):

> **virtuell:** *der Möglichkeit/dem Wesen nach vorhanden, fähig, in bestimmter Weise zu wirken; von französisch* virtuel=*wirkungsfähig, von lateinisch* virtus=*Tugend, Mannhaftigkeit, von* vir=*Mann.*

Wichtige Aufgaben des Betriebssystems liegen darin, dafür zu sorgen, dass Benutzer nicht unnötig lange (oder gar für immer) auf die von ihnen benötigten Ressourcen (inklusive die CPU) warten müssen. Andererseits sollte nicht so schnell multiplext werden, dass durch die technischen Umstände die Effizienz zu stark leidet.

1.1.3 Schnittstelle zur Hardware

Das Rechnersystem besteht ebenfalls aus vielen Komponenten: aus Prozessor oder Prozessoren, Speicher und diversen Peripherie-Geräten. Die Ansteuerung dieser Teile kann recht kompliziert und außerdem von System zu System und Gerät zu Gerät völlig unterschiedlich sein.

Typischerweise spricht man ein Peripherie-Gerät dadurch an, dass man bestimmte Werte in Register des zuständigen Controller-Bausteins schreibt oder nur bestimmte Bits in diesen Registern setzt oder löscht.

Diese Ansteuerung der Hardware sollte aber nie direkt in den Anwenderprogrammen erfolgen:

- **Abschirmung der Anwendungs-Programmierer**:

  Dem Autor von Anwenderprogrammen ist es nicht zuzumuten, die genauen Details zu lernen. Alle Autoren von Anwenderprogrammen müssten sich mit den Hardware-Details bestens auskennen. Das würde Zeit und Energie von der Lösung des eigentlichen Problems abziehen.

- **Code-Duplizierung**:

  Jedes Anwenderprogramm würde fast dieselben Ansteuerungsroutinen enthalten. Um Daten zu sichern, müsste man ja beispielsweise den Harddisk-Controller selbst programmieren. Das läuft im Wesentlichen immer gleich ab, käme in jedem Programm vor und würde es unnötig verlängern.

- **Hardware-Austausch**:

  Wenn Hardware ausgetauscht wird (beispielsweise ein neuer, schnellerer Disk-Controller eingesetzt wird), müssten alle Anwenderprogramme geändert und neu übersetzt werden. Wenn ein Programm eine möglichst große Palette von Hardware unterstützen soll, müsste es Routinen für alle denkbaren Controller enthalten.

- **Benutzer-Konflikte**:

  Bei Multitasking und Mehrbenutzer-Systemen verbietet es sich von vornherein, direkt auf Hardware zuzugreifen. Die Anfragen mehrerer Benutzer kämen bunt gemischt bei der Peripherie an und hätten unvorhersagbare Effekte zur Folge.

Das Betriebssystem implementiert daher eine „**virtuelle Maschine**“, also eine Art Prozessor mit einem größeren Befehlsvorrat, als der tatsächlich vorhandene besitzt. Die zusätzlichen Befehle („**virtuelle Befehle**“) dienen dann zur Ansteuerung der Hardware über eine Ebene von Abstraktion.

Die Abstraktion kann dabei in mehreren Stufen erfolgen. Bei einem Platten-ähnlichen Gerät könnte das wie folgt aussehen:

0. **Steuerung über Werte in Controller-Registern**
   (keine virtuellen Befehle, Transport-Befehle der CPU)

1. **primitive Ansteuerungs-Routinen:**
   – *fahre den Schreib-/Lesekopf an eine bestimmte Position*
   – *schreibe/lies ein Bit nach/von dort, wo der Schreib-/Lesekopf steht*

2. **Aufteilung der Platte in Sektoren/Spuren, bzw. linear numerierte Blöcke:**
   – *schreibe Speicherblock (Adresse a, Länge ℓ) in den Block n auf der Platte*

3. **Dateisystem, hierarchisch über Verzeichnisse geordnet:**
   – *schreibe Speicherdaten (a,ℓ) in Datei Pfad/Name*

4. **Datenbanken**, kein Direktzugriff auf die Einzeldateien
   – *sortiere Datensatz (x, y, z) in Datenbank Q ein*

Man hat es niemals mit den echten Ressourcen, sondern immer nur mit „**virtuellen Ressourcen**“ zu tun, also mit Abstraktionen der physischen Ressourcen:

Prozess:	Virtualisierung des ganzen Rechners (CPU, benutzter Speicher, benutzte Ressourcen)
Virtueller Speicher:	Abstraktionsebene über dem physischen Speicher (mag im Hauptspeicher, mag aber auch gerade auf Platte liegen)
Dateien:	Virtualisierung des Plattenspeichers
Logische Geräte:	Virtualisierung der physischen Geräte (dateiähnlich, `PRN:` statt des echten Druckers, `/dev/console` statt des Terminals, RAM-Disk – Hauptspeicher, der wie eine Platte angesprochen wird, etc.)

Die Aufgaben des Betriebssystems im Umgang mit den Ressourcen sind im Wesentlichen folgende:

- **Zuteilung** von Ressourcen an Prozesse, die sie benötigen
- **Buchführung** darüber, welche Ressourcen frei sind und welche belegt (und von wem)
- **Entscheidung** bei mehreren Prozessen, die dieselbe Ressource benötigen, welcher sie zuerst zugeteilt bekommt
- **Zugriffsschutz** – d.h. Sicherstellung, dass eine Ressource nur angesprochen wird, wenn sie bereit ist

1.1.4 Komponenten des Systems

Es ist nicht immer völlig offensichtlich, welche Teile der Software auf einem Rechnersystem zum Betriebssystem gehören, und welche eigentlich nur Anwenderprogramme sind:

CPU-Verwaltung (Prozesse), Speicherverwaltung:

gehören mit Sicherheit zum Betriebssystem

Verwaltung von I/O-Geräten:

eventuell als „Treiber“ aufzufassen, also aus dem eigentlichen Betriebssystem ausgegliedert – und umgekehrt modular hinzufügbar

Shell, Kommandozeileninterpreter:

strittig, wird meist nicht als Teil des Betriebssystems angesehen

Editor, Compiler, Bibliotheksverwaltung:

definitiv Anwenderprogramme

grafische Oberfläche, Internet-Browser:

noch definitiver

Die Punkte 1 und ggf. 2 bilden den sogenannten „**Kern**“ des Betriebssystems. Genauer legt man meist folgendes fest:

> *Der* **Kern** *des Betriebssystems ist all das, was* **zu jedem Zeitpunkt** *nach dem Start auf dem Rechner betriebsbereit im Speicher liegt.*

1.1.5 Wozu Betriebssysteme studieren?

Warum sollte man sich nun überhaupt mit Betriebssystem-Konzepten beschäftigen? Man wird vermutlich niemals selbst ein (wirkliches) Betriebssystem schreiben. Aber:

- Durch das Verständnis der im Hintergrund ablaufenden Vorgänge kann man eigene Programme effektiver gestalten.
- Eigene Programme lassen sich ggf. günstig durch *mehrere* Prozesse bzw. Threads verwirklichen. Diese Mechanismen lernt man beim Betriebssystem-Studium genau kennen.
- Man lernt einiges über das Design komplexer Systeme, was bei der Entwicklung anderer Software nützlich sein wird.
- Die Prinzipien aus der Ressourcen-Verwaltung finden auch in anderer Software Anwendung.

Das Schreiben eines neuen Betriebssystems ist natürlich außerdem ein Parade-Beispiel für Software-Engineering. Beim Entwurf ist schon zu beachten, dass das Gesamt-Design und die Implementation möglichst einfach sind. Das System soll für die erstmalige Entwicklung und die Weiterentwicklung überschaubar bleiben. Es ist notwendigerweise schon komplex genug – man sollte es also nicht unnötig komplex (= kompliziert) machen.

1.2 Historische Entwicklung der Betriebssysteme

Hier haben seit der Entwicklung der ersten Rechnersysteme besonders viele Veränderungen stattgefunden, mehr als in irgendeinem anderen Gebiet. Die Hardware wurde mit enormer Geschwindigkeit verbessert, wodurch sich völlig veränderte Einsatzmöglichkeiten ergeben haben. Dadurch wiederum wurden allmählich höhere Anforderungen an die Leistung des Gesamtsystems gestellt, die letztendlich die Entwicklung leistungsfähigerer Hardware erzwangen, etc.

1.2.1 Zu Beginn: nackte Rechner

Die Rechner ganz zu Beginn des Computer-Zeitalters waren sehr groß und extrem teuer. Sie wurden per Hand aus einer Unzahl von Röhren zusammengebaut und waren entsprechend anfällig. Alle paar Stunden musste eine defekte Röhre durch spezielles Personal ausgetauscht werden. Diese Rechner waren natürlich völlig experimentell und konnten noch nicht kommerziell eingesetzt werden. Die Hauptaufgaben waren rein numerische komplexe Berechnungen.

- Der Bediener des Rechners war gleichzeitig der **Programmierer** und der **Operator**. Das „**Programmieren**“ bestand in irgendeiner Form von **Umkonfiguration** des Rechners:
  - in den 40er Jahren: durch Umstecken von Kabeln und Umlegen von Schaltern
  - in den 50er Jahren: durch Einlesen von Papierband, später von Lochkarten

  Nach der Konfiguration wurde der Rechner gestartet. Die Ausgabe der Ergebnisse erfolgte dann durch Lichtanzeigen oder über Papierband, d.h. die CPU übernahm diese Art von I/O selbst.

  Zum Debuggen interpretierte man dann die gelieferten Ergebnisse, entwarf Programmteile neu auf Papier und erstellte eine angepasste Rechnerkonfiguration. Eine Interaktion mit dem Rechner während des Programmlaufs war nicht (oder nur sehr schwer) denkbar.

- Zu diesem Zeitpunkt waren **Computer-Benutzer** also ausschließlich **Computer-Experten**. Sie erhielten Aufträge mit Informationen darüber, was zu berechnen war, und waren danach bei der Problemlösung auf sich gestellt.

- Für die Benutzung des Rechners von mehreren Operatoren wurde ein **Stundenplan** geführt, in den jeder seine gewünschten Arbeitsstunden vorab eintragen musste. Dabei ergeben sich natürlich einige Probleme:
  - Eventuell wurde für die Problemlösung **mehr Zeit** benötigt als geplant (unerwartete Fehler). Nach Ablauf der zugeteilten Zeit musste also **abgebrochen** werden, die Ergebnisse mussten in irgendeiner Form gesichert werden (Papierstreifen, etc.). Beim nächsten Mal war dann zunächst einige **Einarbeitungszeit** erforderlich.
  - Andersherum konnte es natürlich auch passieren, dass man einmal **schneller fertig** war, als man erwartet hatte. Wenn der nachfolgende Benutzer noch nicht da war, blieb der Rechner also einige Zeit völlig **ungenutzt**.

  Dieses Verfahren ließ sich schlecht umgehen, war aber nicht besonders effektiv und daher ziemlich teuer.

1.2.2 Röhren → Transistoren

In der Zeit, als Röhren allmählich durch Transistoren ersetzt wurden, entwickelten sich die Rechner auch zu „**universellen Maschinen**“. Sie wurden also nicht speziell für eine Aufgabe gebaut oder konfiguriert, sondern erhielten ihre Aufgabe in Form eines **Programms** beschrieben. Die Software wurde also aus der Hardware herausgelöst.

Die Programme mussten zunächst natürlich in der **Maschinensprache** des jeweiligen Rechners geschrieben werden. Etwas später wurden dann die ersten **höheren Programmiersprachen**

entwickelt, von denen sich `FORTRAN` am schnellsten und weitesten verbreitete, später im kommerziellen Bereich auch `COBOL`. Dadurch wurde die Programmentwicklung schneller und (etwas) sicherer.

Dadurch dass die Hardware zuverlässiger wurde (Transistoren) und das Programmieren einfacher, wurden die Rechner allmählich auch **kommerziell** einsetzbar, wenn auch zunächst nur in großen Betrieben und Universitäten.

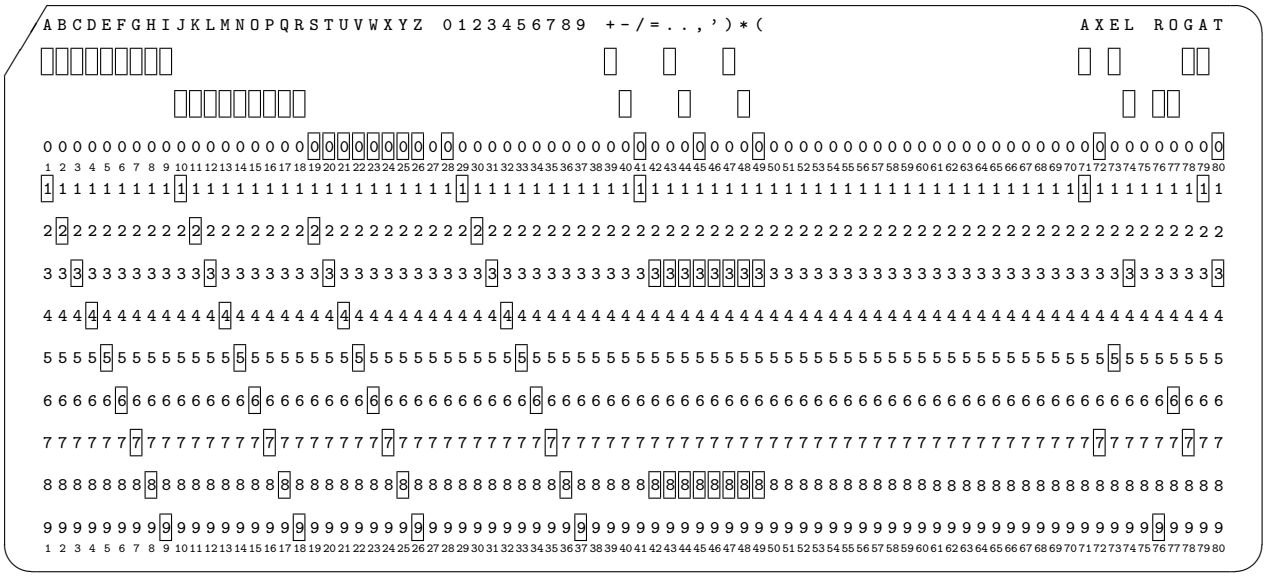
Der Job des Computer-Bedieners teilte sich nun auf in zwei:

- **Programmierer** (Autor der Software)
- **Operator** (Hardware-Bediener)

Die Eingabe erfolgte üblicherweise über **Lochkarten**, die Ausgabe auf Papier oder (zur weiteren Verwendung) wieder auf Lochkarten. Häufig benutzte (unveränderliche) Daten wurden auf Magnetband gespeichert, beispielsweise Assembler, Compiler, Linker, Bibliotheken.

Kurzer Einschub: Format von Lochkarten

Eine Lochkarte stellt üblicherweise eine Zeile Programmtext (80 Zeichen) oder entsprechend viele binäre Daten dar. Sie ist wie folgt aufgebaut:



Die Karte ist in 12 Zeilen zu je 80 Spalten aufgeteilt. In jedes so entstehende Feld kann ein rechteckiges Loch gestanzt werden. In jeder Spalte wird ein Zeichen oder bei Binärdaten ein binäres Datum dargestellt.

In der zeichenorientierten Darstellung (Hollerith-Code) bilden die Daten in den Zeilen 0, 11 und 12 den Zonenteil, die in den Zeilen 1 bis 9 den numerischen Teil eines Zeichens. Es darf jeweils maximal 1 Loch in den Zonenteil und maximal zwei in den numerischen Teil gestanzt werden, sodass viel Platz verschwendet wird (48 definierte Zeichen, $2^{12} = 4096$).

In der binären Darstellung wird die Karte in eine obere und eine untere Hälfte aufgeteilt. In jeder Spalte werden zwei 6-Bit-Binärzahlen dargestellt. Wenn diese Zahlen über einen Code wieder als Zeichen aufgefasst werden (64 mögliche Zeichen), passen so also zwei Zeichen in eine Spalte.

Der übliche Ablauf bei der Entstehung eines Programms war folgender:

1. **Programmierer:**
   - Entwicklung des Programms auf Papier
   - Austesten zunächst mit Bleistift und Radiergummi
   - Stanzen des Programms in Lochkarten
   - Abliefern eines Stapels Lochkarten beim Operator:
     „**Job**“ = Programm und Daten, entsprechende Anweisungen (Compiler-Aufruf, etc.)

2. **Operator:**

   sobald der Rechner „frei“ wird (vorheriger Job beendet):
   - Auswahl eines der vorliegenden Kartenstapel
   - Einlesen des Kartenstapels in den Rechner
   - ggf. Einlesen zusätzlich benötigter Ressourcen (`FORTRAN`-Compiler vom Band)
   - Starten des Programms
   - Ausgabe (Papier/Karten) sammeln und an den Programmierer schicken

3. **Programmierer**:
   - **Debuggen**:
     anhand der gelieferten Ergebnisse
     bei einem **Absturz**: anhand des gelieferten Ausdrucks der Registerinhalte und des kompletten Hauptspeicherinhalts („**core dump**“)
   - neuer Versuch

So wurde einiges an Zeit verschwendet. Der Operator lief zwischen Kartenleser, -stanzer und Magnetbandmaschine hin und her. Die Kartenstapel mussten letztendlich durch die CPU eingelesen werden; in dieser Zeit konnte nicht gerechnet werden. Bei der Preislage der damaligen Rechner (und Operatoren) kostete jede verschwendete Minute Tausende.

Lösung 1: Batch-Systeme

> **batch** = *Schub, Stoß, also hier: Zusammenfassung mehrerer Befehle*

Als erste Verbesserung wurden zusammenpassende Jobs gesammelt, beispielsweise alle, die den `FORTRAN`-Compiler benötigen, alle, die den `COBOL`-Compiler benötigen, etc.

Die Compiler werden jeweils von einer bestimmten Bandmaschine geladen. Der Operator muss das Band jeweils per Hand austauschen und den neu benötigten Compiler einlesen lassen. Durch die Job-Sammlung kann hier Zeit gespart werden – der Compiler muss nicht für jeden Job gewechselt werden.

Lösung 2: Automatic Job Sequencing und Monitore

Bei herkömmlichem Operator-Betrieb war der Kartenstapel, den der Benutzer dem Operator übergab, aufgeteilt in Karten mit Programmen (`FORTRAN`-Quellcode) und numerischen Daten. Was wann einzulesen war, und was damit zu geschehen hatte, entschied der Operator.

Um nun die Daten in den einzelnen Jobs bei Batch-Betrieb günstig zu organisieren, wurden spezielle neue Karten-Typen eingeführt.

Die neuen Karten dienten zur Beschreibung des Zwecks der jeweils nachfolgenden Karten („**Kontroll-Karten**“). Als Kennzeichnung enthielten sie als erstes Zeichen (also in der ersten Spalte) meistens ein Dollar-Zeichen `$` oder ein unbelegter Codewert (wie 7-9). Nur IBM benutzte *zwei* Spalten `//`. Danach folgte jeweils die Spezifikation der Karte – in der Art eines kurzen Befehlsnamens:

`$JOB`	oben auf dem Stapel, Beginn des Jobs, ID, Benutzername, etc.
`$END`	unter dem Stapel, Ende des Jobs, ggf. Wiederholung der ID
`$FTN`	(oder `$FORTRAN`) – Laden und Starten des `FORTRAN`-Compilers (vom Band), das Programm steht auf den nachfolgenden Karten bis zur nächsten Kontroll-Karte. Das übersetzte Programm landet auf einem speziellen System-Band.
`$COBOL`	Laden und Starten des `COBOL`-Compilers
`$ASM`	Laden und Starten des Assemblers
`$LOAD`	Zurückspulen des Systembands, Laden der erzeugten Daten als Programm
`$RUN`	Starten des geladenen Programms. Die Eingabe-Daten für das Programm stehen auf den nachfolgenden Karten bis zur nächsten Kontroll-Karte.
`$LGO`	load-and-go, Kombination aus `$LOAD` und `$RUN`
`$DATA`	Eventuell zusätzliche Markierung eines Bereichs als (Eingabe-)Daten

Ein Job bestand nun also aus einem einzigen Kartenstapel, der intern durch Kontroll-Karten strukturiert war. Die Zugehörigkeit der Daten zum Auftraggeber ist durch die ID des Jobs gegeben. Die CPU kann beliebig viele Jobs direkt hintereinander ausführen – Kartenstapel können miteinander verschmolzen werden. Die CPU startet so lange automatisch neue Programme, wie sie entsprechende Karten erhält („**automatic job sequencing**“).

Die Interpretation der Kontroll-Befehle geschah im Großrechner. Ein entsprechendes kleines Programm musste dafür **immer** ausführbereit im Speicher liegen. Dieses Programm wurde „**Monitor**“ (oder „resident monitor“) genannt. Es stellt damit also das **erste Betriebssystem** der Geschichte dar (genauer einen Betriebssystem-Kern).

Der Monitor wird direkt nach dem Start in den Speicher geladen und bleibt von da an unverändert dort. Er besteht im Wesentlichen aus folgenden Teilen:

Control Card Interpreter: wertet die Befehle auf den Kontroll-Karten aus

Loader: lädt ein Benutzer- oder ein System-Programm (Compiler) in den Speicher und macht es dort lauffähig

Job Sequencer: startet ein geladenes Programm (übergibt ihm die CPU-Kontrolle), übernimmt nach dessen Beendigung die Kontrolle und stößt die weitere Befehlsverarbeitung an

Device Drivers: Geräte-Treiber für die permanent benötigte Peripherie (hauptsächlich die Bandmaschinen); eventuell werden die Treiber bei Bedarf auch nachgeladen.

Als sich diese System-Betriebsart mit Monitor mehr und mehr durchsetzte und die Monitoren mehr Aufgaben übernahmen, bauten die Hersteller eine spezielle Hardware-Unterstützung in die CPUs ein. Der Satz von Maschinenbefehlen wurde zweigeteilt („**dual-mode instructions**“) in:

privilegierte Befehle:	nur vom Monitor verwendbar („Supervisor Mode“)
Benutzer-Befehle:	auch von Anwenderprogrammen verwendbar („User Mode“)

Typische Monitorsysteme der damaligen Zeit waren `FMS` (`FORTRAN` Monitor System) und `IBSYS` (für die IBM 7094). Erst später, als die Systeme ausgebaut wurden und der Name Monitor ihnen nicht mehr ganz gerecht wurde, entwickelte sich allmählich der Begriff Betriebssystem (*Operating System* wie in IBM OS/360, s.u.).

Die Kontroll-Karten bzw. -Befehle wurden oft auch zu einer (sehr kleinen) Batch-Sprache ausgebaut. Beispielsweise bot **JCL** (Job Control Language) von IBM die Ausführung von Befehlen in Abhängigkeit von Bedingungen, etc. Die heutigen Skript-Sprachen sind Abkömmlinge dieser einfachen Batch-Sprachen.

Lösung 3: Offline-Betrieb

Eine wesentliche Effizienzsteigerung ergab sich später durch den sogenannten „**Offline-Betrieb**“. Zusammenpassende Jobs (mit sich überschneidenden System-Ressourcen wie Compiler) werden zunächst auf einem gesonderten kleinen Rechner (im „Eingaberaum“) von Karten auf Magnetband übertragen („**satellite processing**“). Sie können danach direkt hintereinander auf dem eigentlichen Großrechner ausgeführt werden.

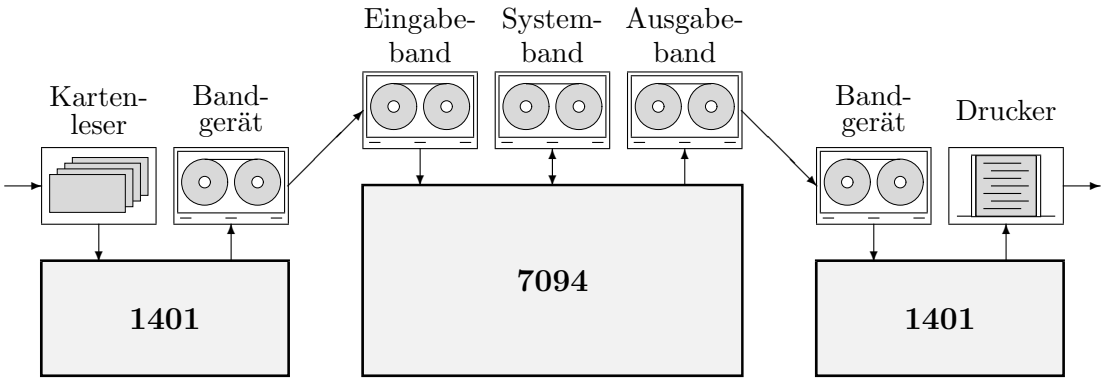
- Auf diese Weise wird der Großrechner von langsamen I/O-Arbeiten befreit. Das Einlesen von Band ist um Größenordnungen schneller als das von Lochkarten (z.B. ca. 1000 Karten = Programmzeilen pro Minute). Müsste die CPU die Karten selbst einlesen, wäre sie 90 % der Zeit mit dem Warten auf das Lesegerät „beschäftigt“. Entsprechendes gilt für die Ausgabe.

- Wenn ein Band voll ist, wird es zurückgespult und für den Großrechner bereitgelegt. Wenn der Großrechner droht, arbeitslos zu werden, kann das Zurückspulen natürlich auch vorzeitig erfolgen.

  Der Großrechner schreibt nun die Ausgaben der Jobs auf dem Band nicht auf Karten oder Papier, sondern (schneller) auf ein weiteres Magnetband. Im „Ausgaberaum“ werden die Daten von Band auf Papier oder Karten übertragen, wiederum durch einen kleineren Rechner, und in dieser Form an die einzelnen Benutzer verteilt.

Voraussetzung für diese Aufteilung ist entsprechende Hardware, nämlich kleine Rechner, die (im Vergleich mit dem Großrechner) rechnenschwach sein dürfen, dafür aber auch wesentlich billiger sein müssen („reader-to-tape“- und „tape-to-printer“-Maschinen). Ein typischer Fall war:

IBM 7094: Großrechner, auf Rechengeschwindigkeit optimiert
IBM 1401: Vorrechner, für die Ein-/Ausgabe-Operationen benutzt



Die IBM 1401 wurde auch alleine genutzt für einfache Datenspeicherung in Banken und Versicherungen.

Im günstigsten Fall gibt es mehrere Vor- und Nachrechner, sodass der teure Großrechner möglichst lückenlos ausgelastet sein kann.

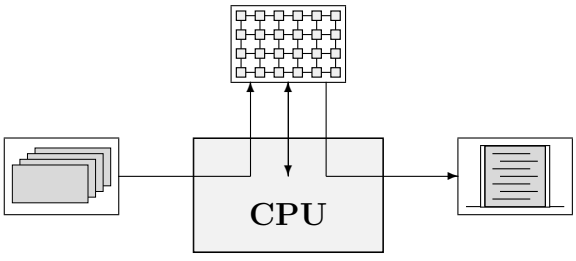
- Für die eigentlichen Anwenderprogramme änderte sich bei Offline-Betrieb in Bezug auf Ein- und Ausgabe nichts. Wurden früher die Datensätze von physischen Karten gelesen, so stammen sie jetzt von „virtuellen Karten“, also solchen, die auf Band überspielt wurden.

  Für das Programm geschieht diese Umsetzung völlig transparent. Es spricht eine System-Routine an, die normalerweise die nächste Karte liest, jetzt aber auf das Lesen vom Magnetband „umgeleitet“ wurde. Das ist das erste Auftauchen einer Art von „**Geräte-Treibern**“.

Lösung 4: Puffern und Interrupts

Eine zusätzliche Verbesserung des Durchsatzes kann im Zusammenhang mit Ein-/Ausgabegeräten bereits durch einfaches Puffern erreicht werden:

- Ein Eingabegerät liefert kontinuierlich Daten, auch, wenn die CPU sie noch nicht benötigt. Die Daten werden bis zu einer bestimmten Datenmenge (z.B. 10 Karten) im Hauptspeicher zwischengelagert (gepuffert).
- Analog schreibt bei der Ausgabe die CPU kontinuierlich die Ausgabedaten, die sie erzeugt, in einen Puffer – auch, wenn das Ausgabegerät nicht mitkommt.



Die Software für diesen Pufferungs-Mechanismus wird im Monitor oder im entsprechenden Geräte-Treiber untergebracht. Es liegt also eine weitere Zwischenstufe an Virtualisierung vor – es wird nicht unbedingt direkt die wirkliche Treiber-Routine angesprochen, sondern die zwischengeschaltete Pufferungs-Routine.

Es gibt eine wichtige Hardware-Voraussetzung für das Puffern, nämlich einen **Interrupt**-Mechanismus in der CPU:

Die CPU muss hardwaremäßig *vom Peripherie-Gerät* unterbrochen werden können. Sie nimmt dann das nächste Datum (oder Datenpaket) vom Eingabegerät an und schreibt es in den Puffer, bzw. liefert dem Ausgabegerät das nächste Datum (Datenpaket) aus dem Puffer. Wenn das geschehen ist, kehrt die CPU zu ihrer ursprünglichen Tätigkeit zurück.

Das Unterbrechen geschieht direkt durch ein Signal („**Interrupt**“) an einem CPU-Anschluss. Es gibt im allgemeinen mehrere Signale oder eine zusätzliche Möglichkeit, Signale durchzunumerieren. Die CPU schaut dann in einer Tabelle nach, an welcher Stelle im Hauptspeicher die zum empfangenen Signal gehörige Behandlungsroutine steht und führt sie aus.

Auf diese Weise können Schwankungen in der Lese-/Schreib- bzw. Berechnungsgeschwindigkeit ausgeglichen werden. Bei den meisten Programmen wird allerdings die Ein-/Ausgabe der langsamere Teil sein, sodass Puffern nur einen (vergleichsweise) geringen Effekt auf die CPU-Ausnutzung hat.

Insgesamt lässt sich über die Rechnersysteme dieser Zeit sagen, dass sie vergleichsweise gut ausgenutzt werden konnten bei rechenintensiven Aufgaben, also in Universitäten, Rechenzentren, etc. Im kommerziellen Betrieb, wie bei der Datenverwaltung in Banken und Versicherungen, überwiegt meist der I/O-Anteil an der Gesamtzeit, und das System wird nicht gut ausgenutzt.

Ein wichtiger Nachteil des Batchings muss erwähnt werden: die Benutzung des Rechners ist **nicht mehr interaktiv**:

Vor dem Batching: der Rechner stand dem Programmierer während der ihm zugeteilten Zeit allein zur Verfügung. Es konnte zwar schwer während der Laufzeit eines Programms eingegriffen werden. Leichtere Fehler konnten aber schnell im Ausdruck erkannt und korrigiert werden, und es konnte schnell ein neuer Durchlauf gestartet werden, ggf. ohne ein komplettes neues Einlesen und Konfigurieren.

Bei Batch-Systemen: Es ist keine direkte Überwachung des Programms mehr möglich. Nach dem Debuggen (per Hand) muss ein neuer Job kreiert und dem Operator übergeben werden. Die Zeit, bis der nächste Testlauf stattfinden kann, nennt man „**Turnaround-Zeit**“ (in dieser Zeit kann sich der Programmierer vom Rechner abwenden). Sie ist bei Batch-Systemen vergleichsweise groß.

1.2.3 Platten und integrierte Schaltungen

In den 60er Jahren gab es viele Entwicklungen im Hardware-Bereich, die direkte Auswirkungen auf den Rechnerbetrieb hatten:

Plattenspeicher lösten Magnetbänder als schnelle Massenspeicher ab. Platten mussten nicht mehr wie Bänder sequentiell gelesen werden. Man konnte mit nur geringer Zeitverzögerung auf Daten an beliebigen Stellen des Mediums zugreifen. Dadurch wurden die rechnerinternen Zwischenspeicher um Größenordnungen schneller.

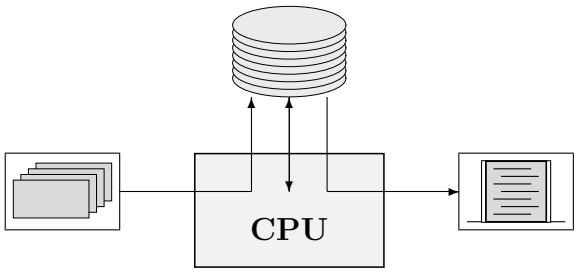
Integrierte Schaltkreise wurden entwickelt, bei denen auf einem Chip Funktionen aufgebaut waren, die zuvor aus vielen Transistoren einzeln aufgebaut werden mussten. Nicht nur konnten die Rechner dadurch räumlich kleiner werden – es war vor allem eine wesentlich schnellere Entwicklung neuer Rechner möglich, da sie nun zu großen Teilen modular aus zuverlässigen Bausteinen zusammengesetzt werden konnten.

Es gab dadurch einige wichtige Erscheinungen bei den Großrechnern:

1. Spooling

Spooling wäre sicher schon eher möglich gewesen, wurde aber erst durch die Einführung der Plattenspeicher sinnvoll.

- „Spool“ steht für „**simultaneous peripheral operation on-line**“ und ist eine Form von **Pufferung auf Platte** (statt im Hauptspeicher).
- Während ein Job sich noch in der Ausführung befindet, können bereits Teile des nächsten Jobs von Karte (oder von Band) auf die Platte übertragen werden. Der nächste Job ist dann meistens direkt nach Beendigung des vorherigen im Speicher verfügbar und direkt lauffähig.
- Es kann auch einfach ein komplettes Band aus dem Eingaberaum auf Platte überspielt werden. Dieses Verfahren heißt „staging a tape“ und war beliebt, da es das Band schonte.
- Entsprechendes gilt für die Ausgabe. Die CPU braucht hier nur das Spooling zum Drucker (oder zum Band) anzustoßen und kann sich dann die meiste Zeit bereits mit anderen Dingen (z.B. mit dem nächsten Job) beschäftigen.



- Voraussetzung für Spooling ist natürlich (wie schon beim Puffern) das Vorhandensein von Interrupt-Mechanismen bei der CPU.

Bei dem Genannten handelt es sich um quantitative Fortschritte. Durch Spooling ergaben sich aber auch qualitativ völlig neue Möglichkeiten:

- Beim Lesen von Band müssen die Jobs genau in der Reihenfolge ausgeführt werden, in der sie auf dem Band vorliegen.

  Bei Spooling liegt meist bei der Beendigung eines Jobs nicht nur ein Nachfolgejob, sondern mehrere auf Platte vor („**job pool**“). Der Monitor kann also einen Nachfolgejob **auswählen**.

- Die Auswahl kann bereits während der **Ausgabe** des **vorigen** Jobs erfolgen, wo die CPU fast (bis auf Interrupts) arbeitslos ist.

  Beispielsweise kann abgeschätzt werden, ob ein Job rechenintensiv ist oder nicht. Wenn ein Job vermutlich längere Zeit keine Ausgabe tätigt, ist es günstig, ihn bereits laufen zu lassen, während immer noch die Ausgabe des vorigen Jobs läuft.

  Ein-/Ausgabe und Berechnungszeiten können sich noch besser überlappen, d.h. die CPU kann noch besser ausgelastet werden.

- Zu beachten ist aber, dass immer noch **nur ein „eigentlicher“** Job läuft, der die CPU (rechnend) beschäftigt!

  Damit mehrere echte Jobs voneinander ungestört ablaufen können, müssen ihnen voneinander abgeschottete Speicherbereiche zugewiesen werden. Hardware für solches „**Memory Management**“ wurde erst später entwickelt.

2. Rechner-Familien und Kompatibilität

IBM „erfand“ mit dem System/360 die „Kompatibilität“. Dabei handelte es sich nicht um einen Rechner, sondern um eine ganze Familie von Rechnern, die ein großes Leistungsspektrum abdeckte:

- von Vorrechnern (langsam, preiswert, wenig Speicher, wenig Anschlussmöglichkeiten)
- bis zu Großrechnern, größer als die der vorherigen Generation (schneller, teurer, mehr Speicher, viele Anschlussmöglichkeiten)

Alle Mitglieder der Familie hatten aber dieselbe Architektur und sogar denselben Befehlssatz. Programme liefen also (theoretisch) unverändert auf all diesen Rechnern. Die parallele Entwicklung solch vieler Rechner war erst durch die IC-Technik möglich geworden.

Vorteil: Die Kunden konnten zu leistungsfähigeren Rechnern „aufsteigen“, ohne Programme oder Daten irgendwie umstellen zu müssen. Das war die Grundlage für den überwältigenden Erfolg von System/360.

Nachteil: Das Betriebssystem (OS/360) musste mit jeder Hardware zurechtkommen:

- Es nutzte oft nicht den jeweiligen Rechner optimal aus.
- Es wurde wesentlich größer als die Vorgänger-Monitore (ein mehrere Meter hoher Kartenstapel).
- Es wurde von einem sehr großen Team von Programmierern erstellt und gewartet.

Da man noch keine wirkliche Erfahrung mit Betriebssystemen hatte, gab es wenig echte Planung im voraus. Viele Fähigkeiten wurden erst spät während der Entwicklungszeit hinzugefügt.

OS/360 wurde gigantisch kompliziert und entsprechend fehleranfällig. Als Reaktion auf Beschwerden von Kunden wurden Fehler in der nächsten Version möglichst schnell „herausgehackt“ – wobei mindestens genauso viele neue Fehler hineingehackt wurden. Bereits die erste Version wurde ausgeliefert, obwohl bereits Hunderte von Fehlern bekannt waren!

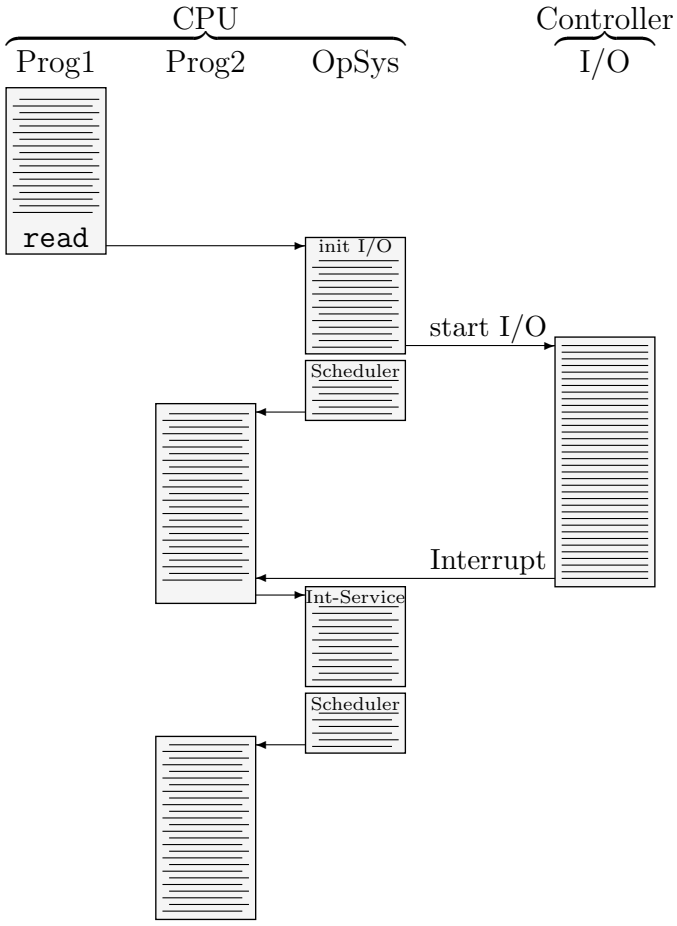
3. Multiprogramming

Die wichtigste Neuerung, die zuerst in OS/360 implementiert war, war aber das „**Multiprogramming**“, später (nachdem der Begriff „Task“ entwickelt worden war) auch „**Multitasking**“.

Die Neuerungen in den Vorgänger-Systemen zielten alleine darauf, den Durchsatz bei der CPU zu maximieren. Durch die fehlende Interaktivität verlängerte sich dagegen die Programm-Entwicklung (lange Antwortzeit auf einen Job). In der Anfangszeit der IC-Technik wurde die Hardware aber sehr schnell wesentlich billiger – und irgendwann gewannen die Löhne der menschlichen Bediener an Bedeutung.

Die Idee des Job Pools (vom Spooling) wurde ausgeweitet, und der Pool von der Platte in den Hauptspeicher verlegt. Der Hauptspeicher, der nicht fest zum Betriebssystem gehörte, wurde in mehrere Teile aufgeteilt, und in jeden Teil wurde ein Job geladen.

Um die Aufteilung durchführen zu können, musste Hardware zur Beschränkung des erlaubten Adressbereichs vorhanden sein. Außerdem war eine automatische Adress-Umsetzung sinnvoll, damit ein Programm in jedem beliebigen physischen Speicherbereich unverändert lauffähig war. Das System/360 gehörte zu den ersten Systemen, die diese Hardware-Voraussetzungen erfüllten.

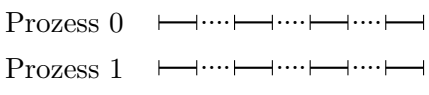


Nachdem ein Job vom Monitor die CPU-Kontrolle übernommen hatte, behielt er sie so lange, bis er wegen I/O-Operationen gezwungen war, zu warten. Für diese Operationen musste er ja ohnehin den Monitor aufrufen – und dieser hatte nun die Möglichkeit, zu entscheiden, lieber einem rechenintensiven Job den Vorrang zu geben.

Die Umschaltung zwischen den Jobs erfolgte also *nicht kontinuierlich* (per Zeitscheiben-Verfahren), sondern nur bei Wartezeiten des jeweils aktiven Prozesses. Diese einfachere Art von Multitasking nennt man auch „**non-preemptive**“ (das Zeitscheiben-Verfahren heißt „preemptive“).

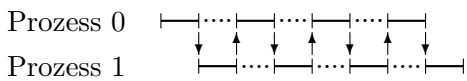
Beispiel: Um die mögliche verbesserte CPU-Ausnutzung zu demonstrieren, betrachten wir als einfache Situation zwei Prozesse, die zwischendurch immer wieder auf Eingaben warten müssen.

In dieser Zeit sind sie „idle“ (untätig). Rechts sind die Arbeitszeiten mit durchgezogenen Linien, die Idle-Zeiten mit Punkten dargestellt.



In einem System ohne Multitasking sind die Prozesse nur Programme, die nacheinander ausgeführt werden. Die Gesamt-Laufzeit ist die Summe der Einzel-Laufzeiten der Prozesse, inklusive ihrer Idle-Zeiten.

Auch bei Multitasking ohne Zeitscheiben ergeben sich Verbesserungen. Wenn ein Prozess warten muss, wird der andere vorgelassen.



Die Pfeile deuten den Wechsel der CPU-Kontrolle an.

4. Timesharing

Timesharing ist im Wesentlichen Multiprogramming mit mehreren Benutzern, die gleichzeitig an mehreren Terminals arbeiten können.

Die meisten Jobs werden „fast immer“ auf Benutzer-Eingaben warten müssen. In dieser Zeit können dann die Jobs laufen, die sich gerade in einer rechenintensiveren Phase befinden.

Vorteil: Es ist wieder interaktive Programmentwicklung möglich. Programme können ediert werden, während die anderer Benutzer laufen. Das Programm kann direkt vom Benutzer Eingaben per Tastatur erhalten (nicht über einen vorproduzierten Kartenstapel). Dadurch ist ein einfacheres und schnelleres Austesten (und damit eine schnellere Programmentwicklung) möglich.

Nachteil: Die Benutzeranzahl ist potentiell unbegrenzt. Bei sehr vielen Benutzern gleichzeitig erhält man leicht einen Effekt, der als „**Thrashing**“ bezeichnet wird (thrash = sich vorwärtsquälen, ein nautischer Begriff). Das System ist dann fast nur noch mit dem Mechanismus des Umschaltens zwischen den Benutzern beschäftigt. Eventuell reicht ein einziger zusätzlicher Benutzer, um die Antwortzeiten zu vertausendfachen.

Timesharing wurde zuerst 1962 (als CTSS) auf einer IBM 7094 entwickelt, also auf einer Hardware ohne Möglichkeit zur Memory Protection. Es wurde also erst später wirklich einsetzbar.

Im Zusammenhang mit Timesharing wurden wichtige Konzepte entwickelt, die bis heute eher noch an Bedeutung gewonnen haben:

Shell: Befehlsinterpreter wurden entwickelt, die eine direkte Schnittstelle vom Benutzer zum System darstellen. Sie leisten im Grunde dasselbe wie der Control Card Interpreter in den alten Monitoren, werden jetzt aber interaktiv an den Benutzerterminals eingesetzt.

Dateisystem: Die Ein- und Ausgabedaten-Sammlungen waren früher in Form von mehreren Kartenstapeln bzw. des Control-Card-Aufbaus der Jobs strukturiert. Diese Struktur wurde auf Band bzw. Platte einfach übernommen. Der Bedarf für ein Dateisystem ergab sich wirklich erst jetzt:

- Unter Multiprogramming kann es gleichzeitig mehrere Sammlungen von Ein- und Ausgabedaten geben, denen physisch ein bestimmter Platz zugeordnet werden muss (**Dateien**).
- Unter Timesharing arbeiten mehrere Benutzer am selben System. Ihre Daten auf der Systemplatte müssen voneinander abgeschirmt werden. Der gesamte Dateibestand wird also in Clustern organisiert (**Directories**, Verzeichnisse).

Eine echte Verzeichnis-*Hierarchie*, also beliebig tief verschachtelte Unterverzeichnisse für jeden Benutzer, wurde erstaunlicherweise bei vielen Systemen nicht oder erst nachträglich viel später implementiert.

- Viele Rechner dieser Zeit waren gemischte Batch- und Timesharing-Systeme. OS/360 war beispielsweise eigentlich ein Batch-System, das mit „TSO“ (timesharing option) modifiziert werden konnte.
- Eine wichtige andere Rechner-Familie waren die PDP-1 bis PDP-11 von DEC. Alle Rechnertypen waren einander zwar ähnlich, aber miteinander inkompatibel. Sie waren „kleiner“ als die durchschnittlichen Rechner dieser Zeit (die ersten „Minicomputer“), boten aber ein gutes Preis-/Leistungs-Verhältnis.

1.2.4 Mikroprozessoren

Als sich die Dichte von integrierten Schaltkreisen immer weiter erhöhte (ab LSI = large scale integration), war es möglich, den kompletten Kern einer CPU auf einem Chip zu vereinigen. Rechner der Minicomputer-Klasse konnten zu wesentlich niedrigeren Preisen angeboten werden. In vielen Bereichen ging daher die Entwicklung weg von einem zentralen Großrechner mit Terminal-Zugang. Das war der Beginn der Ära des PC (personal computer).

Die entsprechenden Betriebssysteme mussten sich jetzt nicht an Operatoren, sondern an reine Bediener ohne tiefe Rechner-Kenntnisse wenden. Dadurch wurden sie zwangsweise benutzerfreundlicher.

Durch das Wegfallen der Mehrbenutzer-Struktur wurde die Hardware zunächst wieder vereinfacht (und dadurch verbilligt) – beispielsweise wurden Memory Protection und privilegierte Befehle wieder fallengelassen. Entsprechend wurden auch die Betriebssysteme vereinfacht, wobei leider auch die Mehrprozess-Elemente eliminiert wurden.

Mit zunehmender Integration und Verbilligung der Hardware – und durch den Ruf nach Benutzer-Komfort – fanden diese Elemente aber nach und nach wieder Eingang in Hardware und Betriebssysteme.

Wesentliche *konzeptionelle* Neuerungen haben in dieser Zeit dagegen nicht stattgefunden. Erst durch die **Vernetzung** der Personal Computer und die entstehenden Probleme im Hinblick auf Sicherheit und Zugriff auf entfernte Ressourcen gab es neue Entwicklungen (Netzwerk-Betriebssysteme, verteilte Betriebssysteme).

1.3 Beispiele für Betriebssysteme

Wir werden uns im weiteren auf drei Betriebssysteme stützen, wenn es darum geht, die praktische Verwirklichung der besprochenen Konzepte zu untersuchen: MS-DOS, Windows und vor allem UNIX. Alle drei sind auf Intel-PC-kompatiblen Rechnern verfügbar und verbreitet (UNIX vor allem in Form des frei vertreibbaren L*i*nux).

1.3.1 MS-DOS

Der Name DOS für Disk Operating System mag zwar ein wenig altertümlich klingen, trifft den Kern des Systems aber recht gut. Es handelte sich ursprünglich hauptsächlich um eine Sammlung von Routinen zum Laden und Speichern von Daten auf Disketten und zum Starten von Programmen.

MS-DOS und Windows sind gute Beispiele für Betriebssysteme, die nicht systemunabhängig definiert sind, und die ursprünglich fest für eine wesentlich andere Hardware entworfen wurden als die, auf der sie mittlerweile laufen müssen. Sie sind im Lauf der Jahre natürlich gewachsen und haben eine große Zahl seltsamer Hilfskonstruktionen hervorgebracht, um immer noch zu ihren Ursprüngen kompatibel zu sein.

Ein kurzer Überblick über die Geschichte von MS-DOS:

vor 1980 Viele Mikrocomputer für den Heimgebrauch basieren auf dem 8-Bit-Mikroprozessor 8080 von Intel. (Einer der ersten war übrigens der Altair (1975) mit 256 Bytes Hauptspeicher, für den Bill Gates einen BASIC-Interpreter schrieb.) Auf einem Großteil dieser Modelle läuft das Betriebssystem CP/M von Digital Research.

1980 IBM plant einen 16-Bit-Mikrocomputer mit dem 8086 als Mikroprozessor. CP/M-86 hat arge Entwicklungsschwierigkeiten, und so kommt man über die Lizensierung seines BASIC-Interpreters mit Bill Gates' neugegründetem Unternehmen Microsoft weiter ins Geschäft und gibt ein einfaches und kompaktes Betriebssystem in Auftrag.

Microsoft kauft ein bereits fast fertiges Projekt namens 86-DOS (von Seattle Computer Products, Programmierer Jim Paterson), das sich stark an CP/M anlehnt, und auf das CP/M-Programme vergleichsweise leicht portierbar sind. Von CP/M her kommen beispielsweise die Dateinamen im Format 8+3 Zeichen, ausführbare (nicht relozierbare) Programme mit der Endung `.COM` („Command“) und die Laufwerksbezeichnungen `A:` etc.

Nur ein 160 KByte-Laufwerk für $5\frac{1}{4}$"-Disketten wird unterstützt. Das Dateisystem hat nur eine Ebene – es gibt also keine Verzeichnisse und Unterverzeichnisse.

1981 Im August kommt zusammen mit dem IBM PC das weiterentwickelte System unter dem Namen MS-DOS 1.0 auf den Markt. Es ist 12 KByte groß und aus 4000 Assembler-Zeilen entstanden.

1982 Die Version MS-DOS 1.1 erscheint im Oktober.

Es werden logische Geräte wie `CON`, `PRN` und `AUX` eingeführt, die wie Dateien behandelt werden können, aber einen Geräte-Treiber und letztlich ein I/O-Gerät ansprechen.

Es sollen zwar nicht mehrere Programme gleichzeitig ablauffähig sein, aber doch gleichzeitig im Speicher liegen können (nachgeladene Treiber, residente Programme, etc.). Da das mit den alten `.COM`-Dateien nicht möglich ist, wird das neue Format `.EXE` eingeführt.

Aufgrund der damaligen Speicherknappheit wird das System aufgesplittet in einen residenten Teil, der nach dem Booten permanent im Speicher verbleibt, und transiente Teile, die nur bei Bedarf nachgeladen und wieder entfernt werden können. Dazu gehören der Befehlszeilen-Interpreter und Befehle wie `FORMAT`, `CHKDSK`, etc.

Auf Wunsch von IBM wird eine Art Batch-Verarbeitung in Form der `.BAT`-Dateien eingebaut. Dennoch ist das Dateisystem immer noch flach.

1983 IBM kündigt den XT an, der standardmäßig mit einer Festplatte ausgestattet ist (10MB). Das flache Dateisystem ist dadurch nicht mehr haltbar, und MS-DOS 2.0 erhält ein hierarchisches Dateisystem, das stark an UNIX angelehnt ist. Viele Systemaufrufe werden von UNIX übernommen. Unter anderem, um nach außen DOS von UNIX abzugrenzen, wird als Trennsymbol in Dateipfaden absichtlich der Backslash '\' statt des Slash '/' von UNIX eingeführt.

Nachladbare Gerätetreiber (wie `ANSI.SYS`) werden endgültig Bestandteil des Systems. Einfaches Spooling (Ausdrucken im Hintergrund) wird möglich – allerdings jeweils nur in dem Zeitraum, wo das Vordergrund-Programm auf ein I/O-Ergebnis warten muss.

1984 IBM kündigt den AT mit dem 80286 an. Der Prozessor bietet Hardware-Unterstützung für 16 MByte Hauptspeicher und Multitasking, wovon Microsoft in der neuen MS-DOS-Version aber keinen Gebrauch macht. MS-D0S 3.0 unterstützt lediglich HD-Disketten, größere Festplatten und ist ein wenig schneller.

1987$^+$ In der Folgezeit erscheinen Versionen bis MS-DOS 3.3 (1987), 4.0 (1988), 5.0 (1991) und 6.0 (1993), die an der Systemstruktur nichts mehr verändern. Das System wird nur erweitert um rudimentäre Netzwerk-Unterstützung (z.B. gemeinsam benutzte Files), Unterstützung neuerer Floppies, größerer Harddisks, etc.

Besonders begehrt sind diverse Hilfskonstruktionen zur Umgehung des unsäglichen Speicherengpasses. Erst die 1991 erscheinende Version 5.0 macht – 7 Jahre nach Erscheinen des 80286 – einigermaßen Gebrauch von mehr als 1 MByte Speicher.

1.3.2 Windows

Microsoft Windows in den Versionen 1.0 bis 3.11 (also nicht NT und vor Windows 95) war kein eigentliches Betriebssystem, sondern ein „Aufsatz“ auf MS-DOS, eine grafische „Betriebsumgebung“.

Die eigentlichen Elemente wie Fenster, Icons, Menüs etc. sind natürlich keine Microsoft-Entwicklung. Eine Fenster-Schnittstelle war schon 1981 bei Xerox entwickelt worden (im Palo Alto Research Center, PARC). Windows war in wesentlichen Teilen von der Oberfläche abgeschaut, die Apple bereits auf seinen Lisa- (1983) und Macintosh-Rechnern (1984) implementiert hatte.

Auf Prozess-Ebene bietet Windows sogenanntes kooperatives Multitasking. Es können zwar gleichzeitig mehrere Prozesse im Speicher liegen – die CPU wird jedoch nicht vom System auf sie verteilt. Ein Prozess muss freiwillig explizit die CPU-Kontrolle aufgeben, bis ein anderer an die Reihe kommt.

Die Kommunikation von Anwenderprogrammen mit dem Fenstersystem erfolgt über ein API (Application Program(ming) Interface), also eine Sammlung von Funktionen in Form von Bibliotheken, die normal wie Unterprogramme aufgerufen werden.

Ein kurzer Überblick über die Geschichte von Windows:

1983 Als Antwort auf Apple's Lisa kündigt Microsoft eine fenster-orientierte Benutzeroberfläche für PCs an.

1985 Windows erscheint im November, gleich mit der Versionsnummer 1.1. Es ist aber ein dilettantischer Abklatsch der Apple-Oberfläche. Fenster dürfen einander nicht überlappen, sondern müssen vollständig nebeneinander oder übereinander liegen.

1987 Windows 2.0 erscheint mit überarbeitetem „Look and Feel". Inzwischen ist OS/2 als geplanter Nachfolger von MS-DOS erschienen, seine grafische Oberfläche, der „Presentation Manager" ist aber noch nicht fertig. Die neue Windows-Oberfläche lehnt sich schon einmal an den PM an, um später einen leichten Übergang zu OS/2 zu ermöglichen.

1988 Windows 2.1 erscheint in den beiden Versionen Windows/286 und Windows/386. In der 386-Version können mehrere DOS-Anwendungen sicher in virtuellen Maschinen laufen.

1990 Microsoft überlässt IBM die Weiterentwicklung von OS/2 und baut Elemente des OS/2 Presentation Managers nun in sein eigenes Produkt ein. Windows 3.0 verschmilzt wieder die 286- und 386-Versionen. Es ist das erste Windows, das als Plattform für sinnvolle größere Anwenderprogramme dienen kann. Es kann ein Arbeitsspeicher von 16 MB angesprochen werden. Man kann zwischen mehreren DOS- und Windows-Anwendungen umschalten. Programm- und Filemanager entstehen. Windows-Anwendungen älterer Versionen führen nun allerdings regelmäßig zum Absturz.

1991 Windows 3.1 unterstützt nur noch Prozessoren vom 80286 aufwärts. Wichtigste Neuerungen sind TrueType-Fonts und OLE (Object Linking and Embedding, zur Ressourcen-Teilung zwischen mehreren Programmen).

Windows NT ist das „High-End"-System von Microsoft (NT=„New Technology"). Es ist nicht allein auf Intel-80x86-Prozessoren festgelegt und im Wesentlichen nicht für Normalbenutzer, sondern für Serversysteme und Netzwerke gedacht. Entsprechend versucht es, besonderen Wert auf Sicherheitskonzepte zu legen.

Mit NT wird das sehr umfangreiche 32-Bit-API Win32 eingeführt, das ausschließlich für 32-Bit-Programme gedacht ist. Später gibt Microsoft die stark abgespeckte Version Win32s für Windows 3.1 heraus.

1995 Windows 95 läuft nur noch im erweiterten Modus, also auf Systemen mit 80386 aufwärts. Es unterstützt einen Großteil von Win32 (manchmal als Win32c bezeichnet).

1998 Windows 98 erscheint als kostenpflichtiges Bugfix von Windows 95.

1.3.3 UNIX

Hier sind viele der noch zu besprechenden Konzepte relativ vollständig verwirklicht. Die meisten der komplexeren Beispiele werden sich daher unter UNIX abspielen.

Außerdem steht mit L*i*nux eine „Version" von UNIX jedem Besitzer eines PC-kompatiblen Rechners (ab 80386) frei zur Verfügung. Es bezeichnet sich selbst nicht als ein UNIX, deckt

aber fast alle Spezifikationen von POSIX und UNIX System V ab. Es ist ein komplett neu geschriebenes System, für das kein Code anderer Versionen übernommen wurde.

Ein wesentlicher Vorteil von L*i*nux ist in unserem Zusammenhang, dass es vollständig im Quellcode (`C`, ein wenig Assembler) vorliegt. Wir wollen aber nur diese Implementation studieren und nicht Änderungen am Kern vornehmen und das System neu übersetzen.

L*i*nux gibt es mittlerweile auf diversen anderen Rechnerarchitekturen als Intel-x86. Wir werden auch nur selten auf prozessorabhängige Gegebenheiten eingehen müssen.

Ein kurzer Überblick über die Geschichte von UNIX:

1963 Das MIT (Massachusetts Institute of Technology), Packard Bell und GE (General Electrics) kündigen einen Rechner zusammen mit einem Betriebssystem an, das Hunderte oder Tausende von Benutzern bedienen können soll. Der Name des Systems ist **MULTICS** (MULTiplexed Information and Computing Service). Die Entwickler sehen den Rechner als Analogon eines E-Werks, und jeder Einwohner soll nur den Stecker in die Wand stecken müssen... Der Rechner soll modular fast unbegrenzt ausbaubar sein, und Teile sollen abgeschaltet und gewartet werden können, ohne die Funktionsfähigkeit des Systems zu beeinträchtigen.

Es gibt leider herbe Entwicklungsschwierigkeiten. Man verspricht sich von `PL/I` Vorteile gegenüber `FORTRAN` und beginnt, das System in dieser Sprache zu schreiben – allerdings verspätet sich der `PL/I`-Compiler immer wieder und ist schließlich sehr fehlerhaft. Außerdem stimmt einfach das Verhältnis der geplanten Benutzer-Anzahl mit der tatsächlichen Hardware nicht.

MULTICS führt deshalb wichtige Konzepte ein, läuft aber erst 1969 halbwegs brauchbar nur auf einigen Laborrechnern (GE645) beim MIT. Die Bell Laboratories steigen ganz aus der Entwicklung aus. MULTICS wird nie kommerziell eingesetzt.

1969 Zwei der MULTICS-Entwickler, Ken Thompson und Dennis Ritchie, sollen bei den Bell-Laboratories ein verbessertes Dateisystem für die GE645 entwickeln. Auf diesem Rechner entsteht als Nebeneffekt aber schon ein kleiner Systemkern.

Als Thompson eine unbenutzte PDP-7 findet, portiert er Kernel und Dateisystem auf diesen Rechner (in Assembler). Das Dateisystem entspricht bereits ziemlich genau dem heutigen, mit hierarchischer Directory-Struktur in einem einheitlichen Dateibaum.

Es entsteht ein System, das zunächst nur für einen Benutzer gedacht ist. Brian Kernighan nennt das Gebilde daher scherzhaft UNICS (U für Uniplexed).

1971 Das System wird auf die größere PDP-11 portiert (Daten: 16 KB System, 8 KB Platz für Benutzerprogramme, 512 KB Plattenkapazität, maximale Dateilänge 64 KB).

Erste Compiler sollen auf dem System entstehen. Thompson konnte sich mit `FORTRAN` nicht anfreunden und entwickelt die Sprache `B` (angelehnt an das schon existierende `BCPL`=binary coded programming language, eine Abart von `CPL`). In `B` gab es als einzigen Datentyp Integer (und eine Art typlosen Pointer). Dennis Ritchie erweitert die Sprache um Datenstrukturen und einiges mehr und nennt sie `C`.

1973 Fast das ganze System wird in `C` umgeschrieben (bis auf sehr kurze maschinennahe Teile in Assembler). Dadurch wird es zwar minimal größer und langsamer als die Assembler-Vorgänger – dafür ist nun die Voraussetzung für die schnelle Weiterentwicklung und die Verbreitung auf andere Rechnersysteme geschaffen.

UNIX wird zum ersten Mal von anderen Benutzern als Thompson und Ritchie eingesetzt (25 Leute bei den Bell Laboratories).

Da AT&T selbst kommerziell keine Rechner oder Software vertreiben darf (Monopol-Aufsicht), wird die UNIX-Lizenz sehr günstig an viele Universitäten in den USA vergeben. Da dort viele PDP-11 verwendet werden, kann sich das System sehr schnell verbreiten. Viele neue Ideen können schnell in das System integriert werden, was vor allem dadurch begünstigt wird, dass der komplette Sourcecode zur Verfügung steht.

Um diese Zeit entsteht das Text-Formatierungsprogramm `troff` (zunächst `roff`), das (bis zum Durchbruch von TEX) der Standard für Textverarbeitung unter UNIX ist.

1977 UNIX wird zum ersten Mal auf einen Rechner außerhalb der PDP-Serie portiert (Interdata 8/32). Einige rechnerabhängige Parameter (16-Bit-Integers, Registeranzahl) waren an diversen Stellen hardcodiert worden, sodass einige Bereinigungsarbeit nötig ist.

Steve Johnson entwickelt den „*portable C compiler*“, der so angepasst werden kann, dass er Maschinencode für einen fast beliebigen Rechner produzieren kann. Er erleichert das Portieren auf weitere Architekturen erheblich.

Der erzeugte Code muss jeweils auf Bändern auf den anderen Rechner transportiert werden. In dieser Zeit entstehen daher auch rudimentäre UNIX-Mechanismen zur Vernetzung von Rechnern.

An der Universität von Kalifornien in Berkeley (unterstützt vom amerikanischen Verteidigungsministerium) wird (aus Version 6) eine Variante namens BSD entwickelt (Berkeley Software Distribution, erste Version 1BSD, erhebliche Erweiterungen ab Version 4.1BSD). In Berkeley werden unter anderem der Editor `vi` und die Shell `csh` entwickelt.

1979 Microsoft entwickelt eine UNIX-Version namens XENIX für Mikrocomputer, die vom ursprünglichen UNIX relativ stark abweicht. AT&T geben „UNIX Time-Sharing-System Version 7“ heraus, mit vielen Verbesserungen wie Unterstützung für sehr große Dateien, ein erweitertes `C` und eine neue Shell.

1983 Viele Hersteller portieren UNIX auf ihre Maschinen, wodurch viele unterschiedliche Versionen entstehen. AT&T schuf 1982 eine verschmolzene Version (mit XENIX-Elementen) namens System III. Die Version IV als Zwischenstadium betrachtet und nur intern verwendet, sodass das nächste öffentliche Release 1983 das System V ist.

1984 Das IEEE (Institute for Electrical and Electronic Engineers) versucht sich mit POSIX (Portable Operating System Interface uniX) an einer Standardisierung des Systems.

1985 AT&T gibt die SVID (System V Interface Definition) heraus, eine Festlegung der Systemaufrufe, Bibliotheksroutinen und Hilfsprogramme, die ein UNIX beinhalten muss, das sich System-V nennt. Sie wird von BSD ignoriert.

1988 Mit System V.4 (Release 4, SVR4) finden viele BSD-Elemente (4.3 BSD) Eingang in das System V.

1990 Das IEEE gibt die neue Version 1 der POSIX-Standards heraus. Systemaufrufe, die in System V *und* BSD vorkamen, werden meist nach POSIX übernommen. Die meisten heutigen UNIXe (aber auch andere Betriebssysteme) implementieren eine *Obermenge* von POSIX.

1991 Die erste Version des L*i*nux-Kernels wird im Internet verbreitet. Der finnische Informatik-Student Linus Torvalds hatte einige Zeit vorher mit einer völlig neuen Implementation eines UNIX-ähnlichen Systems für Intel-PCs begonnen (es orientiert sich an POSIX, System V *und* BSD). Die Weiterentwicklung erfolgte dagegen über das gesamte Netz verteilt.

UNIX leidet noch immer unter einer fehlenden Vereinheitlichung.

Die Open Software Foundation (OSF) mit Mitgliedern wie IBM, DEC und HP standardisierten OSF/1, das stark an 4.3BSD angelehnt ist. Daraufhin gründeten AT&T, SUN und einige andere Hersteller Unix International (UI) und setzen in ihren Systemen schwerpunktmäßig auf System-V-Elemente (SVR4, SunOS).

System V Release 4 ist vollständig POSIX-kompatibel, implementiert aber als Kompatibilitäts-Service Bibliotheken und Kommandos nach 4.3BSD-Standard. BSD grenzt sich dagegen weiterhin weitgehend ab.

L*i*nux mischt SVR4-, POSIX- und BSD-Elemente. Es ist natürlich gewachsen und nicht am Reißbrett entstanden. An einigen Stellen werden wir daher Maschinenbefehle im `C`-Code, `goto`s, hardcodierte Konstanten und statische System-Strukturen finden – Dinge, die einem Software-Ingenieur eigentlich die Haare zu Berge stehen lassen. Diese Dinge sind aber mit neueren Versionen allmählich im Verschwinden begriffen.

Wer bereits einmal auf Entdeckungsreise gehen möchte – die Quellen von L*i*nux liegen üblicherweise im Verzeichnis `/usr/src/linux`. Es gibt u.a. folgende Unterverzeichnisse:

`init`	Initialisierung des Systems (der Kern startet beispielsweise mit der Routine `start_kernel` in `init/main.c`)
`include`	Header-Dateien für den Kernel (`asm` Assembler-nah, `linux` für den reinen `C`-Teil)
`drivers`	Hardware-Treiber (Unterverzeichnisse `block` für plattenähnliche Geräte, `char` für zeichenorientierte Geräte, `sound`, `scsi`, etc.)
`fs`	Filesystem (L*i*nux unterstützt viele verschiedene Dateisysteme, Unterverzeichnisse u.a. `ext2`=neues Standard-Dateisystem, `msdos`, `nfs` für Network File System, `minix`, etc.)
`mm`	Memory Management, Speicherverwaltung (virtueller Speicher, Paging, etc.)
`ipc`	System-V Inter-Prozess-Kommunikation

2 Systemarchitekturen

In diesem Kapitel besprechen wir gesammelt kurz einige Grundlagen zu Strukturen von Rechnersystemen (der Hardware und der Software), deren Kenntnis beim Verständnis der Betriebssysteme hilfreich sein wird.

2.1 Von-Neumann-Architektur

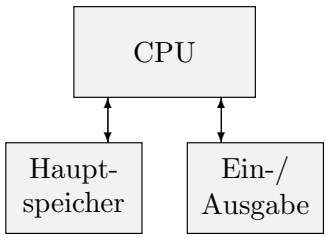
Die meisten heute gängigen Rechnersysteme basieren auf einem Konzept, das zuerst von John von Neumann in den vierziger Jahren formuliert wurde („*Preliminary Discussion of the Logical Design of an Electronic Computing Instrument*“).

Die drei Komponenten eines solchen Systems sind im Bild rechts dargestellt:

CPU: Central Processing Unit, Prozessor, Steuer- und Recheneinheit

Hauptspeicher: beinhaltet Programme, Daten und Hilfsstrukturen

Ein-/Ausgabe (I/O): ggf. mehrere Einheiten zur Kommunikation mit der Außenwelt (Tastatur, Drucker, etc.)



- Wenn das Design aus Effizienzgründen erweitert wird, können diverse Hilfskomponenten hinzukommen – das grundlegende Konzept bleibt aber unverändert.

- Die CPU wird intern manchmal noch in Funktionsgruppen aufgeteilt: Das Steuerwerk dient der Steuerung der CPU-internen Abläufe, das Rechenwerk ist für reine Rechenoperationen (arithmetische/logische) zuständig. In realer Hardware sind beide aber stark miteinander verzahnt.

  Manchmal wird abweichend auch unsere „CPU“ nur als Prozessor und die Gesamtheit von CPU, Speicher und I/O-Steuerung als CPU bezeichnet.

- Wesentliches Merkmal dieser Architektur ist, dass auf der Hardwareebene nicht zwischen Daten und Programmen unterschieden wird. Ältere Systeme waren entweder nicht programmierbar (fest verdrahtet), fest programmierbar (mit Schaltern), oder es gab jedenfalls eine deutliche Trennung zwischen Programmierungs-Informationen und numerischen/logischen Daten.

  Dadurch wird der Hauptspeicher allerdings besonders stark beansprucht. Die Ausführung eines einzigen Befehls kann diverse (und unterschiedlich geartete) Speicherzugriffe auslösen. Daher wird der Hauptspeicher auch als von-Neumannscher Flaschenhals bezeichnet. Seine Geschwindigkeit und Organisation bestimmt zu einem großen Teil die Effizienz des ganzen Systems.

2.2 I/O-Strukturen

2.2.1 Typen von Geräten

Es gibt grundlegend unterschiedliche Typen von I/O-Geräten, die am günstigsten auch unterschiedlich vom System angesprochen werden.

zeichenorientiert: liefern/akzeptieren einen kontinuierlichen Strom von Zeichen/Bytes/Worten, je eines pro I/O-Operation
Beispiele: Tastatur, Drucker, Magnetband, Modem, Text-Terminal

blockorientiert: das Medium ist in Cluster (Blöcke) aufgeteilt, von denen jeder zu jeder Zeit angesprochen werden kann
Beispiele: Platte, CD u.ä.

Beachte: Die zeichenorientierten Geräte können auf höherer Ebene durchaus „gepuffert“ sein, d.h. das Betriebssystem schreibt nicht jedes Zeichen einzeln heraus, sondern sammelt zunächst z.B. 1 KByte im Speicher. Das darf man nicht mit dem Verhalten blockorientierter Geräte verwechseln, die von der Hardware her nur blockweise angesprochen werden können!

Nicht in dieses Schema passen z.B. die meisten Grafik-Systeme (liefern kontinuierlich ein Hauptspeicherabbild) und Timer (ihre Ausgabe besteht in Interrupt-Signalen).

2.2.2 Controller

Der eigentliche I/O am Peripherie-Gerät wird durch passende Hardware, den **Controller** durchgeführt. Er ist bei Mikrocomputern meistens auf einer Karte untergebracht, auf der üblicherweise ein eigener (einfacher) Mikroprozessor sitzt. Das eigentliche Gerät wird an diesen Controller angeschlossen, die CPU kommuniziert mit dem Controller.

- Die Kommunikation zwischen Controller und Gerät wird durch eine spezielle Schnittstellen-Beschreibung festgelegt. Eventuell (und hoffentlich) passen also verschiedene Geräte an einen Controller, vorausgesetzt, sie halten sich an diese Beschreibung (z.B. AT-Bus-Festplatten, etc.).

  Das übertragene Datenformat ist meist sehr simpel: z.B. ein einfacher Vorspann (bei Platten Zylinder/Spur/Sektor), die eigentlichen Daten, gefolgt von einer Checksumme.

- Der Controller arbeitet die Daten für die weitere Benutzung auf und speichert sie zunächst in einem Controller-internen Pufferspeicher (analog wird beim Schreiben erst dieser Puffer gefüllt und dann sein Inhalt auf das Medium geschrieben). Er muss außerdem auf Fehler reagieren (z.B. durch weitere Lese-Versuche bei Fehlersignalen der Hardware oder Checksummen-Fehlern) und erst bei „ausweglosen“ Situationen diesen Fehler nach außen weitergeben.

CPU und Controller kommunizieren über sogenannte **Register**. Dabei handelt es sich um Speicherstellen im Controller, die von der CPU aus je nach System unterschiedlich angesprochen werden (s.u.).

Man benötigt im allgemeinen vier verschiedene Arten von Registern:

Kontroll-Register (nur schreiben):
zum Absetzen der verschiedenen Kommandos, z.B. bei einem Floppy-Controller: Block lesen/schreiben/formatieren, neu justieren, etc.

Status-Register (lesen/schreiben):
Lesen: ist das Gerät lese-/schreibbereit, ist ein Fehler aufgetreten?
Schreiben: Kommando ausführen (manchmal reicht dazu auch schon allein das Schreiben in das Kontroll-Register)

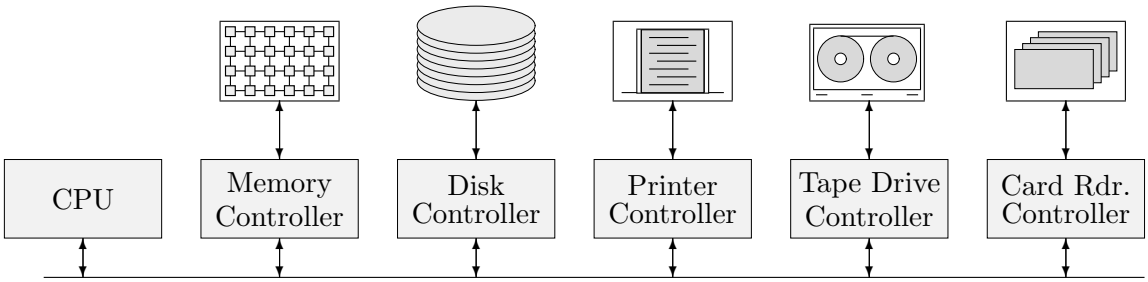
Data-in (nur lesen):
ein Datum (ein Byte/Wort) vom Controller(-speicher) in die CPU lesen

Data-out (nur schreiben):
ein Datum von der CPU in den Controller(-speicher) schreiben

Eventuell gibt es noch Modifikations-Register, die „Parameter“ für den abgesetzten Befehl aufnehmen.

2.2.3 Busse

Die Verbindungen zwischen den Komponenten im ursprünglichen von-Neumann-Design haben sich im Lauf der Zeit zu einer Sammlung von sogenannten „**Bussen**“ entwickelt, also Sammelleitungen, an die sehr viele Bauelemente angehängt werden können:

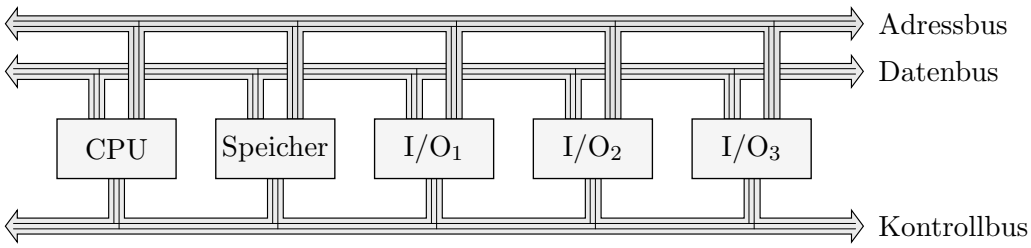


Auf diese Weise ist das System leichter zu erweitern und neu zu konfigurieren, als wenn fest definierte Hardware für die Kopplung von Gerät zu Gerät vorgegeben wird. Es muss natürlich Vorsorge getroffen werden, dass sich jeweils das richtige Gerät „angesprochen“ fühlt (z.B. mit Adressen, also einer Nummerierung).

Meistens gibt es drei getrennte Bus-Systeme:

Adressbus: transportiert die Adresse der anzusprechenden Zelle (im Hauptspeicher, in einem I/O-Bereich o.ä.)
Datenbus: transportiert Datenwörter (in den Speicher, aus dem Speicher, zum/vom I/O-Gerät, etc.)
Kontroll-Bus: transportiert Steuersignale (Speicher lesen/schreiben, Daten übernehmen, Interrupt auslösen, etc.)

Jeder Bus besteht üblicherweise aus einem Bündel von Signalleitungen mit digitalen Daten. Die Anzahl der Leitungen heißt **Breite** des Busses. Systeme mit 32 Datenleitungen können beispielsweise ganze Zahlen in $[0, 4\,294\,967\,295]$ in einem Schritt übertragen, mit 32 Adressleitungen lassen sich 4 GByte Hauptspeicher ansprechen, etc.



Über diese Busse sind für die CPU nun die Register der Controller-Bausteine erreichbar:

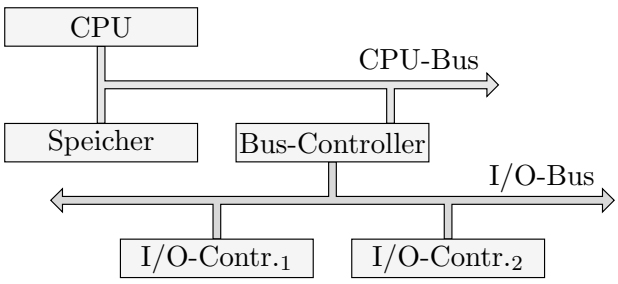
- Eventuell gibt es getrennte Busse für den Komplex CPU/Hauptspeicher und die Verbindung der CPU mit den I/O-Geräten. Letzterer wird dann „*dedizierter I/O-Bus*“ genannt.

  Bei der 80x86-Familie werden I/O-Geräte beispielsweise mit speziellen Prozessor-Befehlen (`in` und `out`) angesprochen. Die Adressierung erfolgt zwar auch über den Adressbus, es wird aber eine spezielle Leitung im Kontrollbus belegt, um einen I/O-Zugriff zu kennzeichnen. Der entstehende Adressbereich heißt auch **I/O-Bereich** (I/O-Ports).

- Keinen dedizierten I/O-Bus gibt es beispielsweise bei der 680x0-Familie. Dort wird den I/O-Geräten ein Teil des normalen Adressbereichs zugeordnet – sie werden in den Adressbereich „eingeblendet“. Sie können dann mit normalen Speicher-Operationen (`move`, etc.) angesprochen werden. Solche Geräte (bzw. Register) heißen „*memory mapped*“.

Mikrocomputer besitzen meist nur einen (oder eben zwei) Bus-Systeme. Großrechner benutzen dagegen meist mehrere Busse für I/O-Geräte unterschiedlicher Klassen, I/O-Subsysteme mit einer eigenen komplexen CPU, wobei sich kompliziertere Topologien als die linearen Busse ergeben.

Auch auf Mikrocomputern gibt es allerdings sogenannte Bus-Controller, also Controller, die einen weiteren Bus verwalten, der nicht direkt an die CPU-Busse angeschlossen ist.



Dadurch wird die CPU von Standard-Aufgaben (Interrupt- und DMA-Behandlung, s.u.) entlastet. Eventuell kann die Kommunikation zwischen zwei Geräten ganz ohne Umweg über die CPU ausgeführt werden („*bus mastering*“).

Außerdem können so Geräte an ganz unterschiedliche Rechnersysteme angehängt werden, vorausgesetzt, der Bus-Controller hält sich an einen Standard. Die wichtigsten solchen Standards (bei PCs und UNIX) sind **ISA**/**EISA**, **PCI** und **SCSI**. Die Controller für erstere sind üblicherweise bereits auf der Hauptplatine des Rechners untergebracht.

2.2.4 Polling

Um eine möglichst gute Auslastung des Gesamtsystems zu erreichen, sollten sich I/O-Operationen (verschiedener Geräte) und eigentliche Rechenzeit der CPU gut überlappen. Dafür waren zunächst keine guten Hardware-Voraussetzungen gegeben:

Der I/O (genauer: die Bedienung der Controller-Register) wurde fast vollständig von der CPU übernommen. Die Geräte sind aber nun meist wesentlich langsamer als die CPU, sodass sich diese meist in einem Zustand namens „**busy waiting**“ befand.

Es musste immer auf Eingabedaten von den viel langsameren Eingabegeräten (Kartenleser, Magnetbänder) gewartet werden oder darauf, dass Ausgabegeräte die geschickten Daten verarbeitet hatten. Daher stammt die Bezeichnung „**Polling**“: die CPU musste permanent den Status des angesprochenen Geräts *nachfragen*, in Pseudo-Code etwa wie folgt (lesend):

```
weiter=true;
do
 {
   do
      lies Controller-Status;
   while Status==”nicht bereit”;
   schreibe das Kommando ”Daten lesen” ins Kontroll-Register;
   schreibe ”Befehl ausführen” in das Status-Register;
   do
      lies Controller-Status;
   while Status==”noch nicht verfügbar”;

   if Controller-Status==”okay”
    {
      lies das nächste Datenwort aus data-in;
      kopiere das Datenwort in den Speicher;
    }
   else weiter=false;
 }
while weiter;
if Controller-Status==”Fehler” // sonst ”Übertragung beendet”
   Fehlerbehandlung();
```

Die CPU darf in der Wartezeit nichts anderes tun, etwa einen anderen Job oder ein anderes Gerät bedienen. In dieser Zeit könnte sie schließlich ein Eingabedatum verpassen.

2.2.5 Interrupts

Die Idee für Interrupts ist vergleichsweise einfach:

- Die CPU **startet** einen I/O-Vorgang durch Schreiben in die Controller-Register.
- Danach arbeitet der Controller zunächst unabhängig von der CPU (fährt beispielsweise den Schreib-/Lesekopf einer Platte an die erforderliche Stelle, liest einen Block in seinen internen Puffer, etc.).
- Wenn der Controller diese Operation beendet hat, löst er ein spezielles Hardware-Signal („**Interrupt**“) aus – z.B. über einen Anschluss direkt am Prozessor der CPU.
- Dieses Signal unterbricht die CPU bei ihrer normalen Programm-Abarbeitung. Sie reagiert dann mit einer festgelegten Routine auf das Signal – beispielsweise kann sie das gelesene

Datum an die gewünschte Adresse kopieren und den nächsten Lesevorgang anstoßen, oder sie kann den Prozess, der die I/O-Operation ausgelöst hat, „aufwecken“, etc.

- Danach arbeitet die CPU (ähnlich wie nach einem Unterprogrammaufruf) an der Stelle weiter, wo sie unterbrochen wurde. Meist gibt es dafür einen speziellen Maschinenbefehl (bei 80x86-CPUs `IRET` gegenüber `RET` nach normalen Unterprogrammen).

Es gibt i.Allg. mehrere Interrupt-Signale, einen für jedes angeschlossene Interrupt-basierte Gerät. Man könnte nun einen CPU-Anschluss für jedes solche Signal vorsehen. Dadurch würden die Prozessoren aber unnötig groß, und das System wäre nicht besonders flexibel. Neue oder veränderte Geräte würden eine komplizierte Verkabelung erzwingen.

- Daher arbeitet man normalerweise mit einem eigenen Baustein, einem **Interrupt-Controller**. Dieser ist meist „programmierbar“, d.h. per Software für die angeschlossenen Geräte konfigurierbar (**PIC**, programmable interrupt controller). Beispielsweise kann ihm die CPU mitteilen, dass bestimmte Signale *ignoriert* („*maskiert*“) werden sollen, etc.
- Die Geräte-Signale gehen zunächst an diesen Baustein, der daraus eine Nummer erzeugt, die er an die CPU meldet. Bei vielen angeschlossenen Geräten gibt es möglicherweise mehrere Interrupt-Controller.
- Zu jeder Interrupt-Nummer gibt es eine „Interrupt-Service-Routine“, die auf den Interrupt gezielt reagiert. Für die Zuordnung Nummer → Routine gibt es eine Tabelle („**Interrupt-Vektor**“). Ihr Standort ist prozessorabhängig, üblicherweise liegt sie an einer festgelegten kleinen Adresse im Hauptspeicher, oder ein Prozessor-Register enthält ihre Adresse.
- Bei den meisten Systemen sind diese Routinen Teil des Betriebssystems und laufen in einem privilegierten Modus ab. Aus Sicherheitsgründen kann ein Benutzer daher den Interrupt-Vektor normalerweise nicht selbst verändern und eigene Routinen einhängen (außer auf einigen kleineren Systemen).

Typische Interrupts sind folgende:

Tastatur:	Taste gedrückt/losgelassen
Maus:	Maus bewegt, Maustaste gedrückt/losgelassen
Festplatte:	Puffer komplett geschrieben/gelesen
Soundkarte:	Sample komplett abgespielt
Grafikkarte:	vertikale Austastlücke erreicht

Als Beispiel ist rechts die Belegung unter MS-DOS aufgeführt.

Außer der Nummer des Interrupts ist jeweils auch der Adressbereich angegeben, in dem die entsprechenden Controller-Register im I/O-Bereich erscheinen.

Controller/Gerät	I/O-Adressraum	Nummer
Uhr	`0x040-0x043`	8
Tastatur	`0x060-0x063`	9
ser. Schnittst. 2	`0x2f8-0x2ff`	11
ser. Schnittst. 1	`0x3f8-0x3ff`	12
Festplatte	`0x320-0x32f`	13
Floppy	`0x3f0-0x3f7`	14
Drucker	`0x378-0x37f`	15

Unter L*i*nux-x86 erhält man die aktuelle Interrupt-Belegung (und die Anzahl bisher eingegangener Signale!) durch „`cat /proc/interrupts`“, die I/O-Ports mit „`cat /proc/ioports`“ (genaueres zu `/proc` später):

```
 0:    1112215   timer
 1:      67892   keyboard
 4:      23299 + serial
 5:          2   sound blaster
 8:          2 + rtc
 9:        123   NE2000
12:     155305   PS/2 Mouse
13:          1   math error
14:     129923 + ide0
15:          0 + ide1
```

```
0040-005f : timer
0060-006f : keyboard
0070-007f : rtc
01f0-01f7 : ide0
0220-022f : sound blaster
02f8-02ff : serial(auto)
0330-0333 : SB MIDI
03c0-03df : vga+
03f0-03f5 : floppy
03f6-03f6 : ide1
b400-b41f : NE2000
```

- Das Ausführen einer Interrupt-Routine ähnelt einem Unterprogramm-Aufruf: beispielsweise muss die Rücksprung-Adresse gerettet werden (früher oft an einer festen Adresse abhängig von der Interrupt-Nummer, inzwischen eher auf dem System-Stack).

  Zusätzlich muss die Routine selbst alle Prozessor-Register retten, deren Inhalte sie verändert. (Oft sind Interrupt-Routinen ohnehin in Assembler geschrieben – von einer Hochsprache aus muss hier ein Assembler-Vor- und Nachspann eingebunden werden.) Nach der Routine muss der Ausgangszustand wiederhergestellt werden.

- Meistens werden während der Behandlung eines Interrupts weitere Interrupts gesperrt, d.h. im Controller hinausgezögert. Aufeinanderfolgende Interrupts desselben Gerät gehen so allerdings meist verloren. Im Allgemeinen sollte man diese Routinen deshalb möglichst kurz halten.

  Eventuell werden den Interrupts Prioritäten zugeordnet, und Interrupts mit höheren Prioritäten werden während der Behandlung eines Interrupts niedrigerer Priorität zugelassen.

Es gibt weitere Situationen, in denen Interrupts ausgelöst werden:

Hardware-Fehler: Hierzu gehört beispielsweise ein Parity-Fehler beim Zugriff auf den Speicher (fehlerhaftes RAM) und ähnliches.

Bei den Fehler-Interrupts ist der „**Page Fault**“ besonders wichtig, nämlich bei der Verwaltung von virtuellem Speicher. Page Fault bedeutet, dass eine virtuelle Speicheradresse angesprochen wurde, deren physisches Gegenstück gerade nicht im Hauptspeicher, sondern ausgelagert auf Platte liegt (siehe dazu den Abschnitt zu virtuellem Speicher).

Software-Fehler: Hier liegt die Ursache meist in einem Fehler des Programms. Beispielsweise wird durch 0 dividiert, oder es soll ein ungültiger Maschinenbefehl ausgeführt werden.

Software-Interrupts: Es gibt meistens auch Maschinenbefehle, die dazu gedacht sind, einen Interrupt auszulösen, z.B. auf Intel-Rechnern der Befehl `INT` (z.B. `INT 21h`, löst Interrupt Nummer 21h=33 aus).

Interrupt-Befehle werden gern dafür benutzt, eine Schnittstelle zwischen Anwenderprogramm und Betriebssystem zu realisieren.

- Beispielsweise braucht eine Routine, die ein Zeichen auf den Bildschirm ausgibt, dann nicht immer an einer festen Adresse zu liegen. Ihr wird einfach eine Interrupt-Nummer oder eine Kombination aus Interrupt-Nummer und Register-Wert zugewiesen, ihre tatsächliche Adresse wird in den Interrupt-Vektor eingetragen. Alle Anwenderprogramme können sie nun z.B. mit dem `INT`-Befehl aufrufen. Es ist kein Linking erforderlich, und bei einer neuen System-Version gibt es keine Probleme mit veränderten Adressen der Routinen.

- Auf Systemen mit Dual-Mode-Befehlen sind Interrupts meist die einzige Möglichkeit, die CPU in den privilegierten Modus zu schalten. Systemaufrufe, die wie oben realisiert sind, können so automatisch in diesem Modus laufen. Durch das `IRET` wird wieder in den User-Mode geschaltet.

Nicht-Hardware-Interrupts werden oft auch „**Traps**“ genannt. In der 68000er-Familie heißt der entsprechende Maschinenbefehl ebenfalls `TRAP`. Manchmal werden Hard- und Software-Interrupts unterschieden und mit verschiedenen Vektoren behandelt.

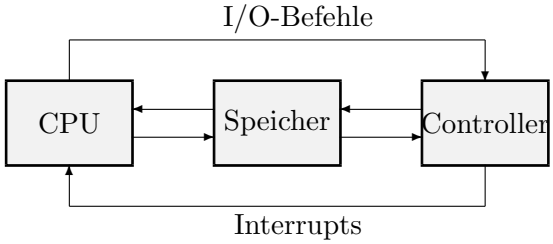
2.2.6 DMA (Direct Memory Access)

Bei langsamen zeichenorientierten Geräten wie Tastatur leidet die Systemleistung unter dem Interrupt-Aufwand nicht merklich.

Blockorientierte Geräte liefern dagegen zusammenhängende Daten, manchmal fast in der Größenordnung der Hauptspeichergeschwindigkeit. Es bietet sich dann natürlich nicht an, bei jedem Byte einen Interrupt auszulösen und durch den damit verbundenen Aufwand (Latenz) das System lahmzulegen.

Üblicherweise landet ja ein Block (beim Lesen) zunächst im Pufferspeicher des Controllers. Statt eines Interrupts pro Byte kann nun ein Interrupt pro Block ausgelöst werden. Danach muss die CPU allerdings (in einer Schleife) die Bytes (oder Worte) einzeln über Controller-Register-Operationen lesen und in den Hauptspeicher schreiben: Bereitschaft abwarten, gewünschte Nummer schreiben, Befehl schreiben, Status schreiben, Status abwarten, Byte lesen, Byte in den Hauptspeicher kopieren. (Analoges gilt für das Füllen des Puffers vor einem Schreibvorgang.)

Statt diese zeitaufwendige Lösung einzusetzen, gestattet man den Controllern den „direkten“ Zugriff auf den Hauptspeicher (über den kleinen Umweg über einen DMA-Controller-Baustein).



Üblicherweise wird dennoch zunächst der Controller-interne Puffer gefüllt und dann per DMA in den Hauptspeicher kopiert. Das liegt daran, dass der Hauptspeicher kurzzeitig nicht zugänglich sein könnte, weil er von einem anderen Controller oder der CPU benutzt wird. Das physische Gerät liefert aber ohne Unterbrechung Daten, von denen keines verlorengehen darf. (Es

gibt Mischlösungen, bei denen normalerweise direkt per DMA geschrieben wird und nur bei Hauptspeicher-Engpässen gepuffert wird.)

Eine interessante Folge des zweistufigen Lesens ist es, dass auf Platten oft keine aufeinanderfolgenden Sektoren gelesen oder geschrieben werden können. Daher erhalten die Blöcke eine andere logische Reihenfolge gegenüber der physischen – es werden Blöcke übersprungen. In der Zeit, in der der Kopf über den übersprungenen Blöcken schwebt, findet z.B. der DMA-Vorgang statt. Die Anzahl übersprungener Blöcke heißt **Interleave-Faktor**.

Bemerkung: Auf Betriebssystem-Seite (und in den I/O-Bibliotheken) wird oft noch zusätzlich im Hauptspeicher gepuffert:

- Der Benutzer fordert die Eingaben byteweise an (`getchar`), das Gerät liefert Blöcke.
- Unterschiedliche Geschwindigkeiten beim Transfer zwischen verschiedenen Geräten können ausgeglichen werden.

Für die koordinierte DMA bei mehreren Geräten wird normalerweise ein weiterer spezialisierter Baustein eingesetzt, ein **DMA-Controller**. Er stellt mehrere „DMA-Kanäle“ zur Verfügung. Einige Geräte sind fest auf einen solchen Kanal eingestellt, andere können per Software oder mit Jumpern konfiguriert werden.

Der typische ISA-DMA-Controller bei Intel-Rechnern stellt acht Kanäle zur Verfügung. Er arbeitet mit 16-Bit-Adressen, sodass DMA nur in die untersten 16 MByte möglich ist. In L*i*nux-x86 erfährt man die Belegung der Kanäle mit „`cat /proc/dma`“, z.B.:

```
1: Sound Blaster8
2: floppy
5: Sound Blaster16
```

2.3 Strukturen von Betriebssystemen

Bei einem Betriebssystem ist es besonders wichtig, dass es vom Beginn seines Entwurfs an vernünftig strukturiert wird. Eine sichere (Weiter-) Entwicklung ist sonst schwer möglich, ohne dass es einmal unter all seinen Bugs begraben wird.

Methoden des *Information Hiding* (Geheimnisprinzips) verbessern eine Software eigentlich *immer*, ob sie nun so konsequent angewandt werden, wie bei objektorientierten Sprachen möglich, oder ob sie zumindest in Form eigenständig übersetzbarer Teilprogramme vorliegen.

Sowohl bei MS-DOS wie auch bei frühen UNIX-Systemen ist eine Struktur nur sehr schwer zu erkennen. Es gibt viele Betriebssysteme, die zwar gut gegliedert sind, aber wenig Verbreitung gefunden haben (OS/2).

Zum groben Aufbau der bekannten Systeme kommen wir weiter unten. Wir betrachten zunächst vier mehr oder weniger sinnvolle Möglichkeiten, Betriebssysteme zu gliedern.

2.3.1 Monolithische Systeme

Eine gewisse Gliederung hat ein Betriebssystem natürlich *immer*, nämlich eine Aufteilung in Prozeduren mit definierten Aufruf-Schnittstellen. In einem sonst nicht weiter strukturierten System kann theoretisch jede solche Routine jede beliebige andere aufrufen.

Betriebssysteme dieser Art heißen **monolithisch**. Es entsteht durch Übersetzen ggf. mehrerer Quelltexte und Zusammenlinken ein einziger großer Programmblock.

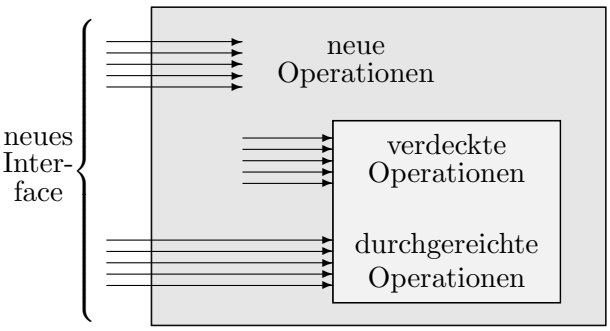
2.3.2 Geschichtete Systeme

Eine gute Möglichkeit, ein komplexes System (wie ein Betriebssystem) zu untergliedern, ist es, es in Schichten (Layers) aufzuteilen, die streng aufeinander aufbauen. Jede Schicht definiert sich durch die Operationen (Schnittstellen), die sie nach außen zur Verfügung stellt.

Grafisch kommt man der Sache ein wenig näher, wenn man die „unteren“ Schichten zu „inneren“ Schichten macht, d.h. abstraktere Schichten um konkretere *herumlegt*. Die innere Schicht wird dadurch nach außen abgekapselt.

Es brauchen nicht *alle* Zugriffe auf die innere Schicht von außen verboten zu werden. Sie werden zu Operationen, die die äußere Schicht an die innere Schicht weiterleitet.

Es kommen neue, abstraktere Operationen dazu. Es ist nun gerade die Aufgabe der äußeren Schicht, sie auf Operationen der inneren zurückzuführen.



Natürlich kann eine neue Schicht um *mehrere* alte Schichten herumgelegt werden.

Es bestehen Ähnlichkeiten zu den Prinzipien von *Layering* und *Vererbung* bei objektorientierten Programmiersprachen, es liegt aber keine genaue Analogie vor. Die Abkapselung alter Schnittstellen entspricht dem Layering – ein Objekt wird explizit als Teilobjekt in ein anderes eingebaut. Das unveränderte Durchreichen alter Operationen und das Hinzufügen neuer entspricht dagegen genau der Vererbung.

MULTICS war ein sehr streng geschichtetes System. Innere Schichten waren nach außen völlig abgekapselt, d.h. das „Durchreichen“ von Operationen musste explizit geschehen. Für jede durchzureichende Operation gab es eine „Dummy-Operation“ in der äußeren Schicht, die einfach die entsprechende innere Operation aufrief. Das war nur auf der damals verwendeten speziellen Hardware möglich. Es konnte hardwaremäßig gekennzeichnet werden, von welchen anderen Prozeduren aus eine Prozedur aufrufbar war.

Ein vom Konzept her geschichtetes System kann durchaus ein monolithisches Programm als Erzeugnis haben. Die Schichtung besteht dann auf Ebene der Programmiersprache, in der es geschrieben ist, hinterher nicht mehr physisch.

2.3.3 Virtuelle Maschinen

Jede der Schichten kann als eine eigene virtuelle Maschine aufgefasst werden. Die unterste Ebene ist die Hardware selbst – die virtuelle Maschine entspricht der physischen. Die Operationen fallen mit den Maschinenbefehlen und dem Schreiben in physische Register zusammen.

Auf der nächsten Ebene werden zusätzliche Operationen (z.B. in Form von Systemaufrufen) hinzugefügt. Es werden alte Operationen abgedeckt – z.B. das Schreiben in I/O-Register. Irgendwann werden auch einige Maschinenbefehle unterbunden (wenn möglich, bei Dual-Mode-Prozessoren).

Typischerweise ergibt sich etwas der folgenden Art: was um den physischen Speicher herumgelegt wird, ist die Schicht des virtuellen Speichers. Um die I/O-Controller wird die Schicht der Geräte-Treiber gelegt. Schließlich liegt irgendwann um die CPU die Prozess-Schicht, sodass jeder Prozess eine eigene CPU zu haben scheint.

Beispiel: Bei einer erfolgreichen Erweiterung des IBM-Systems OS/360, die später VM/370 genannt wurde, wurde dieses Prinzip so weit getrieben, dass jedem Benutzer eine virtuelle Maschine zur Verfügung stand, die exakt der tatsächlichen Hardware entsprach. Die IBM 360 wurde also in sovielen Kopien simuliert wie nötig.

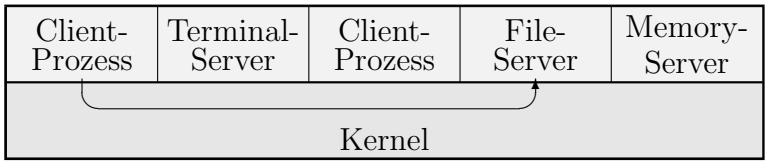
Auf jeder dieser Kopien konnte sogar ein anderes Betriebssystem laufen, beispielsweise auf einer das normale OS/360, auf einer anderen ein Timesharing-System, etc. Während der Entwicklung ließ man auf einer virtuellen Maschine wiederum ein VM/370 laufen – die jeweils aktuelle Testversion.

Ein einfacher Systemaufruf auf einer OS/360-Kopie löst also ggf. eine Serie von I/O-Operationen aus, die aber von VM/370 abgefangen werden und (mit den anderen Kopien multiplext) an die echte Hardware gelangen!

2.3.4 Client-Server-Modell

Modernere Betriebssysteme sind nicht streng gelayert, versuchen aber, den „monolithischen“ Teil so weit wie möglich zu reduzieren. Er überlebt in Form des Kernels. Abstraktere Teile werden in Prozesse ausgelagert, die in einer Schicht oberhalb des Kernels leben.

Ein Prozess, der eine Betriebssystem-Funktion benötigt (client process), ruft dazu eine Kernel-Operation auf. Die eigentliche Ausführung ist aber in einem anderen Prozess implementiert (server process), der durch den Kernel benachrichtigt oder erst erzeugt wird. Server-Prozesse sind im Allgemeinen Teil des Betriebssystems, Clients können auch Anwenderprogramme sein.



Der Kernel übernimmt also im Wesentlichen nur die Kommunikation zwischen Clients und Servern. Er läuft im privilegierten Modus der Maschine, alle darüberliegenden Prozesse nur im normalen Benutzer-Modus. Das hat den Vorteil, dass ein fehlerhafter Server-Prozess zwar abstürzen, aber nicht das ganze System mitreißen kann.

Einige kritische Prozesse laufen meistens doch im privilegierten Modus, z.B. Treiber, die direkt auf I/O-Register von Controllern zugreifen müssen. Alternativ kann man Low-Level-Aufrufe im Kernel realisieren, die auf Veranlassung der User-Modus-Prozesse die eigentlichen Register bedienen. Es liegt dann in der Verantwortung der Prozesse, korrekte Aufrufe zu produzieren, die die simple Kernel-Routine ungeprüft übernimmt.

Eine sehr attraktive Seite des Client-Server-Modells ist, dass es sich relativ einfach auf Systeme vernetzter Rechner übertragen lässt, bei denen Clients und Server nicht notwendigerweise auf demselben Rechner laufen. Die Kernels der beiden Rechner müssen dann miteinander kommunizieren.

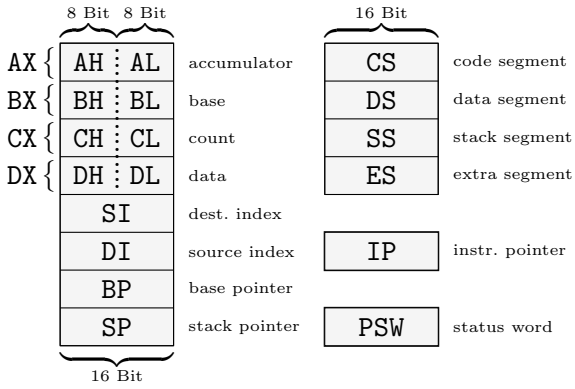
2.4 Prozessorarchitektur bei Intel-CPUs

Da wir uns neben UNIX auch mit MS-DOS und Windows befassen wollen, müssen wir uns zum besseren Verständnis kurz mit den Betriebsmodi der Intel-Prozessoren vertraut machen.

Real Mode:

Das ist der Modus, in dem sich jeder 80x86-Prozessor beim Start oder nach einem Reset befindet, und der einzige Modus der Prozessoren 8088 und 8086. „Real“ ist hier als Gegensatz zu „virtuell“ zu lesen.

Die CPU ist ein reiner 16-Bit-Prozessor, insbesondere werden die Register als 8- oder 16-Bit-Register angesprochen. Im Bild ist der Registersatz dargestellt.



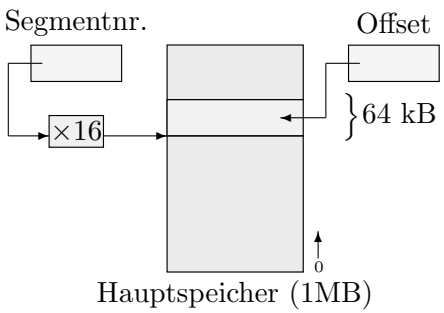
Der Befehlssatz ist nicht „orthogonal“, d.h. ein Befehl ist auf spezielle Register festgelegt. Allgemeinere Operationen und Arithmetik finden in `AX` bis `DX` statt, `DI` und `SI` sind Ziel und Quelle bei indizierten (Tabellen-) Operationen, etc.

- Der Prozessor hat 20 Adressleitungen und kann daher 2^{20} Byte = 1 MB Speicher adressieren (von denen unter DOS 640 KB frei benutzbar sind).

  Eine Einmaligkeit der frühen Intel-Prozessoren ist es, dass man nicht direkt über eine absolute Adresse auf den Hauptspeicher zugreifen kann – man hat keinen „linearen Adressraum“.

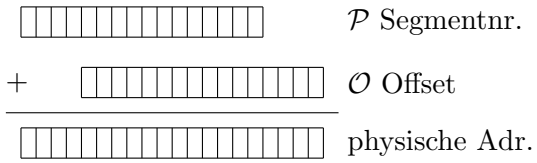
  Der Speicher wird 16-bitweise adressiert. Um eine komplette Hauptspeicheradresse darstellen zu können, benötigt man bei 1 MB aber 20 Bit. Der Operand eines Maschinenbefehls „lade Register `AX` mit dem Wert an Adresse $\mathcal{A}$“ hätte der Operand $\mathcal{A}$ also 32 Bit breit gemacht werden müssen – speicherverschwendend und langsam.

Daher hat man den Speicher in „Segmente“ der festen Größe 64 KB aufgeteilt, die an durch 16 teilbaren Adressen beginnen.



Der Maschinenbefehl oben gibt den Abstand $\mathcal{O}$ („Offset“) vom Segmentbeginn an. Dafür reicht genau ein 16-Bit-Operand. Die Nummer $\mathcal{P}$ des Segments („Paragraph“) stammt aus einem speziellen Prozessor-Register.

Beim Zugriff führt der Prozessor dann die Adressrechnung $\mathcal{A} = 16 \cdot \mathcal{P} + \mathcal{O}$ durch. Eine Adresse kann so natürlich auf unterschiedliche Weise zusammengesetzt werden: $25 = 0 \cdot 16 + 25 = 1 \cdot 16 + 9$, etc.



- Beachte, dass man mit der Segmentnummer `0xffff` Adressen von 16·`0xffff`... 16·`0xffff` +`0xffff` = `0xffff0`...`0x10ffef` erzeugen kann. Ab dem 80286 erhält man so auch unter DOS Zugriff auf die untersten 65520 Bytes *oberhalb* des ersten Megabytes (die sogenannte High Memory Area HMA).

- Der 8086 hat mehrere Segmentregister für unterschiedliche Zwecke. Maschinenbefehle werden aus dem *Code-Segment* gelesen, dem Segment, das durch das Register `CS` festgelegt ist (dafür braucht auch der Programmzähler `IP` nur 16 Bit breit zu sein). Daten stammen aus dem *Datensegment* (Register `DS`). Der Laufzeitstack (Rücksprungadressen etc.) liegt im *Stacksegment* (Register `SS`). Ein weiteres Register `ES` (*Extra-Segment*) ist für freiere Verwendung gedacht, z.B. zum Kopieren aus einem Segment in ein anderes.

- Absolute Adressen werden immer in Paaren von Segment und Offset abgespeichert. Beispielsweise besteht jeder Eintrag im Interrupt-Vektor aus Werten für `IP` (niedrigere Adresse) und `CS` (höhere Adresse), die zusammen die Adresse der Service-Routine bilden.

- Theoretisch hat jedes Programm nun die Möglichkeit, auf das gesamte Megabyte Speicher zuzugreifen. Dazu müsste aber beispielsweise vor jedem Zugriff das Segmentregister `DS` passend neu gesetzt werden, was Programme extrem verlangsamen würde. Normale Programme legen daher zu Beginn `DS` fest, verändern es nicht mehr – und können daher nur auf genau **64 kB** Daten zugreifen.

  Compiler stellen meist verschiedene „Speichermodelle“ als Aufruf-Optionen zur Verfügung. Die kleinen Modelle arbeiten dabei mit einem konstanten Daten- bzw. Codesegment, das auf 64 KB beschränkt ist. Die großen Modelle arbeiten mit mehreren Segmenten, müssen aber bei jedem Zugriff das entsprechende Segmentregister laden und sind daher langsamer.

  Jedes Datensegment ist immer noch auf 64 KB beschränkt, sodass man keine übergroßen Arrays anlegen kann – außer im sehr langsamen Modell Huge.

In `C` gibt es dadurch verschiedene Arten von Pointern (innerhalb eines Segments oder segmentübergreifend) und verschiedene Arten von Funktionen (innerhalb eines Codesegments oder segmentübergreifend aufrufbar).

Modell	Codeseg.	Datenseg.
tiny	1=DS	1=CS
small	1	1
medium	n	1
compact	1	n
large	n	n
huge	n	n^+

Außerdem muss der Compilerhersteller für jedes Modell eine eigene Version der Bibliotheken(!) zur Verfügung stellen, die die Segmente im gleichen Stil wie das Anwenderprogramm behandelt.

Virtueller 8086-Modus:

Dieser Modus wurde mit dem 80386 eingeführt und bietet Hardware-Voraussetzungen für das Anlegen mehrerer virtueller Maschinen (VMs) im System. Alle verhalten sich jeweils wie ein herkömmlicher 8086-Rechner, sind voneinander aber völlig getrennt.

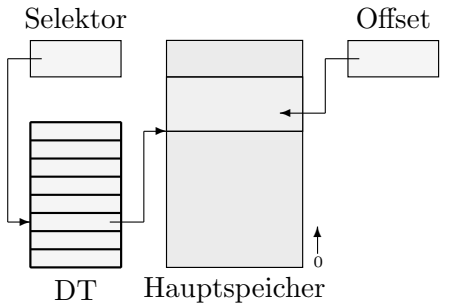
Ein Programm adressiert 1 MB Speicher wie vom Real Mode gewohnt. Die Adressierung wird aber über Seitentabellen auf physischen Speicher an einer anderen Stelle umgeleitet (Page Translation). An Hauptspeicher braucht man also immer mindestens 1 MB für jede gewünschte virtuelle Maschine, auch wenn dieser teilweise unbelegt bleibt.

DOS ist für den Real Mode entworfen worden und kann daher innerhalb einer virtuellen Maschine unverändert ablaufen. Diesen Mechanismus macht sich Windows zunutze, wenn es (mehrere) DOS-Programme ablaufen lässt.

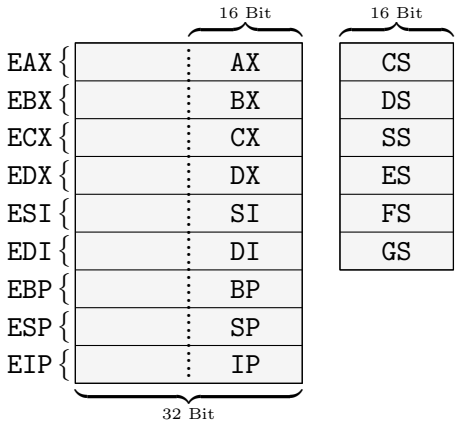
Mit Paging-Mechanismen werden wir uns genauer in einem späteren Kapitel beschäftigen.

Protected Mode:

Es kann Hauptspeicher bis 2^{32} Byte = 4 GByte angesprochen werden (beim 80286 nur 2^{24} Byte = 16 MB). Ab dem 80386 entfällt auch die 64 kB-Segmentgrenze. Der Speicher kann über Deskriptor-Tabellen aufgeteilt werden in ggf. zugangsbeschränkte Bereiche (daher „protected“), eine notwendige Voraussetzung für Multitasking. Es gibt privilegierte Befehle.



Ab dem 80386 sind die Prozessoren echte 32-Bit-CPUs, insbesondere sind die Register 32 Bit breit. Der Befehlssatz ist wesentlich „orthogonaler“, fast alle Register können als Allzweck-Register verwendet werden.



Es gibt zwei zusätzliche Allzweck-Segmentregister `FS` und `GS`. Außerdem werden die Segmentregister in diesem Modus völlig anders verwendet, nämlich als Selektoren, also Zeiger in die Speicher-Deskriptortabellen. DOS ist allein aus diesem Grund in diesem Modus absolut nicht lauffähig.

Bemerkung: Architekturen anderer Hersteller waren bereits in ihrer 16-Bit-Zeit zukunftsorientiert angelegt. Die Prozessoren der 68000er-Familie von Motorola beispielsweise arbeiteten von vornherein intern mit 32-Bit-Adressregistern (und ohne Segmentierung), auch als die Chips eigentlich nur 24 Adressleitungen hatten. Sie hatten schon immer 32-Bit-Befehle. In den 16-Bit-CPUs der ersten Zeit wurde beispielsweise eine 32-Bit-Addition im Chip in zwei 16-Bit-Additionen zerlegt. In den 32-Bit-Nachfolgern war die Addition dagegen mit 32 Bit fest verdrahtet. Dadurch waren *alle* Programme von vornherein 32-Bit-Programme, und es gab nie solche Kompatibilitätsprobleme wie bei Intel.

2.5 MS-DOS

MS-DOS entstand unter sehr schweren Hardware-Beschränkungen. Es musste die wichtigste Funktionalität für ein Ein-Benutzer-System abdecken, aber dabei mit möglichst minimalem Speicheraufwand für System und Daten auskommen.

Die 8088/8086-Prozessoren, für die es zuerst entwickelt wurde, boten u.a. folgende Schwierigkeiten:

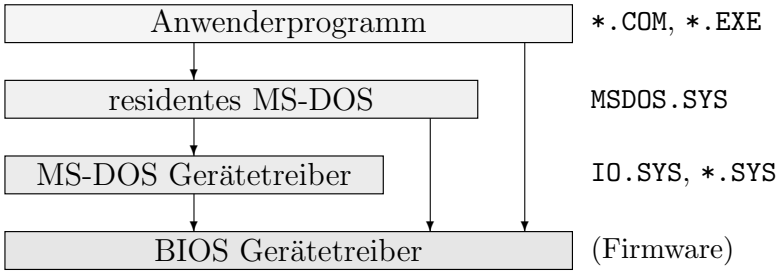
- Extremer Speichermangel. Die Prozessoren hatten überhaupt nur einen *möglichen* Adressraum von 1 MB. Speicher war damals aber ohnehin so kostbar, dass die meisten Systeme mit 128 KB RAM ausgestattet waren. DOS musste mit Speicher stark haushalten (für seinen eigenen Code und für Systemstrukturen).
- Keine Möglichkeit zur Speicherabschottung. Hierdurch wurde ein vernünftiges Multiprogramming unmöglich gemacht.
- Keine privilegierten Befehle. Jedes Programm hat die Möglichkeit, alle verfügbaren Befehle, auch „gefährliche“, zu verwenden. Zusammen mit Punkt 2 kann jedes Programm „Amok laufen“ und das System jederzeit zum Absturz bringen, Daten vernichten, etc.

Die erste Version von MS-DOS wurde zusammengehackt und ist praktisch nicht strukturiert. Es gibt wilde Sprünge zwischen allen möglichen Teilen des System-Codes. Durch die Änderung von Hardware und die starken Speicher-Einschränkungen haben sich danach aber einige Strukturen entwickelt, die im Folgenden kurz beschrieben werden.

Die grundlegenden Hardware-Zugriffe (auf I/O-Register-Ebene) erfolgen nicht wirklich im Betriebssystem, sondern im **BIOS** (basic input/output system). Es ist absolut abhängig von der exakten Hardware und kann und in Form eines ROMs („Firmware“) fest zum jeweiligen Gerät gezählt werden.

Der zentrale Teil des Betriebssystems, der unter anderem die Aufruf-Schnittstellen für die Anwenderprogramme enthält, wird beim Booten resident in den Speicher geladen.

Geräte-Treiber (auf einer von der Hardware entfernteren Stufe als das BIOS) werden nur bei Bedarf, d.h. durch Angabe in einer Konfigurations-Datei, geladen.



Im Bild erkennt man zwar eine Schichtung wie beim oben beschriebenen Layering, sie wird aber stark durchbrochen. Der residente Teil greift nicht nur über die Treiber-Schnittstellen, sondern gern auch direkt auf die BIOS-Services zu – und sogar *Anwenderprogrammen* ist das möglich. Fehlerhafte Zugriffe können aber nicht nur zu *Abstürzen*, sondern sogar zu *Hardware-Schäden* führen.

Die `.SYS`-Dateien sind übrigens vom Format her normale `.COM`-Dateien, die durch die Endung lediglich als innerer Bestandteil des Systems gekennzeichnet werden.

- Das BIOS wird direkt nach dem Booten als erstes aktiv. Es kann Initialisierungen von Geräten und Speicher vornehmen. Letztlich sucht es auf Disketten oder Platten nach einem Medium, auf dem der erste Block als „Bootblock“ (MBR, Master Boot Record) markiert ist (als letztes Wort muss hier 0xaa55 stehen). Der Block wird geladen und als Code ausgeführt. Das kurze Programmstück ist hauptsächlich dafür verantwortlich, die weiteren Teile (vom entdeckten Boot-Medium) nachzuladen.

- Das **ROM BIOS** ist nicht zu verwechseln mit den DOS-Gerätetreibern, die zusammen auch manchmal als DOS-BIOS bezeichnet werden. Es lag zuerst in der Datei `IBMBIO.COM`, mittlerweile in `IO.SYS`. Hieraus werden bei Bedarf Routinen für die logischen Geräte `CON`, `PRN`, etc. nachgeladen.

- Der residente Teil liegt in der (versteckten) Datei `MSDOS.SYS` (früher `IBMDOS.COM`). Interrupt-Routinen etc. *müssen* natürlich *permanent* im Speicher liegen. Dieser Teil ist aber möglichst klein gehalten.

- Nachdem die bisher angegebenen Bestandteile geladen, geschickt im Speicher untergebracht und initialisiert worden sind, wird die Datei `CONFIG.SYS` (die einzige Datei mit dieser Endung, die keinen Binärcode, sondern ASCII-Text enthält!) nach `DEVICE`-Angaben durchsucht, und die entsprechenden Gerätetreiber (in Form von echten `.SYS`-Dateien) werden nachgeladen (z.B. `ANSI.SYS`, `RAMDRIVE.SYS`).

- Das Programm, das Benutzer-Befehle annimmt, interpretiert und Programme startet, heißt Kommando-Prozessor. Er befindet sich in der Datei `COMMAND.COM` und ist nicht resident. Wenn ein (nicht intern implementierter) Befehl ausgeführt wird, wird der Prozessor überschrieben, um Speicher zu sparen, und muss nach Beendigung des Programms erneut aus der Datei geladen werden. Dazu muss ein sehr kleiner Teil natürlich doch resident werden. Er hat nur die Aufgabe, festzustellen, ob der Hauptteil überschrieben wurde, und ihn ggf. neu zu laden.

- Benutzer-Befehle sind entweder in `COMMAND.COM` eingebaut (`DIR`, `COPY`), oder sie sind (z.B. aus Speichergründen) als normale Programme in Dateiform ausgelagert (`FORMAT`, `DISKCOPY`). Letztere gehören oben im Bild also eigentlich in den Kasten „Anwenderprogramme“.

- Anwenderprogramme liegen im Format `.COM` oder `.EXE` vor.

  Die `.COM`-Dateien sind eine Erblast von CP/M. Sie stellen direkte Speicherabbilder dar. Entweder wird dabei mit relativen Adressen (im Code) gearbeitet, oder die Programme können immer nur an eine vorgegebene Adresse im Speicher geladen werden. Außerdem hat dieses Format von CP/M die Maximalgröße von 64 KByte geerbt.

  In MS-DOS 1.1 wurde das neue Format `.EXE` eingeführt, das die Grenze von 64 KByte überschritt. Da man so schwer mit nur relativen Adressbezügen auskommt, wird zusätzlich zum Code eine Relokationstabelle abgespeichert, in der angegeben ist, an welchen Stellen Adressen beim Laden an eine beliebige Speicheradresse geändert werden müssen.

  Das `.EXE`-Format ist z.B. auch von Vorteil für mehrere Programme, die nun gleichzeitig im Speicher liegen können (Treiber, residente Programme), allerdings in MS-DOS nicht gleichzeitig lauffähig sind.

Die tatsächlichen Systemaufrufe erfolgen in MS-DOS mit Interrupts, wie oben schon angedeutet. Es gibt 256 mögliche Interrupts bei den Intel-CPUs. Die niedrigen Nummern sind für Hardware-Interrupts und echte Fehlersituationen reserviert.

Einige Interrupts dienen zum Aufruf von Routinen aus dem BIOS (0x10 bis 0x1A, 0x40), einige zur Kommunikation mit DOS (0x20-0x3f). Viele sind leider für inzwischen nicht mehr unterstützte Hardware, für den BASIC-Interpreter oder undokumentierte Funktionen reserviert.

Es gibt natürlich mehr Systemroutinen als Interrupts. Zusammengehörige Routinen sind zusammengefasst und auf dieselbe Interrupt-Nummer gelegt. Sie werden dann durch den Wert eines Prozessor-Registers voneinander unterschieden. Die Interrupt-Service-Routine ist dann nur ein „Dispatcher“, der abhängig von der übergebenen Nummer über eine Tabelle die passende tatsächliche Routine aufruft. Eine Auswahl:

BIOS		DOS	
0x10	Video-Routinen	0x21	DOS-Funktion aufrufen
0x11	Konfiguration abfragen	0x20	Programm beenden
0x12	RAM-Größe abfragen	0x27	Programm beenden, resident lassen
0x13	Floppy/Harddisk-Routinen	0x25	Floppy/Harddisk lesen
0x14	Bedienung der seriellen Schnittstelle	0x26	Floppy/Harddisk schreiben
0x15	Kassetten-Recorder-Routinen	0x67	Expanded-Memory-Routinen
0x16	Tastatur abfragen		
0x17	Drucker-Routinen		
0x19	Warmstart (CTRL–ALT–DEL)		
0x1a	Zeit-/Datumsfunktionen		
0x40	Floppy-Routinen		

Am stärksten mit Routinen belegt ist wahrscheinlich der Interrupt 0x21. Die Nummer der gewünschten Funktion wird im Register `AH` angegeben. Typische Routinen sind folgende:

0x00	Programm beenden	0x2b	Zeit setzen
0x01	Taste lesen mit Echo	0x30	DOS-Versionsnummer abfragen
0x02	Zeichen ausgeben (Standardgerät)	0x38	Land setzen
0x05	Zeichen an Drucker schicken	0x39	Verzeichnis anlegen
0x07	Taste lesen ohne Echo	0x3d	Datei öffnen (neue Version)
0x0e	aktuelles Laufwerk wählen	0x3e	Datei schließen (neue Version)
0x0f	Datei öffnen (alte Version)	0x3f	aus Datei lesen
0x10	Datei schließen (alte Version)	0x40	in Datei schreiben
0x2a	Datum abfragen	0x48	Speicherblock allozieren
0x2b	Datum setzen	0x49	Speicherblock freigeben
0x2a	Zeit abfragen	0x4b	anderes Programm laden/ausführen

Beispiel: Als Beispiel schauen wir uns an, wie die **Tastatur-Abfrage** funktioniert, und wie DOS und BIOS dort zusammenarbeiten.

- Wenn der Benutzer eine Taste drückt, löst die Hardware in der Tastatur (evtl. ein eigener Mikroprozessor) einen Hardware-Interrupt 1 des Systems auf (Leitungen des Interrupt-Controllers werden angesprochen). Der Controller löst daraufhin den für Tastatur-Ereignisse vorgesehenen Interrupt 9 der CPU aus (siehe die Tabelle auf Seite 29). Mittlerweile legt die Tastatur den Code der Taste („Scancode“) im I/O-Speicher an (an „Port“ 0x60).

- Für die Beantwortung des Hardware-Interrupts ist das BIOS zuständig. Im Interrupt-Vektor ist also an der Stelle 9 ein Verweis auf die passende Adresse im BIOS-ROM eingetragen.

  Die BIOS-Routine holt den Scancode von Port 0x60 ab (mit dem Maschinenbefehl `IN`) und wandelt sie in den passenden ASCII-Code um. Dazu muss sie berücksichtigen, welche Modifikationstasten (SHIFT, ALT, etc.) noch gedrückt sind. Entsprechende Informationen muss sie also intern immer mitspeichern. Den berechneten ASCII-Code legt sie zunächst in einem kleinen Pufferspeicher ab, dem Tastaturpuffer, der typischerweise 10 Zeichen fasst.

- Anwenderprogramme fragen Tastatureingaben meist über DOS-Routinen ab, hauptsächlich über den Software-Interrupt 0x21. Funktionen aus Hochsprachen-Bibliotheken wie `getchar` in `C` basieren intern auf solchen Interrupt-Aufrufen.

  Hinter dem Interrupt 0x21 verbergen sich sehr viele einzelne Routinen. Die Tastatur-Abfrage mit Echo (das Zeichen erscheint auch auf dem Bildschirm) hat die Nummer 1, die dem Dispatcher im Register `AH` übergeben wird. Die Routine wird letztlich den Code der gedrückten Taste im Register `AL` zurückliefern.

  Folgendes Assembler-Stückchen liest solange Zeichen von der Tastatur, bis ein Sternchen eingegeben wird:

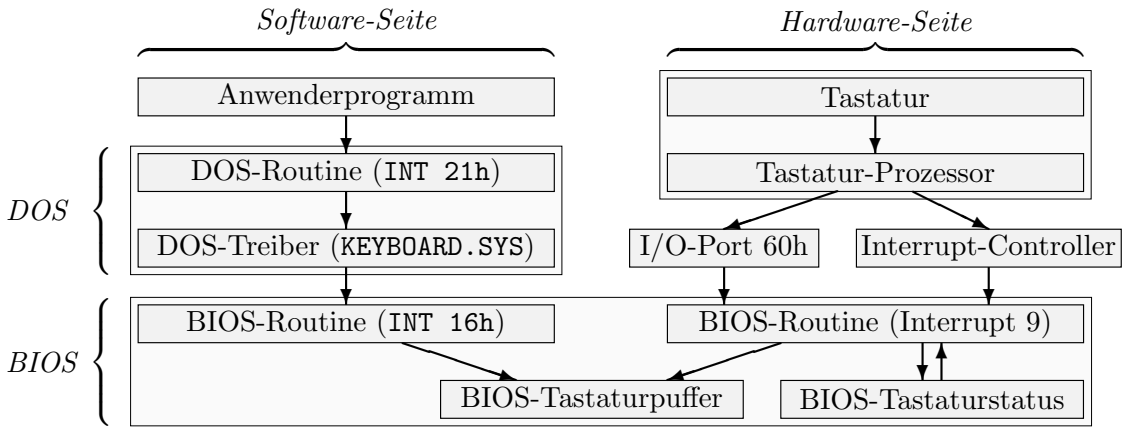
```
loop:   mov ah,1        ; Funktionsnummer 1 für "Tastencode lesen"
        int 21h         ; DOS-Aufruf
        cmp al,'*'      ; gelesene Taste = Sternchen (ASCII-Code 42)?
        jne loop        ; nein, nächster Schleifendurchgang
```

- Die aufgerufene Routine bedient sich nun des DOS-Gerätetreibers für die Tastatur (z.B. `KEYBOARD.SYS` oder `ANSI.SYS`). Der Treiber kann nun noch diverse Code-Umsetzungen vornehmen (z.B. deutsche Tastatureinstellung über eine Code-Tabelle, Funktionstastenbelegung).

- Der Treiber muss auf den Tastaturpuffer zugreifen. Da der aber Teil des BIOS ist, geschieht das über eine BIOS-Routine, nämlich über den Software-Interrupt 0x16 (siehe Tabelle oben). Das einfache Abfragen hat die Unter-Nummer 0 – mit einer anderen Nummer fragt man z.B. den Status der Tastatur (Modifikationstasten) ab.

- Die DOS-Routine 0x21–1 von oben erhält also vom Treiber den ASCII-Code des nächsten noch nicht verwendeten Tastendrucks. Sie ist nun noch für das Echo zuständig, wofür sie Video-Routinen aus dem BIOS benötigt (Software-Interrupt 0x10).

  Schließlich muss sie dem Anwenderprogramm, das sie aufgerufen hat, den gelesenen Code zurückliefern. Das geschieht im Register `AL`.



Beispiel: Die meisten Compiler für Hochsprachen bieten die Möglichkeit an, Interrupt-Routinen (DOS oder BIOS) auch direkt von einem Hochsprachen-Programm aus aufzurufen.

Man kann dort natürlich nicht direkt in die Prozessor-Register schreiben. Stattdessen bedient man sich einer Struktur, bei Borland-`C` zum Beispiel **`union`** `REGS`, definiert in `dos.h`. Sie enthält Felder für alle wichtigen CPU-Register, 8-bit- und 16-bitweise ansprechbar.

Der eigentliche Aufruf erfolgt mit einer `C`-Bibliotheks-Routine, z.B. `intr` für allgemeine und `intdos` für den häufigen DOS-Interrupt 0x21.

`intdos` erhält als Parameter zwei Pointer auf eine `REGS`-Struktur – eine, aus dem vor dem `INT`-Befehl die Werte gelesen und in die CPU-Register übertragen werden, und eine, in die die Register-Werte nach dem Befehl eingetragen werden.

Unsere Tastatur-Abfrage von oben sieht in `C` dann wie folgt aus:

```
union REGS regs;
do
 {
   regs.h.ah = 1;
   intdos(&regs,&regs);
```

```
 }
while ( regs.h.al != '*' );
```

Im Lauf der Zeit hatte DOS mit immer mehr Schwierigkeiten zu kämpfen. Es war für den Real Mode entworfen und konnte die 32-Bit-Architektur späterer Prozessoren nicht ausnutzen. Es war auf den Zugriff auf das unterste Megabyte des Hauptspeichers eingeschränkt.

Außerdem war man beim Speicher-Layout etwas großzügig vorgegangen. Man hatte feste Adressen für das BIOS, den Video-Speicher und ROM-Erweiterungen vorgesehen. Unterhalb dieses reservierten Bereichs blieben nur ca. 640 KB für Anwenderprogramme übrig.

Adressen	Belegung
0xf0000-0xfffff	BIOS-ROM
0xd0000-0xeffff	ROM-Erweiterung
0xc0000-0xdffff	externes BIOS
0xa0000-0xbffff	Video-RAM
0x01000-0x9ffff	DOS/Anwenderprogramm
0x00000-0x0ffff	Systemverwaltung

Oben rechts ist die *ursprüngliche* Verteilung angegeben. Sie hätte mit neuen DOS-Versionen natürlich leicht geändert werden können. Leider verließen sich nun aber schon die meisten Programme auf diese festen Adressen und wären unter einem neuen Layout nicht mehr lauffähig gewesen.

Außerdem war DOS für einen Ein-Prozess-Betrieb entworfen worden – alle Systemstrukturen und Aufruf-Konventionen sind dementsprechend ausgelegt. Der Code von DOS ist nicht reentrant, d.h. er kann nicht von mehreren Prozessen gleichzeitig ausgeführt werden. Wenn Multitasking-Fähigkeiten hätten eingebaut werden sollen, wäre das Ergebnis nicht mehr kompatibel mit Programmen für Vorgänger-Versionen gewesen.

Microsoft Windows hat sich im Laufe der Zeit zu einem System mit Multitasking entwickelt, das dennoch in der Lage ist, auch älteste DOS-orientierte Programme (meistens) fehlerfrei auszuführen. Es soll im nächsten Abschnitt besprochen werden.

2.6 Windows

Das Windows der Versionen bis 3.11 ist keine Betriebssystem, sondern ein Aufsatz auf MS-DOS. Es stellt eine grafische Bedieneroberfläche zur Verfügung, was aber nicht als Teil eines Betriebssystems angesehen werden kann und von uns hier nicht betrachtet wird.

Durch die DOS-Engpässe erhielt es aber mehr und mehr Betriebssystem-Elemente. Beispielsweise erlaubt es das Halten mehrerer DOS- bzw. spezieller Windows-Programme im Speicher, zwischen denen umgeschaltet werden kann.

Zu seinem Start wird von DOS aus das Kommando `WIN` abgesetzt (aus `WIN.COM`), das andere Teile (s.u.) lädt und initialisiert.

Der Hauptbeitrag, den MS-DOS danach leistet, ist allerdings nur das Dateisystem. Da es für Prozessoren entworfen wurde, die maximal 1MB Hauptspeicher ansprechen konnten, ist es für die Speicherverwaltung nicht geeignet, die deshalb von Windows übernommen wird. I/O-Geräte werden teils über DOS, teils über Windows-eigene Treiber, teils über das BIOS angesprochen.

Windows NT und Windows 95 sind dagegen (unter anderem) „echte“ Betriebssysteme.

2.6.1 Windows und die 80x86-Modi

Wir schauen uns zunächst die Eignung der Prozessor-Modi für wichtige Betriebssystem-Konzepte an:

Real Mode: Dieser Modus ist nicht für Multitasking-Systeme geeignet, da jedes Programm alle Befehle ausführen und auf den gesamten Speicher frei zugreifen kann. Außerdem ist hardwaremäßig einfach nicht mehr als 1 MB Speicher ansprechbar, in den nun DOS, Windows und wenn möglich schließlich noch Anwenderprogramme passen müssen.

Die ersten Windows-Versionen hatten nur diesen Modus zur Verfügung. Bis inklusive Version 3.0 wird er auch noch unterstützt, macht auf neueren Prozessoren aber nur für veraltete Anwendungen Sinn. Windows belegt dann selbst 384 KB der belegbaren 640 kB, lässt also wenig Raum für Anwendungen und verkümmert zu einem umfangreichen Dateimanager.

Virtueller 8086-Modus: Dieser Modus ist wegen der Unterstützung virtueller 8086-Maschinen gut für das Nebeneinander mehrerer DOS-Programme geeignet. Unter Windows können mehrere DOS-Sitzungen unabhängig voneinander laufen (genug Speicher vorausgesetzt). Die DOS-Umgebung vor dem Windows-Start wird komplett geerbt, inklusive Interrupt-Tabelle, speicherresidenten Programmen, etc. Die DOS-Programme multitasken (in 16-Bit-Windows) untereinander nicht. Man kann zwischen echten Windows-Programmen und den DOS-Anwendungen per Maus oder Tastatur umschalten.

Protected Mode: Auf Prozessoren ab 80386 mit mindestens 2 MB Speicher läuft Windows ab 3.0 automatisch im erweiterten Modus (Enhanced Mode), der als Prozessor-Modus den Protected Mode benutzt.

16-Bit-Versionen von Windows nutzen die Prozessorfähigkeiten aber fast nur zur Verwirklichung von virtuellem Speicher und zur Erweiterung des Adressraums von 1 MB auf 16 MB bzw. 4 GB. Für DOS-Anwendungen wird weiterhin der virtuelle 8086-Modus verwendet. Die Windows-Schnittstellen bleiben 16-Bit-Schnittstellen. Wirkliches Multitasking wäre hardwaremäßig möglich, wird aber von Windows nicht unterstützt. Der Betriebssystemkern ist aber ein 32-Bit-Programm.

32-Bit-Versionen (NT und 95^+) nutzen den Modus dagegen gut aus. Es gibt echtes Multitasking, und die Windows-Schnittstelle wird auf 32 Bit ausgebaut.

2.6.2 Entwicklungsstufen von Windows

Windows 3.x ist ein System für 16-Bit-Programme.

Das Multitasking ist nicht präemptiv, d.h. es arbeitet nicht mit Zeitscheiben oder I/O-orientierten Mechanismen, um zwischen Prozessen hin- und herzuschalten. Zwischen verschiedenen DOS-Anwendungen und dem normalen Windows-Betrieb muss der Benutzer (z.B. mit ALT TAB) wechseln. Windows-Programme müssen explizit die CPU-Kontrolle abgeben (mit dem System-Aufruf `yield` oder durch das Warten auf eine Nachricht). Wenn sie dies nicht tun, reißen sie also die komplette Systemkontrolle an sich.

Das Dateisystem wird immer noch von MS-DOS verwaltet. Für jeden Zugriff muss also in den virtuellen 8086-Modus geschaltet werden, und dort bestehen die üblichen Speicherbeschränkungen. Datei- und Gerätezugriffe werden unnötig verlangsamt. Wenn der virtuelle Speicher über normale Dateien abläuft, ist auch er hiervon betroffen.

Windows NT ist bei seiner Einführung weniger für den einfachen Anwender als für den Einsatz in Netzwerken und insbesondere als Server gedacht gewesen. Entsprechend umfangreich ist der Anteil von Sicherheitsvorkehrungen (Zugriff auf fremde Dateien, Ressourcen, Prozesse) am gesamten System.

Windows NT ist außerdem nicht nur für Intel-Systeme erhältlich, sondern auf andere Prozessor-Architekturen portiert worden. Immer ist es aber (mindestens) ein eigenständiges 32-Bit-Betriebssystem. Es ist nicht auf MS-DOS angewiesen, sondern bringt ein eigenes schnelles Dateisystem mit.

Es wird mit präemptivem Multitasking (mit Zeitscheiben) gearbeitet. Voneinander abgeschottete Prozesse und zusammengehörige Unterprozesse (Threads) sind möglich. Die Programmierschnittstelle ist eine echte 32-Bit-Schnittstelle namens Win32 geworden, die sehr umfangreiche Dienste bietet.

Windows 95 setzt ebenfalls nicht mehr auf MS-DOS auf, hat aber einige DOS-Strukturen mitgenommen und verwendet sie immer noch. Dennoch stellt es fast die komplette Win32-Schnittstelle zur Verfügung.

Das präemptive Multitasking stimmt bei 32-Bit-Anwendungen mit dem von NT überein. Bei der Abwicklung von DOS-Programmen ist Windows 95 kompatibler als NT. Das normale Dateisystem entspricht dem DOS-System, wird aber über eigene 32-Bit-Treiber geregelt.

Als kleine Bonbons wurden damit auch endlich die überall sonst selbstverständlichen „langen“ Dateinamen erlaubt. Die automatische Hardware-Erkennung soll auch dem Normalbenutzer den Einbau neuer Hardware erlauben, kämpft aber bis heute mit heftigen Schwierigkeiten.

2.6.3 Struktur der Windows-Systeme

Die Kommunikation von Anwenderprogrammen mit dem Fenstersystem erfolgt in jedem Fall über ein API (Application Programmer's Interface), eine Sammlung von einigen hundert Funktionen (gesammelt in Bibliotheken), die normal wie Unterprogramme aufgerufen werden. Die 16- und 32-Bit-APIs sind streng voneinander getrennt.

Einige typische API-Aufrufe sind zum Beispiel:

`CreateWindow`	neues Fenster anlegen
`SetMenu`	Menü für ein Fenster festlegen
`GlobalAlloc`	Speicher reservieren (Win32)
`CreateFile`	Datei anlegen (Win32)
`EnterCriticalSection`	kritische Sektion eines Threads signalisieren (Win32)

Mit MS-DOS und dem BIOS kann weiterhin über Interrupts gearbeitet werden. DOS-Interrupts sind dabei der Normalfall (sie werden vom Windows-Kernel abgefangen!). Unter älteren Versionen konnten BIOS-Interrupts dagegen die interne Verwaltung durcheinanderbringen. Durch die Interrupt-Virtualisierung bei 32-Bit-Systemen sind auch sie inzwischen wieder ungefährlich.

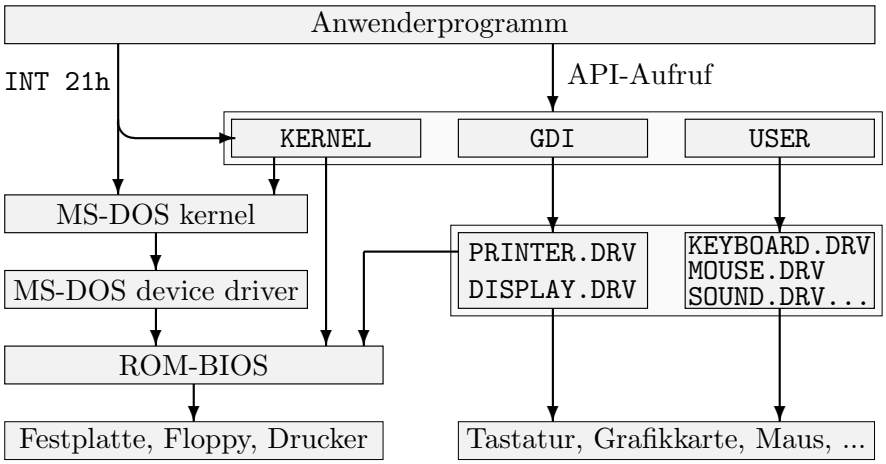
Die drei Kernbestandteile von Windows sind folgende drei Module:

16 Bit	32 Bit	
`KERNEL.EXE`	`KERNEL32.DLL`	Laden/Starten von Programmen, Prozess- und Speicherverwaltung
`GDI.EXE`	`GDI32.DLL`	Graphics Device Interface, grafische Grundoperationen
`USER.EXE`	`USER32.DLL`	verschiedenes, „der Rest“

16-Bit-Systemstruktur: Unten ist eine 16-Bit-Version von Windows dargestellt, die also auf MS-DOS aufsetzt.

Für I/O-Geräte gibt es Gerätetreiber `*.DRV`, die von `USER` und `GDI` (oder auch direkt vom Anwenderprogramm) aus aufgerufen werden. Datei-orientierte Treiber (für Platten und Drucker) rufen dazu die passenden BIOS-Routinen auf. Treiber für Bildschirm, Maus, Sound, etc. sprechen die Hardware direkt an.

Im Bild ist eine Installation gezeigt, bei der jedem Gerät ein eigener `*.DRV`-Treiber zugeordnet ist, der leicht einzeln ausgetauscht werden kann. Es gibt auch eine Sammelinstallation, wo die meisten Treiber in zwei Dateien zusammengefasst sind (`WINxxx.BIN` und `WINxxx.OVL`), die beim Booten schneller geladen werden können. Dann muss allerdings bei jedem Gerätewechsel Windows neu installiert werden!



Real-Mode-Probleme: Die Aufgaben der meisten `*.DRV`-Treiber hätten durchaus durch bestehende DOS- oder BIOS-Routinen übernommen werden können. Ihre konsequente Einführung hatte einen speziellen Grund.

Interrupts im Protected Mode werden intern anders beantwortet als Interrupts im Real Mode. Die im Bild angegebenen DOS-Aufrufe mit `INT 21h` landen deshalb immer im

Windows-Kernel, der dann entweder selbst die Verarbeitung übernimmt oder einen DOS- oder BIOS-Aufruf tätigt. Dann muss jedesmal in den Real Mode geschaltet und ein Real-Mode-Interrupt abgesetzt werden.

Es gibt nun aber zwar einen Maschinenbefehl, um vom Real in den Protected Mode zu gelangen – aber nicht umgekehrt, das geht nur mit einem Prozessor-Reset. Deshalb muss jedesmal ein absurder Umweg gegangen werden – beispielsweise muss dem System vorgegaukelt werden, der Benutzer hätte `CTRL-ALT-DEL` gedrückt, und der entsprechende Vektor muss vorher kurzfristig verbogen werden. Bei jedem Aufruf geht daher viel Zeit für dieses technische Detail verloren. Wenn mehrere Prozesse *irgendeinen* Real-Mode-Dienst benötigen, werden die Anfragen in *ein und dieselbe* Warteschleife gelegt, was zusätzlich verlangsamt.

Die `*.DRV`-Treiber sind deshalb so ausgelegt, so weit wie möglich ohne BIOS-Zugriff und Real Mode auszukommen. Das ist nicht immer möglich, und dann kommt es dennoch zu den beschriebenen Zeiteffekten.

32-Bit-Systeme: An der Grundstruktur des Systems ändert sich nichts Wesentliches. Die MS-DOS-Bestandteile sind aber eher in Windows integriert zu sehen, und es gibt eine neue Art von Treibern. Virtuelle Gerätetreiber `*.VxD` (virtual *x* device) dienen dazu, Geräte und Systemdienste zu virtualisieren, sodass Anwenderprogramme davon ausgehen können, den Rechner allein zu besitzen.

Die Aufgabe der `*VxD`s ist also die Verteilung der physischen Ressourcen auf all die Programme, die diese anfordern – und zwar geräteweise getrennt und damit intelligenter als die einfache Warteschlange von weiter oben. Außerdem sind Bestandteile des Kernels in virtuelle Treiber ausgelagert worden, beispielsweise `VMM32.VXD` für die virtuelle Speicherverwaltung.

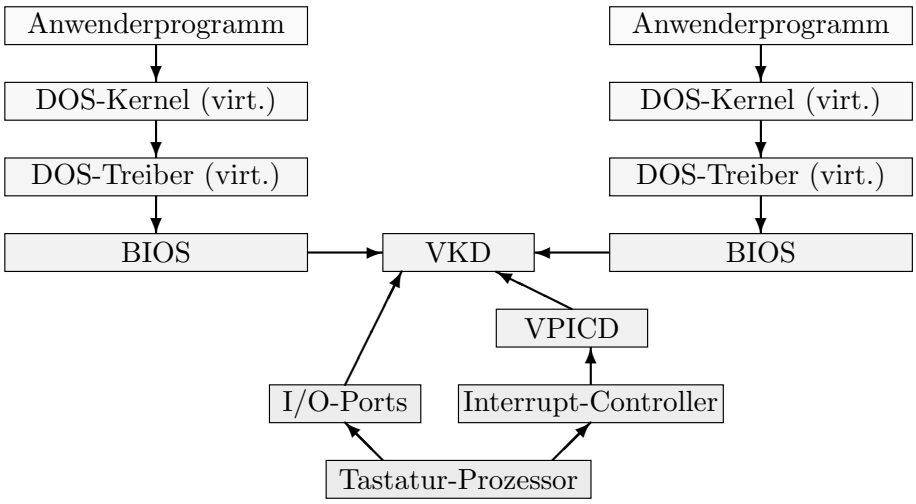
Es gibt beispielsweise `VFD.VXD` für die Floppy, `SERIAL.VXD` für die serielle Schnittstelle, `MSMOUSE.VXD` für die Maus, `PCI.VXD` für den PCI-Bus, etc.

Aus Kompatibilitätsgründen gibt es nun meist Treiber in zwei Stufen: ein `*.DRV`-Treiber, der aber oft nichts weiter tut, als die passenden Funktionen des `*VxD`-Treibers aufzurufen, also eine Schnittstellenanpassung von 16 nach 32 Bit vorzunehmen.

Der Kernel benutzt natürlich direkt die `*VxD`s, und auch echte 32-Bit-Programme können über die Win32-Funktion `DeviceIOControl` direkt auf ihre Dienste zugreifen.

Virtuelle Interrupts: Wichtig für die Virtualisierung der Hardware (inklusive der Interrupts!) ist die Möglichkeit vom 80386 aufwärts, die I/O-Maschinenbefehle `IN` und `OUT` für bestimmte Ports zu beschränken. Es wird dann durch eine solche Instruktion ein Protected-Mode-Interrupt ausgelöst, der von den `*VxD`-Treibern behandelt wird.

Für mehrere parallel laufende MS-DOS-Anwendungen (in eigenen virtuellen Maschinen) sieht unser Beispiel für Tastaturdrücke von oben schematisch wie folgt aus:

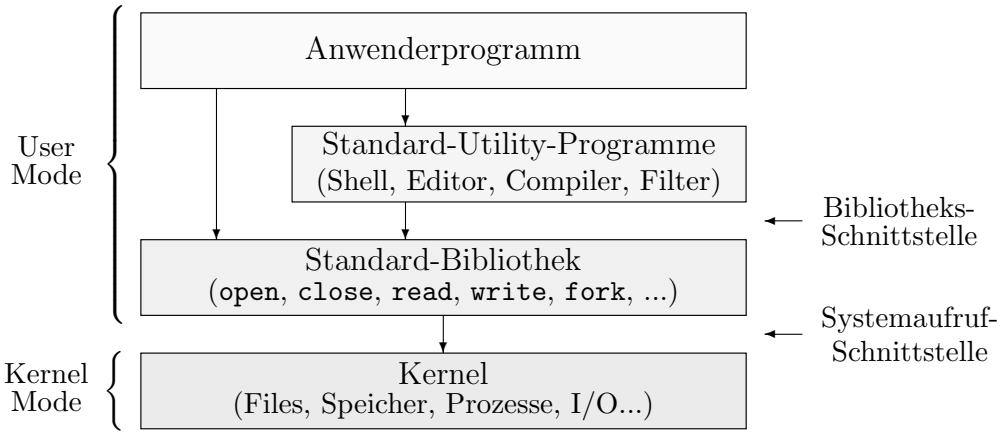


2.7 UNIX

UNIX ist jedes Betriebssystem, das einem der Standards entspricht, die im ersten Kapitel erwähnt wurden (hauptsächlich SVR4, BSD, POSIX). Die Definitionen sind völlig unabhängig von der Rechnerarchitektur. Vorgeschrieben wird ein Satz von Systemaufrufen und deren Funktionalität. Ein direkter Zugriff auf die Hardware ist normalerweise nicht möglich.

Es gibt wohl keinen Fall, in dem ein UNIX schrittweise für wachsende Hardware angepasst werden musste, wie das bei MS-DOS und Windows der Fall war. Daher ergeben sich auch nicht so komplizierte Strukturen von Zusammenarbeit mehrerer Teile unterschiedlicher Herkunft und Geschichte.

Abstraktere Funktionalität wie Kommandozeilen-Interpreter und Editor sind aus dem Kern ausgegliedert. Der Kern selbst ist allerdings sehr groß und beinhaltet viel Funktionalität (Prozesse, Speicher, Filesystem). Bei vielen UNIXen ist er in sich fast völlig ungegliedert und entsprechend unübersichtlich.



Die beiden angegebenen Schnittstellen müssen deutlich voneinander unterschieden werden! Bibliotheks-Aufrufe erfolgen durch normale Unterprogramm-Aufrufe, bei denen der User-Mode

der Programme nicht verlassen wird. Aufrufe des Kernels erfolgen nur innerhalb dieser Bibliotheken – meist durch Belegen von Registern mit Werten und Auslösen eines Software-Interrupts.

Die meisten normalen Benutzer identifizieren UNIX eher mit der *Benutzerschnittstelle*, d.h. mit der Shell und den Standard-Utilities, wie sie von der Shell aus aufgerufen werden. Diese hat zwar Parallelen in der Bibliotheks-Schnittstelle, liegt aber auf einer ganz anderen Ebene. Die Systemaufruf-Schnittstelle kann völlig anders geartet sein.

Die verschiedenen Standardisierungs-Versuche legen dabei immer nur das Bibliotheks-Interface fest. Das Kernel-Interface ist hardwareabhängig und kann von UNIX zu UNIX völlig unterschiedlich sein. Oft gibt es aber zwischen Bibliotheksfunktionen und Systemaufrufen starke Entsprechungen in der Funktionalität.

Als Beispiel geben wir fünf Bibliotheks-Routinen an, wie sie im System V definiert sind:

`create(name,amode)`
: Legt eine (leere) neue Datei mit dem Namen `name` (ggf. inklusive Pfad) an mit den Zugriffsrechten, die `amode` beschreibt.

`link(f1,f2)`
: Legt einen Hard Link an, d.h. einen „neuen Namen“ `f2` für die schon existierende Datei mit dem Namen `f1`.

`mount(filesys,dir,rflag)`
: Das neue Dateisystem `filesys` wird an der Stelle `dir` in den Dateibaum eingehängt. Wenn `rflag` nicht 0 ist, wird das System Read-Only.

`kill(pid,sig)`
: Schickt dem Prozess Nummer `pid` das Signal `sig` (z.B. das Signal zum Abbruch).

`_exit(status)`
: Beendet den aktuellen Prozess und gibt den Wert `status` an den erzeugenden Prozess zurück, der ihn mit `wait` oder `waitpid` abfragen kann. (Nicht verwechseln: Die `C`-Funktion `exit` räumt vor dem Aufruf *dieser* Routine noch Dateien und Speicher auf.)

2.7.1 L*i*nux

L*i*nux auf 80x86-Systemen existiert meistens nicht allein auf dem Rechner, sondern lebt mit DOS oder/und Windows zusammen. Wenn es gewünscht wird, gibt es aber auch Konfigurationsmöglichkeiten, ohne andere Systeme auszukommen.

Das Booten des Rechners erledigt (aus offensichtlichen technischen Gründen) auch bei L*i*nux das BIOS. Danach können mehrere Wege beschritten werden:

- Der Master Boot Record (von DOS oder Windows) wird durch einen speziellen L*i*nux-Loader ersetzt (evtl. bei mehreren Festplatten zwingend notwendig).
- Der Bootsektor einer anderen Partition auf der ersten Festplatte wird zum L*i*nux-Loader, der vom DOS-MBR gestartet wird.

- Es wird zunächst DOS oder Windows gebootet. Von dort aus wird der L*i*nux-Loader gestartet, der das schon gestartete Betriebssystem eliminiert.

Nach seinem Start ist L*i*nux jedenfalls nicht auf irgendwelche Dienste von DOS angewiesen. Zum Zugriff auf Ressourcen auf dem Motherboard (z.B. Platten/IDE-Controller) wird entweder das BIOS bemüht, oder die Ressourcen werden über das BIOS erkannt und später direkt angesprochen.

2.7.2 L*i*nux-Systemaufrufe

Unter L*i*nux auf PCs wird für Systemaufrufe ebenfalls ein Interrupt bemüht, nämlich der mit der Nummer 0x80=128. In Prozessorregistern werden die Nummer der gewünschten Funktion (`EAX`) und die eigentlichen Argumente übergeben. Auf diese Weise gelangt man am bequemsten in den privilegierten Modus.

Die Nummern aller Systemaufrufe findet man beispielsweise als Konstanten definiert in der Datei `include/asm/unistd.h`. Ein Auszug:

```
#define __NR_setup      0               #define __NR_waitpid    7
#define __NR_exit       1               #define __NR_creat      8
#define __NR_fork       2               #define __NR_link       9
#define __NR_read       3               #define __NR_kill       37
#define __NR_write      4               #define __NR_rename     38
#define __NR_open       5               #define __NR_mkdir      39
#define __NR_close      6               #define __NR_rmdir      40
```

Von `C` aus kann man einen der `syscall`-Makros (aus derselben Datei) verwenden, die automatisch Inline-Assemblercode erzeugen.

Es wird eine Kernel-Routine ab der Adresse `_system_call` aufgerufen. Üblicherweise ist sie in Assembler geschrieben (z.B. in der Datei `arch/i386/kernel/entry.S`). Sie rettet zunächst alle Prozessor-Register auf den Stack:

```
pushl eax     ; Funktionsnummer auf dem Stack merken
push gs       ; ab hier: Register retten
push fs
...
```

Danach überprüft sie, ob die angegebene Nummer einen gültigen Aufruf darstellt (sonst wird abgebrochen), und ermittelt dann die tatsächliche Adresse der aufzurufenden Routine, wobei sie auf die Tabelle `sys_call_table` zugreift. Schließlich ruft sie diese Routine auf:

```
cmpl $(NR_syscalls),eax         ; eax=Nummer, Vergleich mit größter Nummer
jae  ret_from_sys_call          ; zu groß ⇒ Abbruch
movl sys_call_table(eax,4),eax  ; Adresse aus Tabelle lesen
...
call *eax                       ; eigentlicher Aufruf
```

Die Tabelle ist auch in `entry.S` definiert:

```
.data
ENTRY(sys_call_table)
        .long SYMBOL_NAME(sys_setup)
        .long SYMBOL_NAME(sys_exit)
        .long SYMBOL_NAME(sys_fork)
        .long SYMBOL_NAME(sys_read)
        ...
```

Es sind einige zusätzliche Arbeiten notwendig, wenn sich der aufrufende Prozess gerade im Einzelschritt-Modus befindet. Außerdem werden ggf. an dieser Stelle „aufgeschobene“ Aktionen aus zurückliegenden Interrupts („bottom halves“) ausgeführt – die Details würden hier zu weit führen. Falls der Prozess durch den Aufruf nicht mehr weiterlaufen kann (`wait`, `sleep`) oder zufällig seine Zeit um ist, wird hier außerdem der Scheduler aufgerufen.

Ansonsten werden zum Schluss die Register des Prozesses restauriert, und es wird aus der Interrupt-Routine zurückgekehrt:

```
...
pop fs
pop gs              ; bis hier: Register retten
addl $4,esp         ; Funktionsnummer vom Stack nehmen
iret                ; Interrupt beenden
```

Durch Ergänzung der Aufruftabelle `sys_call_table`, die Anpassung von `NR_syscalls` und Neuübersetzung des Kernels kann man natürlich leicht zusätzliche Aufrufe implementieren. Das entstehende L*i*nux ist aber nicht mehr mit anderen Versionen kompatibel!

Beispiel: Ein sehr einfacher Systemaufruf ist `getpid`, der die Nummer des gerade aktuellen Prozesses liefert. Sein Code steht in `kernel/sched.c`. **`struct`** `task_struct *current` ist eine statische Variable im Kernel und zeigt auf die Verwaltungsstruktur des aktuellen Prozesses. Auch ohne deren genaue Kenntnis können wir den Code von `getpid` schon verstehen:

```
asmlinkage int sys_getpid(void)
 {
   return current->pid;
 }
```

Die Aufrufe für die ID des Benutzers, der Gruppe etc. sehen ähnlich aus. Die meisten anderen Systemfunktionen sind natürlich wesentlich komplexer ;-)

Beispiel 2: Interessant zu lesen ist auch der Code für den Systemaufruf `reboot` (Nummer 88), definiert in `kernel/sys.c`:

```
asmlinkage int sys_reboot(int magic, int magic_too, int flag)
 {
   if (!suser())
      return -EPERM;
   if (magic != 0xfee1dead || magic_too != 672274793)
      return -EINVAL;
   if (flag == 0x01234567)
      hard_reset_now();
```

```
  else if (flag == 0x89ABCDEF)
     C_A_D = 1;
  else if (!flag)
     C_A_D = 0;
  else if (flag == 0xCDEF0123)
   {
     printk(KERN_EMERG "System halted\n");
     sys_kill(-1, SIGKILL);
     do_exit(0);
   }
  else return -EINVAL;
  return 0;
}
```

Die Fehlerkonstanten `EPERM` (permission denied) und `EINVAL` (invalid) sind in `errno.h` definiert.

Man sieht, dass nur ein Prozess des Super-Users den Aufruf tätigen darf. Zur Vorsicht muss auch er einige kryptische magische Zahlen als Parameter übergeben. Abhängig vom Wert des letzten Parameters `flag` passiert folgendes:

0x01234567:	Es wird sofort ein Prozessor-Reset ausgelöst (gefährlich, da nicht aufgeräumt wird).
0x89abcdef:	Control-Alt-Delete wird zugelassen. Es wird nicht gebootet.
0:	Control-Alt-Delete wird abgeschaltet. Es wird nicht gebootet.
0xcdef0123:	Das System wird regulär heruntergefahren, indem der Startprozess das KILL-Signal erhält.

3 Grundlegende Konzepte

Wichtige Konzepte wie Benutzerverwaltung, Dateisystem, Shells und mehr werden ausführlich später in eigenen Kapiteln behandelt. Wir wollen aber an dieser Stelle als Vorwegnahme eine grobe Übersicht über sie geben, um von vornherein den Umgang mit dem System zu erleichtern, um sinnvolle Beispiele in den Abschnitten über Prozess- und Speicherverwaltung betrachten zu können, etc.

- Die angegebenen „Kommandos“ sind jeweils in der Shell (UNIX) bzw. an der „MS-DOS-Eingabeaufforderung“ einzugeben (genaueres zu diesen Kommando-Prozessoren folgt später). Kommandos und Funktionen, die sich mit *mehreren Prozessen* oder *Benutzern* befassen, beziehen sich – wenn nicht anders angegeben – immer auf UNIX. Die Befehls-Beschreibungen sind meist kurz gehalten, da man ausführlichere Informationen mit `man` bzw. `help` erhalten kann:

  `help` (unter DOS) ohne Parameter liefert eine menüartige Liste aller verfügbaren Einträge. `man` (bei UNIX) bietet diese Möglichkeit nicht, `xman` (unter X-Window) schon.

  Das UNIX-Manual ist in Abschnitte gegliedert, die beim Aufruf angegeben werden können. Das ist unerlässlich, wenn es in mehreren Abschnitten gleichbenannte Einträge gibt: „`man -s1 mktemp`“ findet den Shell-Befehl, „`man -s3 mktemp`“ die Bibliotheks-Funktion (unter L*i*nux entfällt das „`-s`“).

  Die Abschnitte sind wie folgt numeriert (die ursprünglichen AT&T-Nummern stehen in Klammern):

User's Reference Manual		
1	Commands/Applications	`bash cp g++ man`
6	Games	
Programmer's Reference Manual		
2	System Calls (Kernel Calls)	`open fork waitpid execv`
3	Subroutines (System Libraries)	`fopen printf strcpy system`
5	File Formats	`crontab passwd termcap`
7 (4)	Miscellaneous Facilities	
Administrator's Reference Manual		
4 (7)	Devices/Special Files	`console null mouse`
8 (+1M)	System Administration/Maintenance	`cron shutdown useradd`

- Die meisten angegebenen UNIX-Systemaufrufe sind POSIX-kompatibel, und ihre Prototypen und die zugehörigen symbolischen Konstanten sind in der Header-Datei `unistd.h` definiert.

  Wenn spezielle Typen für Systemgrößen benutzt werden, enden ihre Namen meist auf „`_t`“, beispielsweise `pid_t` (Prozess-ID), `uid_t` (User-ID), `key_t` (Key bei IPC), `ino_t` (inodes). Alle diese Typen sind in der Datei `sys/types.h` definiert.

  Die entsprechenden **`#include`**-Angaben werden wir manchmal aus Platzgründen weglassen.

- Wenn für eine Funktion angegeben ist, dass sie von DOS aus aufrufbar ist, gilt das – wenn nicht ausdrücklich anders angegeben – auch für Windows.

- C-Compiler-Hersteller für DOS- bzw. Windows haben im Lauf der Zeit viele Systemaufrufe bei UNIX abgeschaut. Leider sind die entstandenen Bibliotheksfunktionen aber fast immer in völlig anderen Header-Dateien deklariert. Deshalb listet die folgende Tabelle vorab die *Dateisystem-bezogenen* Aufrufe mit dem jeweils zugehörigen Header auf. Die DOS-Angaben beziehen sich auf Borland- und Microsoft-Compiler. Die Funktionen selbst werden in den folgenden Abschnitten einzeln besprochen.

  Mit $^+$ markierte Funktionen sind von Windows aus nicht benutzbar.

Funktion	UNIX	DOS/Win	Funktion	UNIX	DOS/Win
access	unistd.h	io.h	getdisk	–	dir.h
chdir	unistd.h	dir.h	link	unistd.h	–
chmod	sys/stat.h	io.h	lseek	unistd.h	io.h
chown	unistd.h	–	mkdir	sys/stat.h	dir.h$^+$
close	unistd.h	io.h	open	fcntl.h	io.h
closedir	dirent.h	dirent.h	opendir	dirent.h	dirent.h
creat	fcntl.h	io.h	popen	stdio.h	–
dup	unistd.h	io.h	read	unistd.h	io.h
dup2	unistd.h	io.h	readdir	dirent.h	dirent.h
fchdir	unistd.h	–	readlink	unistd.h	–
fchmod	sys/stat.h	–	remove	stdio.h	stdio.h
fchown	unistd.h	–	rewinddir	dirent.h	dirent.h
fdopen	stdio.h	stdio.h	rmdir	unistd.h	dir.h$^+$
filelength	–	io.h$^+$	setdisk	–	dir.h
fileno	stdio.h	stdio.h		sys/stat.h	sys.h
fstat	sys/stat.h	sys.h	symlink	unistd.h	–
getcurdir	–	dir.h	unlink	unistd.h	io.h
getcwd	–	dir.h	write	unistd.h	io.h

- Wenn eine Systemfunktion, die von C aus aufgerufen wird, einen Fehler auslöst, wird (auf jedem System!) die externe Variable **int** errno (aus errno.h) auf die Fehlernummer gesetzt.

  – In errno.h sind symbolische Konstanten für die Fehlernummern definiert, mit denen man errno vergleichen kann, z.B. EPERM (permission, „Not super-user“), ENOENT (no entry, „No such file or directory“), etc.

  – Die Funktion **char* strerr(int e)** (aus string.h) liefert einen String zurück, der Fehlermeldung Nummer e im Klartext enthält.

  – Die Funktion perror (aus stdio.h) gibt die zu errno passende Fehlermeldung auf den Standard-Fehlerkanal aus, genauer:

```
void perror(const char *s)
 {
   fprintf(stderr,"%s: %s\n",s,strerr(errno));
 }
```

Bei folgendem Programmstück könnte die Ausgabe z.B. sein „`sorry: No such file or directory`“:

```
FILE *fp=fopen("[?*?*?]","r");
if (fp==0) perror("sorry");
```

3.1 Mehrbenutzersysteme

Wenn ein System von mehreren Personen genutzt werden soll oder gar in einem Netzwerk zugänglich ist, ist es im Hinblick auf Datensicherheit unerlässlich, eine Benutzer-Verwaltung in irgendeiner Form zu realisieren. Ressourcen (hauptsächlich Dateien) sind dann einem bestimmten Besitzer zugeteilt, der allein auf sie zugreifen kann oder bestimmten kann, wer sonst dieses Recht besitzen soll.

Die Angaben in diesem Abschnitt beziehen sich zunächst meist auf UNIX. Auf Windows-Spezialitäten wird später eingegangen.

3.1.1 Benutzer

Ein *Account* auf einem System ist eine Zugangsberechtigung, vergleichbar mit einem Konto, was Rechenzeit, Hauptspeicher- und Datei-Verbrauch angeht. Der *Besitzer* eines Accounts heißt *Benutzer* (User).

Zur Überprüfung der Identität eines Benutzers muss dieser sich üblicherweise beim System mit seinem Benutzernamen (Login Name, User Name, 3 bis 8 Buchstaben) *anmelden* (*einloggen*, *log-in*) und durch die Eingabe eines *Passworts* ausweisen. (Mehr zu Passwörtern später im Abschnitt zur Systemverwaltung.)

Für jeden Benutzer können spezielle Einstellungen an dessen Arbeitsumgebung getätigt werden, wie z.B. der „Eintrittsort“ in das Dateisystem, automatisches Starten von Programmen (insbesondere von grafischen Oberflächen), etc.

Unter UNIX werden für die Benutzer *Nummern* vergeben („User ID“, UID), anhand derer sie unterschieden werden. Dateien erhalten beispielsweise als besonderes Attribut die UID des erzeugenden Benutzers.

Oft sind nicht nur echte Personen „Benutzer“ mit einer UID. Auch speziellen Prozessen (wie Dämonen, Server) oder Prozessgruppen, die regelmäßig oder permanent im Hintergrund arbeiten, wird oft eine eigene UID gegeben. Sie erhalten meist Nummern von 1 bis 99, der Administrator erhält 0, normale Benutzer Nummern von 100 bis 50000.

Die „eigene“ Benutzer-ID erfährt man durch folgenden Systemaufruf:

	UNIX
`uid_t getuid(void);`	
gibt die Benutzer-ID des „Besitzers“ des aktuellen Prozesses zurück	

Das Angeben bzw. Wechseln des Passworts ist von System zu System unterschiedlich. In UNIX kann jeder Benutzer mit dem Kommando `passwd` (oder systemabhängig bei Rechnernetzen auch `yppasswd`, `nispasswd`, o.ä.) sein Passwort ändern, eventuell allerdings nicht beliebig schnell

hintereinander. Möglicherweise wird auch ein „Passwort-Aging“ verwendet, das bei zu alten Passwörtern zunächst warnt und sie schließlich für ungültig erklärt.

Das Passwort soll 6 bis 8 Stellen haben und mindestens zwei Buchstaben und eine Ziffer enthalten. Viele Versionen von `passwd` lassen sich aber auch zu Passwörtern überreden, die sich nicht an diese Konvention halten, wenn man es nur lange genug versucht.

Unter UNIX gibt es auch „spezielle User“ oder Pseudo-User, die nicht wirklichen Personen entsprechen. Beispielsweise hat der Benutzer `lp` Zugriff auf die druckerbezogenen Strukturen, der Benutzer `uucp` auf das UUCP-System (zum Datei- und Mailaustausch), etc. Auf diese Weise braucht man (aus Sicherheitsgründen) den passenden Prozessen nicht gleich *alle* Rechte eines Administrators zu geben.

Genaueres zu Grundsätzen bei der Verwaltung mehrerer Benutzer folgt später im Kapitel zu Systemverwaltung.

3.1.2 Der Administrator

Unter einem System-Administrator versteht man üblicherweise eine Person, die dafür verantwortlich ist, dass sich das System zu jeder Zeit in einem einwandfreien, benutzbaren Zustand befindet. Im Hinblick auf die Konsistenz des Systems hat sich herausgestellt, dass es sinnvoller ist, einer *einzelnen* Person diese Verantwortung zu übertragen und nicht einer kleinen Gruppe.

Er

- installiert das Betriebssystem und später ggf. neue System-Versionen
- installiert neue Hard- und Software
- vergibt Nutzerrechte an andere Personen und legt deren Rechte fest
- sichert wichtige Datenbestände regelmäßig (Backup)
- überwacht die Sicherheit des Systems
- überwacht die Performance und leitet ggf. Korrekturmaßnahmen ein
- passt ggf. Einstellungen an die Wünsche der Benutzer an
- sorgt bei Hardware- oder Installationsproblemen möglichst schnell für Abhilfe

Der Administrator sollte sich nur zum Zweck administrativer Arbeiten als solcher beim System anmelden. Durch seine umfassenden Rechte könnte er sonst leicht unabsichtlich Inkonsistenzen erzeugen oder Daten zerstören!

Der Status des Administrators spiegelt sich meist auf Systemebene in Form eines speziellen Benutzers mit besonderen Rechten wieder. Unter UNIX gibt es dazu den Benutzer `root`, der auch Super-User genannt wird.

- Der Name rührt daher, dass das Startverzeichnis dieses Benutzers ursprünglich das Wurzelverzeichnis `/` des Dateisystems war. Heutzutage gibt es stattdessen meist ein eigenes Verzeichnis `/root`. In der Shell erhält er standardmäßig das Prompt `#` (statt `$`, `%` o.ä.).

- `root` hat Zugriffsrechte auf alle Dateien, auch die Systemdateien der Benutzerverwaltung, was es ihm ermöglicht, eine Person als neuen Benutzer einzutragen. Er ist außerdem der einzige, der das System mit Kommandos wie `shutdown` oder `halt` geordnet herunterfahren kann.

- Aus Sicherheitsgründen werden alle Anmeldungen als Administrator in einer Datei wie z.B. `/var/adm/sulog` oder `/var/log/messages` abgelegt.

Der Superuser kann unter anderem Folgendes:

- das System herunterfahren (`shutdown`)
- die Identität beliebiger anderer User annehmen (`su`)
- auf beliebige Dateien und Geräte zugreifen
- beliebige Programme ausführen
- Prozess-Prioritäten verändern (`nice`)
- beliebige Signale an beliebige Prozesse versenden (`kill`)
- Dateisysteme einhängen (`mount`, `umount`)
- CPU-Zeit und Speicherbedarf quotieren

Genaueres zu den speziellen Rechten der Administratoren erfahren wir später z.B. in den Kapiteln über Dateisysteme und Systemverwaltung.

3.1.3 Identitätswechsel

Bei einigen Systemen kann man seine Identität nur durch Verlassen und Wiedereinloggen verändern (Windows 95 o.ä.). Unter UNIX kann ein Benutzer mit dem Systemkommando „`su` *username*“ (set user, substitute user) *zwischenzeitlich* seine Identität ändern. Das hat Auswirkungen nur auf den *Prozess*, der das `su` absetzt (und auf dessen Kinder).

- Wenn man nicht gerade der Super-User ist, muss man natürlich das Passwort des neuen Benutzers angeben. Auf manchen Systemen braucht man eine spezielle Erlaubnis, um `su` ausführen zu dürfen. Aus Sicherheitsgründen werden fehlgeschlagene `su`-Versuche aufgezeichnet (z.B. in `/var/log/messages`, als „`FAILED SU`“ oder „`BAD SU`“).

- Wenn kein Name hinter „`su`“ angegeben wird, wird automatisch `root` angenommen. Fälschlicherweise wird `su` deshalb oft auch als „Super-User-Kommando“ bezeichnet.

- **Vorsicht:** Normalerweise bewirkt ein `su` lediglich das Ändern der UID. Es wird beispielsweise nicht in das Heimatverzeichnis des neuen Benutzers gewechselt. Wenn dieser keine Rechte hat, sich im aktuellen Verzeichnis aufzuhalten, wird man daraus herausgeworfen oder besitzt kein aktuelles Verzeichnis o.ä. Man sollte daher als erstes Argument „`-`“ angeben. Die gesamte „Umgebung“ (wie Heimatverzeichnis, Shell, etc.) wird dann so angelegt, als hätte sich der neue Benutzer frisch eingeloggt.

- **Vorsicht 2:** Wenn man Superuser werden möchte, sollte man das Kommando immer mit vollem Pfad angeben: „`/bin/su`“. Ansonsten führt „`su`“ eventuell ein gleichlautendes Kommando im aktuellen Verzeichnis aus (je nach Pfad-Variable) – und das könnte eine Hacker-Falle sein, die das `root`-Passwort abfängt!

3.1.4 Benutzergruppen

Innerhalb einer Arbeitsgruppe ist es sinnvoll, wenn nicht alle Benutzer *alle* ihre Daten vor den anderen verstecken. Einige Daten, die für alle Mitglieder sinnvoll sind, sollten auch für alle sichtbar und zugreifbar sein.

Entsprechend werden Benutzer unter UNIX in Benutzer-Gruppen zusammengefasst. Für jede Datei kann festgelegt werden, welche Rechte die anderen Mitglieder der Gruppe (neben dem Besitzer der Datei) haben sollen.

Ein Benutzer kann Mitglied beliebig vieler Gruppen sein, ein Prozess ist dagegen eindeutig einem Benutzer und einer Gruppe zugeordnet (nur wenige UNIXe erlauben auch mehrfache Gruppenzugehörigkeit von Prozessen). Wenn ein Benutzer seine *aktuelle* Gruppenzugehörigkeit verändern will, kann er das mit dem Kommando „`newgrp` *grpname*“ tun. Eventuell muss dazu das Gruppenpasswort eingegeben werden.

Wenn ein System mehrere Administratoren hat, kümmert sich meistens ein Administrator um ganze Gruppen. Die entsprechende Zuordnung Gruppe → Administrator wird auch (allein für Informationszwecke) im System festgelegt.

Auch die Gruppen erhalten eine Nummer, die „Group ID“ (GID). Die eigene fragt man mit folgendem Systemaufruf ab:

UNIX
`gid_t getgid(void);` gibt die Group-ID des „Besitzers“ des aktuellen Prozesses zurück

Die Nummern 0 bis 99 sind wieder intern reserviert, 100 ist meist eine Sammelnummer für Normalbenutzer. Die Zahlen 101 bis 50000 können frei verwendet werden.

Wie es „spezielle Benutzer“ gab, so gibt es auch „spezielle Gruppen“. Beispielsweise hat die Gruppe `tty` Zugriff auf alle Terminals, `lp` auf die Drucker, etc. Entsprechende Programme laufen mit solchen Gruppenzugehörigkeiten.

Durch Gruppen kann man auch den Zugriff normaler Benutzer auf bestimmte Programme einschränken. Beispielsweise ordnet man alle Programme und Dateien einer bestimmten Anwendung einer Gruppe desselben Namens zu, etwa `dbase` oder `netscape` (siehe den folgenden Abschnitt über das UNIX-Dateisystem). Benutzer, die auf diese Programme zugreifen wollen, müssen zur entsprechenden Gruppe gehören. Das bedeutet keine Einschränkung, da ja Benutzer beliebig vielen Gruppen angehören können.

3.2 Dateisystem

Das Handhaben von Daten auf externen Speichern ist der Systembereich, bei dem normale Benutzer am direktesten mit dem Betriebssystem in Verbindung treten. Aufgaben des Betriebssystems in diesem Bereich sind die Handhabung der unterliegenden Geräte, die Verteilung der Ressourcen auf die Benutzer und die Sicherung der Daten gegenüber unerlaubtem Zugriff.

Der zentrale Begriff ist hier natürlich die **Datei** (das File). Eine Datei ist eine benannte Sammlung von Daten auf einem Speichermedium. Inwiefern die Daten (logisch) zusammengehören, wird durch ihren Erzeuger definiert.

Es finden Abbildungen in zwei Richtungen statt:

- Dateien werden auf physische Geräte abgebildet (Verteilen von Byteströmen auf die Gegebenheiten auf dem Gerät, wie Plattenoberflächen, Zylinder, Sektoren, etc.)

- logische Strukturen werden auf Dateien abgebildet (komplexe Datenstrukturen werden in einfache „Byteströme“ geglättet)
- abstrakte Mechanismen werden auf Datei-Mechanismen abgebildet (Zugriff auf Fenster, Drucker, Mäuse, Prozessdaten)

Etwas abstrakter kann man eine Datei als Datensammlung verstehen, in der Form, wie sie dem Benutzer des Betriebssystems zusammen mit den typischen Datei-Operationen zur Verfügung gestellt wird. Es ist dabei unerheblich, auf welchem Medium die Daten lagern, ob sie überhaupt physisch vorhanden sind, etc. Jedes System-Objekt, auf das die Operationen wie *create*/*delete*, *read*/*write* (und ggf. *seek*, *truncate* und mehr) anwendbar sind, ist eine Datei.

Die klassischen Probleme, mit denen sich ein Dateisystem beschäftigt, sind folgende:

- Speicher-Verwaltung (auf den externen Medien)
- Benennung (Mechanismen, über die Daten angesprochen werden)
- Sicherheit (Zugriffsschutz, wer darf welche Daten ansprechen)
- Zuverlässigkeit (Daten dürfen nicht verlorengehen)
- Performance (geschickte Verwaltung kleiner und großer Datenmengen)

3.2.1 Dateien

- Je nach System sind Dateien

  unstrukturiert (featureless) – eine sequentielle Folge von Zeichen/Binärworten

  (logisch) strukturiert – beispielsweise mit Blockstruktur, nach Schlüsseln sortiert über einen Index, mit Kapitel/Paragraphen-Struktur bei Texten, etc.

  Unter MS-DOS, Windows und UNIX gibt es auf der Systemebene nur unstrukturierte Dateien. In allen Fällen ist eine Datei eine Folge von (8-Bit-)Bytes.

  Die Byteströme können unterschiedliche Dinge wie ausführbare Programme (Objektcode), Quelltexte, andere Texte, Binärdaten etc. darstellen. Die entsprechenden Anwendungen (Loader, Compiler, Textverarbeitung, etc.) müssen ihre eigenen passenden Strukturen auf die Byteströme abbilden.

- Man macht zwar manchmal eine grobe Unterscheidung zwischen *binären* und zeichenorientierten Dateien. In letzteren Fall gibt es eine primitive *Zeilenstruktur* auf der Datei, die i.Allg. durch wenige spezielle Steuerzeichen festgelegt wird (`CR`=carriage return, `LF`=LineFeed). Diese Unterscheidung hat aber (fast) nie Auswirkungen auf das *eigentliche Dateisystem* des Kerns, sondern wird auf der Ebene von Anwenderprogrammen (Editoren, Konvertierungsprogrammen, etc.) vorgenommen.

  Eine Ausnahme sind ggf. Druckertreiber, die sich weigern könnten, Binärfiles zu drucken, oder versuchen, das Format der Files genauer zu erkennen und unterschiedlich zu reagieren (Textdruck, Grafikdruck, etc.).

- Außerdem gibt es Unterschiede beim **Zugriff** auf die Dateien:

  sequentiell (in der physischen Reihenfolge)

  wahlfrei (random access, Zugriff auf beliebige Stellen – ggf. mit Wartezeiten)

Eine Abstraktionsebene höher gibt es dann noch den Zugriff **per Schlüssel**) (keyed), also eine Form von Assoziativspeicher, bei der Elemente über Teile ihres Inhalts angesprochen werden. (Große kommerzielle IBM-Rechner stellen so etwas auf Betriebssystem-Ebene zur Verfügung.)

- UNIX unterscheidet bei Dateien (von Directories abgesehen) zwischen folgenden Typen:

regular file	normale Datei (Datensammlung auf einem Massenspeicher)
special file	Gerätedatei, Schnittstelle zum Kernel-Treiber
named pipe	FIFO-Kommunikationsmechanismus zwischen Prozessen
symbolic link	Verweis, Verknüpfung mit einem anderen Datei-Objekt

  Die Dateien der vier Typen sind auf Dateisystem-Ebene prinzipiell unterschiedlich aufgebaut. MS-DOS hat keine solchen Unterscheidungen. Am nächsten kommt es der UNIX-Unterteilung durch folgende Attribute, die eine Datei haben kann:

system	System-Teil (DOS), Gerätetreiber, etc.
hidden	versteckt (wird normalerweise nicht aufgelistet/kopiert, etc.)

  Windows (ab 95) hat zwar symbolische Links eingeführt, die aber für das Dateisystem ganz normale Dateien sind, die durch ihre Extension `.LNK` als Links identifiziert werden. Außerdem beziehen sie sich immer auf eine absolute Adresse, sodass beim Verschieben von Ordnern leicht die Verbindung verlorengeht.

Die wichtigsten Datei-bezogenen Kommandos sind folgende (genaueres per `help`/`man`):

	UNIX	**MS-DOS**
Edieren einer Datei	`vi, emacs, ed`	`edit`
Anzeigen einer Datei	`cat`	`type`
Anzeigen, seitenweise	`more, less`	`more`
Umbenennen einer Datei (alt neu)	`mv`	`ren`
Kopieren einer Datei (von nach)	`cp`	`copy`
Löschen einer Datei	`rm`	`del`

3.2.1.1 Dateinamen

Eine Datei wird im System über einen symbolischen Namen identifiziert. Die Handhabung der dafür erlaubten Zeichenkombinationen ist in den Systemen ganz unterschiedlich.

- Das klassische MS-DOS-Format 8.3 (für Name.Endung) stammte noch aus CP/M, wo es aus Platzgründen auf den damals winzigen Medien eingeführt wurde. Leider wurde es aus Kompatibilitätsgründen viel zu lang beibehalten.

  Die Endung wird dazu herangezogen, Schlüsse über die Dateistruktur und -Funktion zu ziehen. Das System und die meiste Software wird durcheinandergebracht, wenn man von den entsprechenden Konventionen abweicht.

  Windows-Versionen ab Windows 95 bzw. Windows NT haben die 8.3-Einschränkung nicht. Windows ordnet darüber hinaus über die Endung den Dateien die Programme zu, von denen es annimmt, das es sie am besten bearbeiten kann (vom Benutzer konfigurierbar).

- Unter UNIX dürfen Dateinamen sehr lang sein (von System zu System unterschiedlich, früher 14, inzwischen meist 255 Zeichen). Es sind beliebige Zeichen außer `/` erlaubt – die Spezialzeichen der Shells wie `*?&` usw. (s.u.) machen aber Probleme bei der Befehlseingabe und sollten vermieden werden.

  Dateinamen sind nicht strukturiert. Es brauchen keine Punkte vorzukommen, es dürfen beliebig viele vorkommen. Endungen wie `.c` für `C`-Programme, `.a` für Archive etc. sind nicht vorgeschrieben, sind aber jahrelange Konvention und erleichtern das Arbeiten mit den entsprechenden Programmen (Compiler, Archivierer o.ä.).

In UNIX-Dateisystemen wird streng zwischen Groß- und Kleinschreibung unterschieden. In MS-DOS sind überhaupt nur Großbuchstaben erlaubt. In Windows ab 95 sind Kleinbuchstaben erlaubt – Dateinamen, die sich nur in Groß-/Kleinschreibung unterscheiden, bezeichnen aber dieselbe Datei.

3.2.1.2 Zugriff aus Programmen

Eine Datei wird im Dateisystem eindeutig durch ihren Namen (bzw. Pfad) identifiziert. Es wäre aber umständlich und langsam, in Programmen bei jedem Zugriff diese Identifikation durchführen zu lassen. Beim Schreiben jeder kleinen Datenmenge müsste das System über den Dateinamen erneut herausfinden, wo (z.B. in welchen Blöcken auf der Platte) die Daten untergebracht sind.

Daher wird eine Serie von Lese-/Schreib-Operationen auf einer Datei üblicherweise initialisiert („**open**“, die Datei wird „**geöffnet**“), wodurch eine systeminterne Struktur angelegt wird, die eine möglichst schnelle Manipulation des unterliegenden Geräts ermöglicht. Man erhält irgendeine Art von **Schlüssel** (s.u.), über den man auf die geöffnete Datei Zugriff hat. Am Ende der Zugriffsserie wird die Datei „**geschlossen**“ („**close**“), die interne Struktur wird aufgelöst.

Durch das Öffnen und Schließen von Dateien können auch elegant Zugriffe mehrerer Benutzer auf dieselbe Datei geregelt werden. Das mehrfache gleichzeitige Lesen ist beispielsweise unproblematisch, schreiben dagegen sollte nur ein Benutzer zur selben Zeit tun dürfen. Auch deshalb muss beim Öffnen die Zugriffsart mit angegeben werden.

3.2.1.3 Zugriff in `C`

Ein File, als Instanz eines abstrakten Datentyps betrachtet, bietet wünschenswerterweise mindestens Operationen zum **Anlegen/Entfernen**, **Öffnen/Schließen**, **Schreiben/Lesen** und ggf. **Positionieren** (Angabe der Lese-/Schreibposition innerhalb der Datei).

Unter `C` gibt es dafür die bekannten normierten, system-unabhängigen Funktionen in der Standard-Bibliothek (deklariert in `stdio.h`) – in der obigen Reihenfolge:

(`fopen`)/`remove`, `fopen`/`fclose`, `fread`/`fwrite` und `fseek`.

Der Zugriffsschlüssel ist vom Typ `FILE*`.

Die Lese-/Schreib-Operationen `fread`/`fwrite` sind unformatiert und behandeln Bytefolgen. Es gibt natürlich die beliebten formatierten Versionen `fscanf`/`fprintf`. Sie haben aber nichts mit

der Ablage der Daten im Dateisystem zu tun, sondern implementieren eine Formatinterpretation auf Anwenderebene und werden hier nicht betrachtet.

Folgendes Beispiel schreibt binär 4 Fließkommazahlen in eine neue Datei:

```
#include <stdio.h>
#include <math.h>

int main(void)
 {
   static double data[]= { M_PI, M_SQRT2, M_E, M_LN2 };

   FILE *myfile=fopen("float.dat","wb");

   if (myfile==0)
      fprintf(stderr,"Ich kann die Datei nicht öffnen!\n");
   else
    {
      if (fwrite(data,sizeof(double),4,myfile)!=4)
         fprintf(stderr,"Schreibfehler!\n");
      fclose(myfile);
    }
 }
```

Diese Art von I/O ist immer gepuffert (es werden keine einzelnen Bytes physisch gelesen/geschrieben, es wird blockweise gearbeitet). Außerdem wird zwischen Binär- und Text-Dateien unterschieden, und ggf. werden passende Byte-Umsetzungen vorgenommen.

3.2.1.4 Zugriff in C++

In C++ sind Dateien als Ströme abstrahiert, also Objekte der Klasse `ios` bzw. einer Unterklasse davon, z.B. `istream` zur Eingabe und `ostream` zur Ausgabe. Diese Objekte sind auch die Zugriffsschlüssel.

Die Operationen sind echte Klassen-Methoden – in der obigen Reihenfolge:

(`ios`)/`remove`, (`ios`)–`open`/(`~ios`)–`close`, `>>`/`<<` und `seekg`/`seekp`.

Objekte bringen ihre eigenen Formatierungen über überladene Operatoren `>>` und `<<` mit. Hier kann nur auf eine C++-Einführung verwiesen werden, z.B. das Skript „Objektorientiertes Programmieren mit C++ und JAVA“.

Das Beispiel von oben sieht in C++ wie folgt aus:

```
#include <fstream.h>
#include <math.h>

int main()
 {
   static double data[]= { M_PI, M_SQRT2, M_E, M_LN2 };

   ofstream myfile("float.dat",ios::binary);
```

```
  if (!myfile) cerr << "Ich kann die Datei nicht öffnen!" << endl;
  else
   {
     if (myfile.write(data,sizeof(data)))
        cerr << "Schreibfehler!" << endl;
     myfile.close();
   }
}
```

3.2.1.5 Zugriff mit UNIX-Systemaufrufen

Auf **Betriebssystem-Ebene** gibt es aber immer einen niedrigeren Level von Dateibehandlung. Die `stdio`/`iostream`-Ebene ist dann um sie herumgelegt und verwendet intern die Systemfunktionen. Manchmal (z.B. bei Prozesskommunikation) ist es aber unumgänglich, diese Low-Level-Ebene zu benutzen.

Unter UNIX ist diese Ebene die der direkten **Systemaufrufe** (Operationen) und der **File-Deskriptoren** (Zugriffsschlüssel). Die Operationen in der Reihenfolge von oben sind:

`creat`(*sic*)/`remove`, `open`/`close`, `read`/`write` und `lseek`.

Systemweit wird eine Liste mit Verwaltungsstrukturen für alle offenen Dateien geführt (Ort auf der Platte, Schreib-/Leseposition, etc.). Jeder Prozess hat eine eigene Tabelle für Dateien, die er geöffnet hat, die nur Verweise auf die Systemliste enthält. Er spricht seine Dateien über ihre Nummer in seiner Tabelle an. Diese Nummer heißt File-Deskriptor und ist einfach eine ganze Zahl ≥ 0.

Die Tabelle wird nicht dynamisch geführt, sondern hat eine feste Größe (`OPEN_MAX`, definiert in `limits.h`, z.B. 64). Mehr Dateien kann ein Prozess nicht gleichzeitig geöffnet haben.

Die sechs klassischen Datei-Systemaufrufe (POSIX-kompatibel, auch auf Nicht-UNIX-Systemen meist so implementiert) sind folgende:

UNIX/DOS
`int open(const char *pathname, int flags, mode_t mode);` **öffnet** eine Datei. Der neue File-Deskriptor wird zurückgegeben (oder -1 bei Fehlern). `flags` gibt die Zugriffsart an: `O_RDONLY` (nur Lesen), `O_WRONLY` (nur Schreiben) oder `O_RDWR` (beides). Zusätzliche Angaben können mit '`|`' dazugeodert werden, u.a.: `O_CREAT` die Datei soll angelegt werden, falls sie nicht schon existiert `O_TRUNC` die Datei soll auf Länge 0 gekürzt werden, falls sie schon existiert `O_APPEND` neue Schreibdaten sollen immer an das Ende angehängt werden
`int creat(const char *name, mode_t mode);` **legt** eine leere Datei **an**. `mode` beschreibt die Zugriffsrechte (dazu später). Bei Misserfolg wird -1 zurückgegeben, sonst der gültige neue File-Deskriptor.
`int close(int fd);` **schließt** eine geöffnete Datei.

UNIX/DOS
`ssize_t read(int fd, void *buf, size_t count);` **liest** aus einer offenen Datei `count` Bytes in den Puffer `buf`.
`ssize_t write(int fd, const void *buf, size_t count);` **schreibt** in eine offene Datei `count` Bytes aus dem Puffer `buf`.
`off_t lseek(int fildes, off_t offset, int whence);` verschiebt die Schreib-/Leseposition in einer offenen Datei. Die Verschiebung ist relativ zum Dateianfang (`whence`=`SEEK_SET`), zur momentanen Position (`SEEK_CUR`) bzw. zum Dateiende (`SEEK_END`).

Die Aufrufe existieren alle auch in DOS (ab Version 3). Die File-Deskriptoren werden dort „Handles“ genannt.

Es gibt keine Systemaufrufe mit *Formatierung* wie `printf` und `scanf` bei `stdio`.

In diesem Zusammenhang sei noch die ANSI-C-Bibliotheksfunktion `remove` aufgeführt (die in UNIX intern auf den Systemaufruf `unlink` zurückgeführt wird, siehe Abschnitt 3.2.4):

ANSI-C
`int remove(const char *path);` **löscht** die angegebene Datei (unter UNIX siehe genauer den Abschnitt 3.2.4!)

Das Beispiel von oben sieht mit Low-Level-Dateien so aus:

```
#include <unistd.h>        /* Standard-Unix-Systemaufrufe         */
#include <fcntl.h>         /* File-Control, fur O_WRONLY, etc.    */
#include <math.h>

int main(void)
 {
   static double data[]= { M_PI, M_SQRT2, M_E, M_LN2 };

   int fd=open("float.dat",O_WRONLY|O_CREAT);

   if (fd<0)
      write(2,"Ich kann die Datei nicht öffnen!\n",33);
   else
    {
      if (write(fd,data,sizeof(data))<=0)
         write(2,"Schreibfehler!\n",15);
      close(fd);
    }
 }
```

3.2.1.6 Standard-Kanäle

Jeder Prozess ist von vornherein mit drei I/O-Kanälen verbunden:

- **Standard-Eingabe** (normalerweise die Tastatur des Terminals)
- **Standard-Ausgabe** (normalerweise der Bildschirm oder ein Konsolen-Fenster, etc.)
- **Standard-Fehlerkanal** (fällt meist mit der Standard-Ausgabe zusammen)

Das Öffnen und Initialisieren der drei Kanäle geschieht automatisch und braucht nicht vom Autor eines Programms explizit vorgenommen zu werden.

Diese Kanäle erhalten (in dieser Reihenfolge) immer die File-Deskriptoren 0, 1 und 2 (die erste weitere geöffnete Datei erhält die Nummer 3). Eigentlich sollte man aber in `C` besser die POSIX-Konstanten `STDIN_FILENO`, `STDOUT_FILENO` und `STDERR_FILENO` aus `unistd.h` verwenden.

Es macht durchaus Sinn, Ausgabe und Fehlerkanal voneinander zu trennen. Alle Kanäle können von Tastaur bzw. Bildschirm *umgelenkt* werden auf Dateien (dazu später). So kann man die eigentlichen Ausgabedaten in einer Datei ablegen, während die Fehlermeldungen weiterhin auf dem Bildschirm erscheinen.

Beispiel: Ohne die ggf. aufwendigen `stdio`-Funktionen zu bemühen, kann man also wie folgt ein Programm schreiben, das von der Standard-Eingabe auf die Standard-Ausgabe kopiert, dabei aber Groß- in Kleinbuchstaben umwandelt:

```
int main()
 {
   ssize_t b;
   char buffer[80];

   while (b=read(0,buffer,80), b>0)
    {
      char *P=buffer;
      ssize_t i=b;
      for (;i>0;--i) *P++=tolower(*P);
      write(1,buffer,b);
    }
 }
```

Die drei Kanäle haben aber High-Level-Entsprechungen als `C`-Dateien (über `FILE*` ansprechbar) und `C++`-Ströme (`istream`, `ostream`):

File-Deskriptor	C-Datei	C++-Strom
(int) 0	FILE * stdin	istream cin
(int) 1	FILE * stdout	ostream cout
(int) 2	FILE * stderr	ostream cerr

Beispiel: Wir geben drei Arten an, die drei Kanäle von `C`- bzw. `C++`-Programmen aus anzusprechen:

```
#include <unistd.h>

int main()
 {
  char Buffer[80];
  int len;
  len=read(0,Buffer,80);
  write(1,Buffer,len);
  write(2,"OK\n",3);
 }
```

```
#include <stdio.h>

int main()
 {
  char Buffer[80];
  int len;
  len=fread(stdin,Buffer,80);
  fwrite(stdout,Buffer,len);
  fprintf(stderr,"OK\n");
 }
```

```
#include <iostream.h>

int main()
 {
  char Buffer[80];
  cin.getline(Buffer,80);
  cout << Buffer << endl;
  cerr << "OK" << endl;
 }
```

3.2.1.7 /dev/fd

In SVR4 (nicht POSIX!) gibt es ein spezielles Verzeichnis `/dev/fd`, in dem das System jeden File-Deskriptor als Pseudo-Datei widerspiegelt, beispielsweise `/dev/fd/0` für die Standard-Eingabe:

```
cat /dev/fd/0
```

Dieses Kommando liefert jede eingegebene Zeile als Echo zurück. Dieser Mechanismus ist bei Kommandos günstig, die einen Filenamen als Argument erwarten, aber Standard-Ein- oder -Ausgabe verwenden sollen.

Unter L*i*nux ist `/dev/fd` ein symbolischer Link nach `/proc/`n`/fd`, wobei n die Nummer des aktuellen Prozesses ist (zu `/proc` später genauer). Schreibberechtigung vorausgesetzt, kann man so auch auf die Standard-Ausgabe eines anderen Prozesses schreiben, etwa:

```
cat > /proc/1231/fd/1
```

Die Einträge stehen für „character special files“ (dazu später) oder werden als symbolische Links verwaltet (erste Zeile SUN, zweite L*i*nux):

```
crw-rw-rw-   1 root    root   159,  0 Nov  3 16:55 /dev/fd/0
lrwx------   1 root    root        64 Nov  3 21:18 /dev/fd/0 -> [0308]:11353
```

Je nach System liefert eine Auflistung von `/dev/fd` nicht nur Einträge für die tatsächlich benutzten File-Deskriptoren, sondern `OPEN_MAX` Einträge (z.B. 0 bis 63 oder bis 255). Auf momentan nicht verwendete Nummern kann man aber nicht zugreifen.

3.2.1.8 Streams vs. File-Deskriptoren

Meist ist es komfortabler, mit den `stdio`-Funktionen zu arbeiten, insbesondere, wenn Formatierung mit ins Spiel kommt. Im Zusammenhang mit einigen Systemaufrufen benötigt man aber zwingend File-Deskriptoren.

Über folgende beiden Funktionen (aus `stdio.h`) kann man von Low-Level- auf High-Level-Funktionen übergehen und umgekehrt:

UNIX/DOS
`int fileno(FILE *stream);` liefert zum angegebenen `C`-Strom den zugrundeliegenden File-Deskriptor.
`FILE *fdopen(int fd, const char *type);` öffnet einen `C`-Strom, basierend auf dem File-Deskriptor `fd`. `type` entspricht dem letzten Argument bei `fopen` (z.B. `"r"` oder `"w"`) und sollte zum Deskriptor passen. Eine bereits vorgenommene Positionierung in der Datei wird übernommen.

Beispielsweise liefern die System-Routinen für Pipes und Sockets (dazu später) File-Deskriptoren, um die herum man mit `fdopen` einen Strom legen und dann mit `fprintf` und `fscanf` arbeiten kann. Der `C`-Startup-Code benutzt `fdopen`, um aus 0 den Strom `stdin` zu machen, etc. Es gibt kein Analogon für `C++`-Streamklassen.

Die Datei sollte am Schluss mit `fclose` geschlossen werden – ohne zusätzliches `close` auf den File-Deskriptor.

Während bei `write`-Kommandos i.Allg. ohne Pufferung direkt geschrieben wird, schreiben die `stdio`-Funktionen zeilenweise oder erst bei Pufferüberlauf. Um sicherzugehen, sollte man in kritischen Fällen mit `fflush(FILE*)` den Puffer leeren.

Bemerkung: Die Low-Level-Funktionen sind allerdings manchmal auch gepuffert. Um ganz sicherzugehen, kann man mit der Funktion `fsync` das Herausschreiben des Puffers auf Platte erzwingen:

UNIX
int fsync**(int** fd**);** schreibt geänderte Pufferinhalte, die sich auf die offene Datei `fd` beziehen, physisch auf das Medium.

3.2.1.9 Mapped Files

In UNIX gibt es eine einfache Möglichkeit, Dateien wie Hauptspeicher anzusprechen (mittlerweile standardisiert, POSIX.1b) – mit den Funktionen `mmap` und `munmap` (aus `sys/mman.h`, *memory management*). Es werden Mechanismen des virtuellen Speichers ausgenutzt, die wir genauer noch in einem eigenen Kapitel kennenlernen werden.

Die Datei wird in spezielle Speicherseiten in den Hauptspeicher eingeblendet. Sie wird dazu nicht vollständig eingelesen. Erst ein Zugriff auf die virtuellen Adressen löst ggf. eine Seiten-Operation aus, die den jeweils neu benötigten Teil von Platte lädt (bzw. auf sie schreibt).

UNIX
void ***mmap(void ***start**, **size_t** length**, int** prot**, int** flags**,** **int** fd**, **off_t** offset**);** blendet einen Teil der offenen Datei mit dem File-Deskriptor `fd` in den Hauptspeicher ein. `offset` ist der Beginn des Bereichs, gezählt in Bytes vom Dateianfang, `length` seine Länge. Die virtuelle Adresse des angelegten Speicherbereichs wird zurückgegeben (-1 im Fehlerfall).
int munmap**(void ***start**, **size_t** length**);** der mit `mmap` angelegte Bereich ab `start` der Länge `length` wird wieder freigegeben.

Der Parameter `start` bei `mmap` ist ein Vorschlag für die virtuelle Adresse des Bereichs, der nicht befolgt zu werden braucht. Meist wird hier einfach der Nullpointer angegeben. (Bei Gerätetreibern kann die Angabe Sinn machen.)

Der Parameter `prot` bei `mmap` gibt das `rwx`-Verhalten der Seite an (mit Konstanten `PROT_READ`, etc.). In `flags` ist angegeben, ob andere Prozesse, die dieselbe Datei mappen, denselben Speicherbereich sehen sollen (`MAP_SHARED`, bei `MAP_PRIVATE` bekommen sie einen eigenen Bereich).

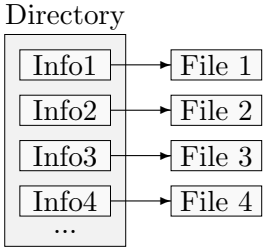
Wenn man viel in nicht zu großen Dateien hin- und herfahren möchte, lohnt sich der Einsatz von `mmap`. Man macht von den virtuellen Speichermechanismen als komfortabler Dateicache Gebrauch. Außerdem ist die Schnittstelle (Hauptspeicherzugriff über Pointer) meist bequemer als das Lesen/Schreiben per `read` und `write`.

Beispiel: Folgendes Programm liest einen Dateinamen ein und bildet die Datei in den Hauptspeicher ab. Danach kann man beliebig oft das Vorkommen eines bestimmten Zeichens in der Datei zählen lassen. Beim ersten Mal wird auf die Platte zugegriffen, danach (hoffentlich) nicht mehr.

```
int main()
 {
   char Buffer[256],*start;
   size_t length;
   int fd;

   printf("filename: ");
   fgets(Buffer,256,stdin);
   Buffer[strlen(Buffer)-1]=0;
   fd=open(Buffer,O_RDONLY);
   if (fd<0) { perror("oops"); exit(0); }
   length=lseek(fd,0,SEEK_END);
   lseek(fd,0,SEEK_SET);

   start=mmap(0,length,PROT_READ,MAP_PRIVATE,fd,0);
   if (start==(char *)-1) { perror("oops"); exit(0); }

   for (;;)
    {
      char letter,*Q=start,*Qend=start+length;
      unsigned long count=0;
      printf("count which letter (ESC to stop): ");
      letter=fgetc(stdin);
      if (letter==27) break;
      while (Q<Qend) if (*Q++==letter) ++count;
      printf("found ASCII(%d) %d times\n",letter,count);
      if (letter!='\n') while (fgetc(stdin)!='\n');
    }

   munmap(start,length);
   close(fd);
 }
```

3.2.2 Directories

Ein **Directory** (Ordner, Verzeichnis, Catalog) ist eine Sammlung von (logisch zusammengehörigen) Dateien – oder (je nach Standpunkt) eine Sammlung von allen notwendigen *Informationen* über diese Dateien.



Abstrakter kann man als Directory jedes System-Objekt ansehen, das Operationen wie *create_entry*, *delete_entry*, *search_entry*, *rename_entry*, *list_contents* (ggf. mehr) zur Verfügung stellt. (Die Einträge sollen sich natürlich auf Files beziehen, sonst haben wir einen mengenartigen abstrakten Datentyp definiert.)

Früher (z.B. unter CP/M und MS-DOS Version 1) gab es keine Strukturierung der Dateisammlung auf dem Medium – es gab also sozusagen nur genau ein Verzeichnis. Daraus ergaben sich einige Probleme:

- Namenskonflikte (besonders bei mehreren Benutzern)
- Abwesenheit jeglicher Datensicherheit (keine privaten Daten)
- maximale Unordnung

Auf einigen Großrechnersystemen gab es ein Directory pro Benutzer. So waren die Daten zwar voneinander abgeschottet, jeder Benutzer hatte aber immer noch einen unordentlichen Haufen Daten mittels geschickter Namensvergabe zu verwalten.

Daraus entwickelte sich erst später eine **Directory-Hierarchie** – d.h. Directories dürfen weitere Directories enthalten, rekursiv in beliebiger Tiefe.

Durch diese Rekursivität entsteht eine (**baumartige**) Struktur der Directories. Die Gesamtheit aller Dateien und Directories heißt **Dateibaum**. Das „oberste“ Directory (das ohne Vorgänger) in diesem Baum heißt **Wurzelverzeichnis**. Meistens wird es mit „`/`“ angesprochen, unter MS-DOS/Windows mit „`\`“.

Beispielsweise sollten Systemdaten in einem Unterbaum untergebracht sein (dort wiederum feiner klassifiziert), die Benutzerdaten in einem anderen Unterbaum. Dort sollte es wiederum für jeden Benutzer einen eigenen Unterbaum geben, etc.

- In MS-DOS/Windows gibt es einen Dateibaum pro Speichermedium (bzw. pro Partition bei partitionierten Festplatten). Es gibt entsprechend viele Wurzeln (genannt `A:\`, `C:\`, etc.).
- In UNIX gibt es immer exakt einen Dateibaum. Verschiedene Medien sind in den Gesamtbaum „eingehängt“. Es gibt immer genau eine Wurzel namens „`/`“.

Wie die Systeme diese Directory-Ordnung realisieren, wird im Kapitel über Dateisysteme besprochen werden.

- Ein **Directory-Eintrag** ist eine Sammlung von Daten über eine Datei, die sich im Directory befindet (bzw. über ein Unterverzeichnis). Das sind natürlich der Name, systemabhängig die Größe, das Datum des Anlegens/der letzten Änderung, Zugriffsrechte etc.
- Ein **Pfad** ist die Positionsangabe einer Datei bzw. eines Verzeichnisses innerhalb des Dateibaums. Pfade sind entweder
  - **relativ** zur Position eines Verzeichnisses
  - **absolut**, die Angabe, wie der Dateibaum von der Wurzel aus zu durchlaufen ist, um die Position aufzufinden.

  Absolute Angaben beginnen immer mit „`/`“ bzw. „`\`“, dem die zu durchlaufenden Directories folgen, durch „`/`“ bzw. „`\`“ getrennt, z.B. „`/bin/login`“ oder „`C:\WINDOWS\SYSTEM\MSMOUSE.VXD`“.

  Relative Angaben beginnen mit einem Datei- oder Verzeichnisnamen (insbesondere mit „`.`“ oder „`..`“).

- Das „**aktuelle Verzeichnis**“ ist eine Art „Anker“ für relative Pfadangaben – damit nicht immer mit langen absoluten Pfaden gearbeitet werden muss.

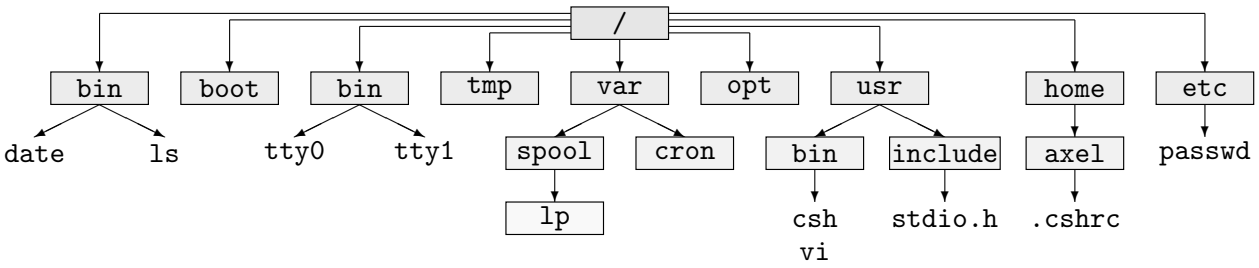
  In UNIX ist beim Betreten des Systems das aktuelle Verzeichnis üblicherweise das Wurzelverzeichnis des Unterbaums, der dem Benutzer gehört, sein „Heimatverzeichnis“, z.B. „`/home/axel`“. In MS-DOS ist es das Wurzelverzeichnis des Boot-Mediums, z.B. „`C:\`“. Danach kann es vom Benutzer bzw. vom Programm gewechselt, also im Dateibaum bewegt werden.

  In Kommandos, bei Dateizugriffen in Programmen etc. werden relative Pfadangaben immer auf das gerade aktuelle Verzeichnis bezogen. Das aktuelle Verzeichnis heißt immer „`.`“, das übergeordnete (Parent-Directory) „`..`“ – relative Pfadangaben dürfen hiermit beginnen. In den meisten Shells ist außerdem „`~`“ für das Heimatverzeichnis des Benutzers erlaubt. Beachte: Das Parent-Directory von „`/`“ ist wieder „`/`“.

Die wichtigsten Directory-bezogenen Kommandos sind:

	UNIX	**MS-DOS**
Anzeigen des Directory-Inhalts	`ls`	`dir`
Anlegen eines neuen Verzeichnisses	`mkdir`	`mkdir, md`
Löschen eines leeren Verzeichnisses	`rmdir`	`rmdir, rd`
Umbenennen eines Verzeichnisses	`mv`	`move`
Wechseln des aktuellen Verzeichnisses	`cd`	`cd`
Ausgabe des Pfads des aktuellen Verzeichnisses	`pwd`	`pwd`

Der Dateibaum unter UNIX ist nicht genormt; einige Unterbäume muss es aber immer geben. Typischerweise beginnt die Hierarchie wie folgt:

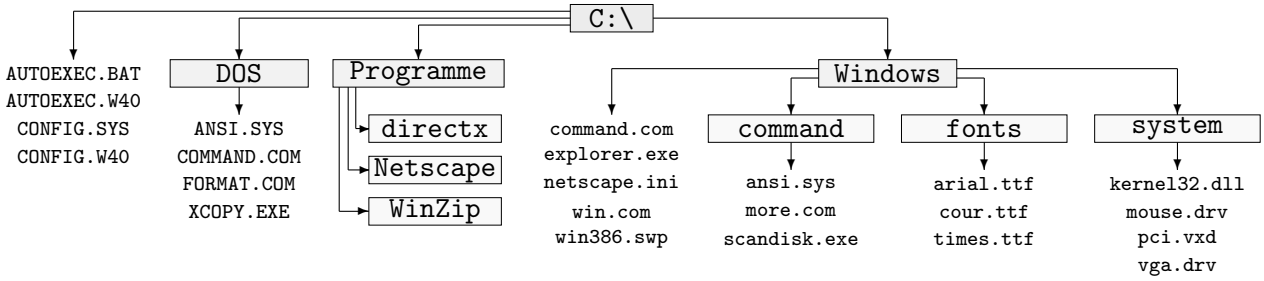


Dabei haben die Verzeichnisse grob folgende Bedeutung:

`/bin`	essentielle Systemkommandos
`/dev`	Gerätetreiber (Verweise auf die eigentlichen Treiber im Kernel)
`/etc`	diverse Dateien/Programme zur Systemverwaltung
`/tmp`	temporäre Dateien
`/lib`	essentielle System-Bibliotheken
`/boot`	beim Booten verwendete Dateien
`/home`	Home-Verzeichnisse der Benutzer (manchmal `/usr/home`)
`/usr/bin`	zusätzliche Systemkommandos
`/usr/lib`	zusätzliche Bibliotheken und Konfigurationsdateien, etc.
`/usr/man`	Manual-Seiten
`/usr/include`	Include-Dateien von Compilern
`/usr/local`	Unterbaum mit lokalen Ergänzungen analog zu `/usr`

Diese Struktur ist historisch gewachsen (seit AT&T, BSD weicht hiervon eher ab) und manchmal ein wenig unpraktisch. Sie neigt dazu, Dateien nach ihrem *Typ* und nicht nach *logischen* Zusammenhängen zu sortieren. Die meisten ausführbaren Programme liegen in `/bin` und `/usr/bin`, ihre Initialisierungsdateien in `/etc` oder `/var` oder `/usr/lib`, etc. Das führt zu unübersichtlichen Directories und gelegentlichen Suchaktionen.

Es folgt ein kleiner Ausschnitt des Dateibaums auf der Bootpartition eines Windows95-Systems (mit einer alten DOS-Version):



Auch das Windows-System ist natürlich gewachsen, und der Dateibaum ist nicht besonders gut strukturiert. Im Verzeichnis `Windows` finden sich Kommandos, fensterorientierte Systemprogramme, Konfigurationsdateien, Bildschirmschoner (!) etc. Im Unterverzeichnis `system` sind die meisten Treiber und Bibliotheken zu finden – leider auch viele, die nur von einem einzigen Programm gebraucht werden und in diesem allgemeinen Verzeichnis eher fehl am Platz sind.

Systemaufrufe im Zusammenhang mit Directories sind folgende:

UNIX/(DOS)
`char *getcwd(char *buf, size_t size);` legt den absoluten **Pfad** des **aktuellen Verzeichnisses** in `buf` ab, maximal `size-1` Zeichen. Reicht der Puffer nicht, wird 0 zurückgegeben, sonst `buf`. Unter UNIX beginnt der Pfadname immer mit '`/`', unter DOS mit Laufwerksbuchstaben und „`:\`“.
`int chdir(const char *path);` **wechselt** das aktuelle Verzeichnis in das, das der Pfad `path` bezeichnet.
`int fchdir(int fd);` *`/* kein POSIX, nicht unter DOS! */`* **wechselt** das aktuelle Verzeichnis in das, das der File-Deskriptor `fd` bezeichnet.
`int mkdir(const char *path, mode_t mode);` *`/* nicht unter Windows! */`* **legt** ein neues Verzeichnis **an** (im aktuellen Verzeichnis, falls `path` nur ein Name ist). `mode` gibt (wie bei Dateien und `creat`) die Zugriffsrechte an.
`int rmdir(const char *path);` *`/* nicht unter Windows! */`* **löscht** das angegebene Verzeichnis.

Ein Rückgabewert von 0 signalisiert jeweils Erfolg, -1 Misserfolg (`errno` enthält den genauen Fehlercode).

Diese Funktionen sind unter UNIX Systemaufrufe. Aus Kompatibilitätsgründen liefern für DOS bzw. Windows die meisten Compiler-Hersteller aber Bibliotheksfunktionen derselben Funktionalität mit.

Gelesen und *beschrieben* wird ein Directory normalerweise durch Anlegen und Ansprechen von *Dateien* darin (siehe den vorigen Abschnitt). Jedes Directory lässt sich auch als normale Datei mit `open` öffnen und lesen – das Format ist aber systemabhängig. Das Schreiben ist normaler- und sinnvollerweise nur dem Kern erlaubt.

3.2.2.1 Lesen von Directories

Der interne Aufbau von Directories ist abhängig vom verwendeten Dateisystem. Es gibt aber C-Bibliotheksfunktionen, die in POSIX eingeführt wurden und sich dann über alle UNIX-Varianten und sogar MS-DOS verbreitet haben.

Für einen Directory-Eintrag wird eine Struktur **`struct dirent`** (aus `dirent.h`) verwendet. Sie enthält für den Dateinamen eine Komponente **`char d_name[]`**. Weitere Informationen über die Datei kann man sich dann über den Namen beschaffen (mit der Funktion `stat` o.ä., mehr dazu im Kapitel über Dateisysteme).

Directories werden zunächst mit `opendir` für den Auslesevorgang geöffnet. Es wird ein „Handle“ vom Typ `DIR*` (analog zum File-Handle `FILE*`) zurückgegeben. In der nur intern verwendeten Struktur `DIR` merken sich die Funktionen die momentane Ausleseposition, und hier ist eine *statische* `dirent`-Struktur untergebracht.

Danach kann man mit `readdir` jeweils einen Eintrag lesen. Die statische Struktur wird jeweils überschrieben, sodass man die relevanten Informationen (wie den Dateinamen) direkt nach dem `readdir` in eigene Variablen kopieren sollte.

UNIX/DOS
`DIR *opendir(const char *pathname);` Öffnen des durch `pathname` angegebenen Directories für die folgenden Lese-Operationen; liefert ein „Handle“ zurück (oder 0 bei Fehler)
`struct dirent *readdir(DIR *dirp);` Lesen des „nächsten“ Directory-Eintrags über das Handle `dirp`; es wird ein Zeiger auf eine statische Struktur `dirent` zurückgeliefert, bzw. 0, wenn es keine weiteren Einträge mehr gibt
`void rewinddir(DIR *dirp);` Zurückspulen, ermöglicht das erneute Lesen des Directories von Beginn an
`int closedir(DIR *dirp);` Schließen des Directories

3.2.3 Zugriffsrechte

Auf Systemen mit mehreren Benutzern ist es unumgänglich, den Zugriff auf Dateien auf deren „Besitzer“ oder andere autorisierte Personen zu beschränken.

Zugriffsrechte sind Festlegungen darüber, welcher Benutzer welche grundlegenden Dinge mit einer Datei tun kann: Lesen, Schreiben, Ausführen, ggf. Löschen oder einige mehr.

Unter MS-DOS und Windows (außer NT) gibt es keine vollwertigen Zugriffsrechte. Man kann allerdings eine Datei *vor jedermann* schreib- bzw. leseschützen. Die weiteren Betrachtungen beziehen sich auf UNIX.

3.2.3.1 Lese-, Schreib- und Ausführungs-Flags

- Der Benutzer, der eine Datei anlegt, wird zu deren Besitzer und hat als einziger (außer dem Super-User) das Recht, die Zugriffsrechte auf die Datei zu ändern. Es gibt drei Berechtigungen: Lesen (`r`), Schreiben (`w`) und Ausführen (`x`), zusammen die „`rwx`-Flags“. Gelöscht werden kann eine Datei vom Besitzer des umliegenden Directories.

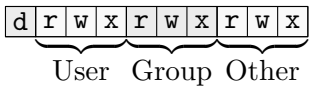
- Die Benutzerverwaltung unter UNIX wird später besprochen. Es gibt jedenfalls aus der Systemsicht drei Arten von Beziehungen von Benutzern zu einer Datei:

Besitzer (User, `u`)	derjenige, der die Datei angelegt hat (kann noch nachträglich geändert werden, `chown`)
Gruppe (Group, `g`)	Sammlung von Benutzern, z.B. eine Arbeitsgruppe, deren Mitglieder teilweise Zugriff auf die Daten anderen Mitglieder haben sollen
Andere (Other, `o`)	der Rest der Benutzer auf dem System

  Für jede Datei/jedes Verzeichnis werden im Dateisystem für diese drei Kategorien die `rwx`-Flags gespeichert. Mit „`ls -l`“ erhält man vor dem Dateinamen diese Angaben:

  ```
  -rwxr-xr-x   1 axel     am          4204 Nov  2 13:45 opsys.tex
  ```

  Die Verteilung ist wie folgt:

- Mit dem Befehl `chmod` kann der Besitzer die Zugriffsrechte ändern. Im Parameter sind folgende Zeichen erlaubt:
  - ‘`ugoa`’ für User, Group, Other und All,
  - ‘`+/-`’ für Erlauben/Nicht-Erlauben, `=` für die Angabe *aller* erlaubten Rechte,
  - ‘`rwx`’ für Read/Write/Execute (evtl. mehr, siehe Filesysteme),
  - ‘`,`’ zum Trennen mehrerer Einzelangaben.

  Alternativ können die 9 Flags als dreistellige Oktalzahl angegeben werden (mit der niederwertigsten Stelle rechts, in der obigen Darstellung). Jede Stelle steht für eine Benutzerklasse, wobei sich für jede Folgendes ergibt:

0	`---`	3	`-wx`	6	`rw-`
1	`--x`	4	`r--`	7	`rwx`
2	`-w-`	5	`r-x`		

  Die Berechtigungen aus der `ls`-Ausgabe von oben erhält man also wie folgt (zwei gleichwertige Alternativen):

  ```
  chmod go+rx-w,u+rwx opsys.tex
  chmod 755 opsys.tex
  ```

- In den Systemaufrufen `creat`, `open` (Seite 63) und `mkdir` (Seite 71) kann der Zugriffsmodus direkt als letzter Parameter `mode` angegeben werden.

  Wenn eine Datei z.B. in der Shell, von einem Editor aus, etc. neu angelegt wird, erhält sie ihre Zugriffsrechte wie folgt:

Von einer systemweiten Konstante (z.B. `666` für `rw-rw-rw`) wird eine benutzerspezifische Konstante (z.B. `022` für `----w--w-`) *subtrahiert* (genauer: es wird die UND-Verknüpfung mit dem Komplement gebildet, hier bleibt `rw-r--r--` übrig).

Diese benutzerspezifische Konstante (*user file creation mode mask*) wird mit dem Kommando `umask` (nur „`umask`“) abgefragt oder gesetzt (z.B. „`umask 022`“). Auch Aufrufe wie „`umask a=rx,ug+w`“ im Stil von `chmod` sind möglich.

- Es gibt zwei Systemaufrufe, um von Programmen aus die Benutzerrechte zu ändern:

UNIX/(DOS)
int chmod**(const char** ***path, mode_t mode);** setzt die Zugriffsrechte per Pfadname (zur DOS-Version siehe 3.2.6)
int fchmod**(int fildes, mode_t mode);** */* nicht unter DOS! */* setzt die Zugriffsrechte per File-Deskriptor

Die Rechte müssen in `mode` immer als Binärmuster angegeben werden. In `sys/stat.h` gibt es symbolische Konstanten, die (zusammengeodert) dafür verwendet werden können, entweder nach dem Schema `S_Ixyyy`, wobei `x`∈{`R`,`W`,`X`} und `yyy`∈{`USR`,`GRP`,`OTH`}, oder nach dem Schema `S_IRWXy` mit `y`∈{`u`,`g`,`o`}.

Unter DOS/Windows ist `mode` vom Typ **`int`**, und es sind aufgrund des beschränkten Dateisystems nur die beiden Bits `S_IREAD` und `S_IWRITE` erlaubt.

- Zur Abfrage der Rechte gibt es folgenden Systemaufruf:

UNIX/(DOS)
int access**(const char** ***pathname,* int** *mode);* fragt ab, ob der Benutzer die in `mode` angegebenen Rechte für die Datei `pathname` besitzt

Das Recht, das abgefragt werden soll, wird als `mode` angegeben (Konstanten `R_OK`, `W_OK`, `X_OK` und `F_OK` für bloße Existenz). Mehrere Angaben können zusammengeodert werden. Wenn der aufrufende Benutzer *alle* abgefragten Rechte hat, liefert die Funktion 0, ansonsten -1.

Unter DOS/Windows sind die Konstanten nicht definiert – man muss die Werte numerisch angeben: `F_OK`=0, `X_OK`=1 (wird aber ignoriert!), `W_OK`=2, `R_OK`=4.

- Wenn eine Datei angelegt wird, wird im Filesystem der aktuelle Benutzer als Besitzer der Datei abgespeichert. Als Gruppenbesitzer wird entweder die Gruppe des Benutzers oder der Gruppenbesitzers des Directories eingetragen. (BSD macht immer letzteres, unter SVR4 kann man für jedes Directory gesondert festlegen, unter L*i*nux für jedes Dateisystem).

  Der Besitzer der Datei kann diese Einträge mit den Kommandos `chown` (change owner) oder `chgrp` (change group) nachträglich ändern – (auf einigen Systemen) beides kombiniert in der Form

  `chown` *user*`.`*group file(s)*

  Es gibt kombinierte Systemaufrufe für beides:

UNIX
int chown**(const char** ***path, uid_t owner, gid_t group);** setzt Besitzer und Gruppe per Pfadname
int fchown**(int**fd**, uid_t owner, gid_t group);** setzt Besitzer und Gruppe per File-Deskriptor

3.2.3.2 SUID, SGID und SVTX

Es gibt in UNIX eigentlich nicht nur 9, sondern 12 Flags für Zugriffsrechte.

- Wenn ein Kommando ausgeführt wird, erbt der Prozess normalerweise die Rechte des Programms (z.B. der Shell), von dem aus es aufgerufen wurde. Ein Normalbenutzer kann deshalb niemals administrative Kommandos wie `shutdown` absetzen, die einen Versuch mit der Meldung „`must be root`“ quittieren.

  Es gibt ein spezielles Flag namens **SUID**-Bit (*set user id*). Wenn es für die Datei eines ausführbaren Programms gesetzt ist, läuft der Prozess so, als wäre er vom *Besitzer* der Datei gestartet worden. Wenn der Besitzer also `root` ist, kann jeder Benutzer dieses Kommando mit Superuser-Privilegien ausführen.

  Das Analogon für die Gruppenzugehörigkeit ist das **SGID**-Flag (*set group id*). Wenn es gesetzt ist, wird für den Prozess als Gruppenzugehörigkeit der Gruppenbesitzer der Datei verwendet.

  Beispielsweise erfolgt das Ausdrucken auf dem Standard-Drucker durch das Kopieren von Daten auf den Gerätetreiber `/dev/lp` (dazu später). Der Treiber darf nicht von jedermann beschreibbar sein, da sonst Daten aller Benutzer unkontrolliert gemischt gedruckt würden. Üblicherweise hat nur der Superuser oder eine Gruppe „`lp`“ Schreibrecht.

  Stattdessen benutzt man ein Kommando wie `lpr`, das der Gruppe „`lp`“ gehört, und dessen SGID-Bit gesetzt ist. Normalbenutzer können das Kommando aufrufen, und es besitzt dennoch die notwendigen Schreibrechte für den Drucker-Treiber. (In der Praxis wird noch der Umweg über ein Spooling-Directory und einen Drucker-Dämon gegangen.)

  Vorsicht: Programme, die `root` gehören, und bei denen das SUID-Bit gesetzt ist, machen *jeden Benutzer*, der das Programm aufruft, zum *Super-User* mit allen Rechten. Bei einzelnen Kommandos mit eingeschränkter Funktion ist das (kontrolliert) unproblematisch. Schwierigkeiten gibt es immer, wenn von dem SUID-Kommando aus weitere Systembefehle absetzbar sind. Man sollte als Administrator also nicht unbedingt eine SUID-`root`-Version der Shell herumliegen lassen!

  Vorsicht 2: Man sollte vorsichtig sein, wenn man einem Programm die SUID-`root`-Berechtigung gibt oder ein solches schreibt. Viele Programme geben dem Benutzer die Möglichkeit, in einer Zeile ein Kommando abzusetzen. Zu dessen Ausführung wird dann intern eine Shell gestartet, die die `root`-Rechte erbt. Solche Programme sollten also explizit (z.B. in einem Kindprozess) vor der Kommando-Ausführung die Spezialrechte aufgeben: `setgid(getgid())` gleicht die effektive der tatsächlichen ID an.

  Vorsicht 3: Wenn man sein Terminal kurz unbeaufsichtigt lässt, könnte eine vorbeikommende andere Person eine `SUID`-Version der Shell mit den ihr fremden Rechten (z.B. in

ihrem eigenen Verzeichnis) erzeugen. Diese Person kann dann jederzeit die fremde Identität vollständig annehmen!

- Es gibt ein weiteres (zwölftes) Bit, das sogenannte „Sticky-Bit“ (oder *save text image*, **SVTX**). Wenn es für die Datei eines ausführbaren Programms gesetzt ist, ist das ein Hinweis an das System, das Programm nach seiner Ausführung nach Möglichkeit nicht direkt aus dem Hauptspeicher zu entfernen, weil zu vermuten ist, dass es nach kurzer Zeit erneut aufgerufen werden wird. Es braucht dann nicht erneut von Platte geladen zu werden.

- Die Bits haben die Kurznamen ‘`s`’ (SUID für den Besitzer, SGID für die Gruppe) bzw. ‘`t`’ (SVTX). Sie erscheinen so in Directory-Auflistungen mit „`ls -l`“ und können in `chmod`-Kommandos verwendet werden.

  ```
  -rwsr-sr-t   1 root     root        41231 Nov  2 23:23 cmd
  ```

  Die Oktal-Entsprechungen sind `S_ISUID=4000`, `S_ISGID=2000` bzw. `S_ISVTX=1000`. Die Bits können z.B. mit folgenden Kommandos gesetzt werden:

  ```
  chmod ug+s command     # setzt zusätzlich SUID und SGID
  chmod o-t  command     # löscht SVTX
  chmod 4711 command     # setzt genau SUID und rwx--x--x
  chmod 6755 command     # setzt genau SUID und SGID und rwxr-xr-x
  chmod 7777 command     # setzt genau SUID, SGID, SVTX und rwxrwxrwx
  ```

Die Funktionen `getuid` (Seite 55) und `getgid` (Seite 58) liefern die „echte“ User-/Gruppen-ID, d.h. die des Benutzers, der den aktuellen Prozess gestartet hat. Es gibt zwei weitere Funktionen `geteuid` und `getegid`, die die „effektive“ User-/Gruppen-ID liefern. Diese weicht eventuell im Fall des gesetzten SUID- bzw. SGID-Bits von der echten ab und entspricht der ID des Besitzers bzw. Gruppenbesitzers der gerade ausgeführten Datei.

Beispiel: Wir geben alle IDs in Kurzform aus:

```
int main()
 {
   printf("UID:%d GID:%d EUID:%d EGID:%d\n",
      getuid(),getgid(),geteuid(),getegid());
 }
```

Wir setzen mit „`chown axel a.out`“ und „`chgrp users a.out`“ explizit die Besitzer der Datei:

```
-rwxr-xr-x   1 axel     users        4329 Nov 18 00:33 a.out
```

Vom Benutzer `root` ausgeführt, gibt das Programm Folgendes aus:

```
UID:0 GID:0 EUID:0 EGID:0
```

Wir setzen nun mit „`chmod u+s,g+s a.out`“ das SUID- und SGID-Bit:

```
-rwsr-sr-x   1 axel     users        4329 Nov 18 00:37 a.out
```

Immer noch von `root` ausgeführt, erzeugt das Programm jetzt (z.B.) Folgendes:

```
UID:0 GID:0 EUID:500 EGID:100
```

Bemerkungen:

- Einige UNIX-Varianten interpretieren das SGID-Bit anders bei Dateien, die keinen ausführbaren Code enthalten („*mandatory record locking*“, Bedeutung später). Bei einer Auflistung wird dann ein großes ‘`S`’ dargestellt.
- Die meisten Systeme löschen die SUID- und SGID-Bits, wenn die Datei (mit `chown`, `chgrp`) den Besitzer wechselt.

3.2.4 Links

Der Graph des Dateisystems unter UNIX und Windows 95/NT ist eventuell kein Baum, sondern ein **azyklischer Graph** (ein Graph ohne gerichtete Zyklen). Es können „*Verweise*“ auf Dateien oder Directories angebracht werden (**Links**, **Verknüpfungen**), sodass sich die physischen Objekte scheinbar in mehreren Directories befinden.

Es wird *keine Kopie* der Datei angelegt. Änderungen über den einen Directory-Eintrag werden auch direkt über den anderen Eintrag widergespiegelt.

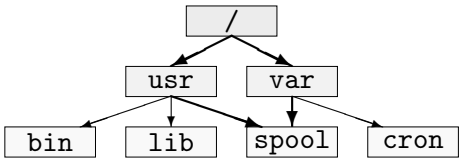
Die Verbindung kann auf zwei Weisen geschehen:

physisch:	die *physische Position* der Datei auf dem Medium wird eingetragen
symbolisch:	der *Pfadname* der Datei wird als Verweis eingetragen

Ein physischer Link heißt auch „**Hardlink**“. Für ihn müssen sich aus offensichtlichen Gründen Quelle und Ziel auf demselben Medium (derselben Platte/Partition etc.) befinden. Symbolische Links haben diese Einschränkung nicht – sie sind nur Dateien eines speziellen Typs, die zeichenweise den eigentlichen Pfadnamen enthalten. Windows kennt nur symbolische Links und erlaubt keine Links auf Directories.

Eine solche Mehrfach-Eintragung macht als „Alias-Name“ einen Sinn:

- Beispielsweise könnte eine Bibliothek in mehreren Versionen physisch vorhanden sein. Ein Link (mit Namen ohne eine Versions-Nummer) „zeigt“ dann auf die jeweils aktuellste oder anderweitig günstigste Version.
- Ein Programm kann über mehrere Namen erreichbar sein. Beispielsweise ist `gunzip` oft nur ein Link auf `gzip`. Das Programm ist so intelligent, an den angegebenen Dateien selbst zu erkennen, ob es komprimieren oder dekomprimieren soll.
- Wenn ganze Directories gelinkt werden, kann man einen speziellen Dateibaum an die Eigenheiten unterschiedlicher Programme anpassen. So könnte es Programme geben, die temporäre Daten in `/temp` speichern möchten. Anstatt zusätzlich zu `/tmp` ein solches Verzeichnis anzulegen und Daten zu verstreuen, setzt man einfach einen Link von `/temp` nach `/tmp`.
- So kann man auch Directories, die einen speziellen, festen Platz im UNIX-Dateibaum haben, physisch an anderen Stellen unterbringen, z.B. auf geräumigeren Platten-Partitionen (z.B. Link von `/usr/spool` nach `/var/spool` o.ä.).



Der hier entstandene Graph ist kein Baum mehr, hat aber keine *gerichteten* Zyklen. Solche Zyklen kann man verbieten, muss dann aber vor jedem Linken überprüfen, ob ein Zyklus entstehen würde. Eventuell sind auch solche Zyklen erlaubt – dann müssen Vorkehrungen getroffen werden, dass sich Programme nicht endlos verlaufen.

Hardlinks: In UNIX legt man einen Hardlink mit dem Befehl „`ln` *von nach*“ an – dabei ist *von* der Pfad der *existierenden* Datei, *nach* der Pfad des *neuen* Eintrags. So ein Link ist danach von einem „normal entstandenen“ Directory-Eintrag nicht zu unterscheiden ist. Nach den Befehlen

```
cat > text     # (Beenden mit CTRL-D = End-Of-File)
ln text text2
```

erhält man mit „`ls -l`“ z.B. folgende Ausgabe:

```
-rw-r--r--  2 root    root       22 Nov 10 22:47 text
-rw-r--r--  2 root    root       22 Nov 10 22:47 text2
```

Wenn in UNIX eine Datei gelöscht werden soll, bleibt sie so lange physisch erhalten, bis es keinen Eintrag mehr gibt, der sich auf sie bezieht. Dazu wird pro Datei eine Liste mit den Bezügen auf sie geführt.

Wenn wir oben also `text` löschen („`rm text`“), bleibt die Datei erhalten, ist aber nur noch unter dem Namen `text2` ansprechbar. Nach „`rm test2`“ wird sie physisch gelöscht.

Die Maximalzahl von Links auf eine Datei wird durch die POSIX-Konstante `LINK_MAX` (aus `limits.h`) festgelegt, z.B. 32767.

Symbolische Links: Mit `ln` und der Option „`-s`“ erzeugt man einen symbolischen Link. Er erzeugt bei bei „`ls -l`“ z.B. eine Ausgabe wie folgt:

```
lrwxrwxrwx  1 root   root   14 Nov 14 23:07 libcurses.a -> libcurses.so.1
```

Wenn man die Bezugsdatei löscht, erzeugen Zugriffe über den Link Reaktionen, als wäre der Eintrag nicht im Directory vorhanden. (In Windows führt danach ein Zugriff über den Link zu einem „Suchvorgang“.)

Zyklen im Dateibaum können natürlich nur durch Links auf Directories entstehen. Hardlinks auf Directories sind nur dem Superuser erlaubt, dem man zutraut, dadurch keine gefährlichen Situationen zu erzeugen. Symbolische Links auf Directories kann jeder User erzeugen. Systemfunktionen (z.B. zum rekursiven Durchlaufen eines Teilbaums) erkennen (hoffentlich) entstandene Schleifen. Sie brechen dann mit `errno=ELOOP` ab.

Systemaufrufe für Links sind folgende:

UNIX
`int link(const char *oldpath, const char *newpath);` legt einen **Hardlink** an
`int symlink(const char *oldpath, const char *newpath);` legt einen **symbolischen Link** an

UNIX
int unlink**(const char** ***pathname);** **löscht** einen Directory-Eintrag – wenn dadurch der letzte Link auf eine Datei verlorengeht, wird diese physisch gelöscht!
int readlink**(const char** ***path, void** ***buf,** size_t bufsiz**);** **liest** das Ziel des symbolischen Links `path` in den Puffer `buf` der Größe `bufsiz` ein. Ketten von Links werden verfolgt. `buf` wird *nicht* mit einem Nullbyte abgeschlossen, stattdessen wird die Länge des Namens zurückgegeben.

3.2.5 stat

Die UNIX-Systemaufrufe der `stat`-Familie dienen zum Einholen von Informationen über Dateien. Sie und die verwendete Struktur **struct** `stat` sind in `sys/stat.h` definiert.

UNIX/(DOS)
int stat**(const char** ***path, struct** stat ***buf);** füllt den Puffer `buf` mit Informationen über die Datei mit dem Pfad `path`. Wenn es sich um einen symbolischen Link handelt, erhält man die Information über die vom Link angegebene Datei.
int lstat**(const char** ***path, struct** stat ***buf);** */*nicht unter DOS!*/* wie `stat`; wenn `path` allerdings einen symbolischen Link bezeichnet, erhält man Informationen über den Link selbst.
int fstat**(int** fildes**, struct** stat ***buf);** wie `stat`, die Datei muss aber geöffnet sein und wird durch ihren File-Deskriptor angegeben.

Die Struktur **struct** `stat` ist systemabhängig. Einige wichtige, von POSIX vorgeschriebene Felder sind folgende:

`mode_t st_mode;`	Modus der Datei, enthält in den untersten 12 Bit die erweiterten Zugriffsrechte, außerdem den Dateityp mit folgenden zulässigen Bitmasken: `S_IFREG` regular file `S_IFDIR` directory `S_IFLNK` symbolischer Link (nur bei `lstat`) `S_IFIFO` fifo, named pipe `S_IFCHR` character special file `S_IFBLK` block special file
`nlink_t st_nlink;`	Zähler für Hardlinks
`uid_t st_uid;`	UID des Besitzers
`gid_t st_gid;`	GID des Gruppenbesitzers
`off_t st_size;`	Größe in Bytes (nicht für special files, bei symbolischen Links die Länge des Pfads)

Weitere Einträge (Datumsvermerke, etc.) werden im Kapitel über Dateisysteme behandelt. Einige hängen mit der Organisation der Dateisysteme unter UNIX zusammen (I-Nodes, Geräte-Informationen).

Um `stat` auf eine Datei anwenden zu können, braucht man keine Lese-Erlaubnis für die Datei. Vielmehr benötigt man die Lese-Erlaubnis für alle Directories im Pfad `path`, die zu der Datei hinführen.

Beispiel: Folgendes Programmstück ermittelt die Länge (in Bytes) der Datei „`a.out`“:

```
#include <sys/stat.h>
#include <sys/types.h>
...
struct stat statbuf;
if (stat("a.out",&statbuf)) perror("stat");
else printf("Länge=%ld\n",statbuf.st_size);
```

3.2.6 DOS-spezifische Aufrufe

Es gibt noch einige Bibliotheksfunktionen, die bei DOS-Compilern zusätzlich zu den von UNIX übernommenen eingeführt wurden. Die Laufwerks-bezogenen Aufrufe machen unter UNIX beispielsweise gar keinen Sinn, da es dort ja keine unterschiedlichen Laufwerke, sondern einen einheitlichen Dateibaum gibt.

DOS
`int getcurdir(int drive, char *directory);` schreibt nach `directory` den absoluten Pfad des aktuellen Verzeichnisses auf Laufwerk Nummer `drive` (0=Standard, 1=`A:`, 2=`B:`, etc.)
`int getdisk(void);` liefert die Nummer des aktuellen Laufwerks (0 für `A:`, 1 für `B:`, etc.)
`int setdisk(int drive);` wechselt zum Laufwerk mit der angegebenen Nummer (0 für `A:`, 1 für `B:`, etc.)
`long filelength(int handle);` gibt zu einer offenen Datei die Länge in Bytes zurück (-1 im Fehlerfall)

`filelength` lässt sich unter UNIX leicht mit einem `fstat`-Aufruf realisieren:

```
long filelength(int fd)
 {
   struct stat fileinfo;
   if (fstat(fd,&fileinfo)) return -1;
   return fileinfo.st_size;
 }
```

Ursprünglich war die DOS-Dateiverwaltung CP/M-kompatibel. Für den Dateizugriff musste mit einer Datenstruktur FCB (*File Control Block*) und einem DOS-Interrupt gearbeitet werden.

Die entsprechenden Aufrufe stehen aus Kompatibilitätsgründen bis heute (auch unter Windows) zur Verfügung, sind aber im Vergleich zu den Funktionen mit Handles umständlich. Allerdings sind einige Geräte-Zugriffe *nur* mit erweiterten FCBs möglich. Wir wollen uns hier nicht ausführlicher mit FCBs beschäftigen.

3.3 Aufrufparameter und Environment

Ein Programm erhält auf zwei einfache Weisen schon beim Start Informationen – über Parameter aus der Kommandozeile und über Umgebungsvariablen. Die Mechanismen stammen zwar ursprünglich aus UNIX, sind mittlerweile aber in die ANSI-C-Definition eingegangen und funktionieren daher auch unter MS-DOS, auf dem AMIGA, etc.

3.3.1 Aufrufparameter

Beim Start eines Programms über eine Kommandozeile werden Angaben hinter dem Programmnamen als Parameter an das Programm interpretiert.

Die Aufrufzeile selbst ist ein String, der aber vom Startup-Code, der z.B. `C-` und `C++`-Programmen automatisch zugelinkt wird, aufbereitet wird. Die Zeile wird durch Folgen von *Spaces* oder *Tabulatoren* in *Worte* zerlegt. Wenn ein Wort Spaces enthalten soll, müssen Anführungszeichen gesetzt werden. Diese Worte werden der `main`-Funktion als Parameter übergeben:

```
int main(int argc, char *argv[]);
```

`argc` (argument count) enthält die Anzahl der Worte inklusive des Programmnamens. Das Array `argv` (argument vector) enthält die eigentlichen Worte. `argv[0]` ist immer der Programmname. Die Variablen dürfen natürlich anders benannt werden; diese Schreibweise hat sich aber eingebürgert und ist deshalb immer gut lesbar.

Beim Aufruf

```
myprog say "hello world!"
```

ist also `argc`=3, `argv[0]`=`"myprog"`, `argv[1]`=`"say"` und `argv[2]`=`"hello world!"`.

Die Parameter werden vom System immer wie reine Strings behandelt. Wenn sie z.B. numerisch interpretiert werden sollen, muss das explizit im Programm selbst geschehen (mit `atoi`, etc.)!

Der *Rückgabewert* des Hauptprogramms (immer vom Typ **`int`**) kann von *dem* Programm verwendet werden, der *dieses* Programm aufgerufen hat, unter UNIX beispielsweise die Shell (dazu später genauer).

Beispiel: Folgendes Programm gibt alle Bestandteile der Kommandozeile aus, ohne sie irgendwie zu interpretieren:

```
#include <iostream.h>
int main(int argc, char *argv[])
 {
   for (int i=1;i<argc;++i)
      cout << i << ": " << argv[i] << endl;
   return 0;
 }
```

Zur *Interpretation* von Parametern, etwa als Optionen (eingeleitet mit '`-`' bzw. '`/`'), kommen wir im Abschnitt über Shell-Kommandos.

Wenn man die Parameter ignorieren möchte, kann man folgenden alternativen Prototypen verwenden:

```
int main(void);     /*  in C++ kann das void entfallen   */
```

3.3.2 Environment

Ein Programm erhält Informationen über seine Umgebung, also über Benutzer, Pfade bestimmter wichtiger Verzeichnisse, Shells, Terminal und Bildschirm etc., in Form einer Liste von Paaren „Name/Wert“, die als „**Environment**“ bezeichnet wird.

Ein einzelner Eintrag wird auch als „**Umgebungsvariable**“ bezeichnet. Mit Shell-Kommandos oder von Programmen aus kann man die Werte der Variablen verändern und abfragen, sowie neue Variablen anlegen.

Die Variablen haben für den Betriebssystem-Kern keine Bedeutung. Sie entsteht erst durch die Interpretation durch die Anwendungen, z.B. Shell, Compiler, Manual-Browser, etc.

Der Mechanismus ist zwar systemunabhängig, Namen und Bedeutung der Variablen ist aber nicht in ANSI-C festgelegt. Die Umgebung unter UNIX sieht völlig anders aus als die unter MS-DOS.

Der UNIX-Befehl `env` ohne Parameter gibt alle Umgebungsvariablen aus. Seine Ausgabe könnte wie folgt beginnen:

```
HOSTNAME=vulcan
DISPLAY=:0.0
PATH=/sbin:/usr/sbin:/home/axel/bin: ...
```

Einige typische UNIX-Umgebungsvariablen sind folgende:

`HOME`	das Heimatverzeichnis des jeweiligen Benutzers
`PATH`	Liste von Pfaden, in denen nach ausführbaren Programmen (und Skripten) gesucht werden soll (mit ‘`:`’ getrennt)
`LOGNAME`	der Benutzername des jeweiligen Benutzers
`SHELL`	der Pfad der Standard-Shell des Benutzers
`PWD`	der (absolute) Pfad des aktuellen Verzeichnisses
`TERM`	Typ des aktuellen Terminals (z.B. `xterm` oder `vt100`)
`MANPATH`	Liste von Pfaden, die Manual-Seiten enthalten
`DISPLAY`	ID des X-Window-Displays, auf dem standardmäßig Fenster geöffnet werden sollen
`PRINTER`	der Standard-Drucker, z.B. `lp`

Zu MS-DOS-Umgebungsvariablen siehe den Abschnitt zu `COMMAND.COM` am Ende des Kapitels.

Der Startup-Code übernimmt die Aufgabe, die interne System-Liste in ein Format umzusetzen, das von `C` aus günstig zu verwenden ist:

`C`-Programme erhalten eigentlich immer **drei** Parameter, von denen der letzte aber meist ignoriert wird:

```
int main(int argc, char *argv[], char *envp[]);
```

Es handelt sich bei ihm um ein weiteres String-Array – eines, das stringweise die Umgebungsvariablen enthält. Der letzte **`char*`** ist der Null-Pointer, um das Ende des Arrays zu markieren.

Außerdem gibt es eine globale Variable `environ` (deklariert in `stdlib.h`), die ebenfalls auf dieses Array zeigt:

```
extern char **environ;
```

Wenn man außerhalb von `main` das Array benötigt, wird man deshalb meistens `environ` verwenden.

Beispiel: Folgendes C++-Programm gibt das Environment *zweimal* aus – auf die beiden angedeuteten Weisen:

```
#include <iostream.h>
#include <stdlib.h>

int main(int argc, char *argv[], char *envp[])
 {
   for ( int i=0 ; envp[i]!=0 ; ++i ) cout << envp[i] << endl;

   char **e=environ;
   while (*e!=0) cout << *e++ << endl;

   return 0;
 }
```

Man beachte, dass es kein Analogon zu `argc` gibt, sondern dass das Array-Ende weiterhin durch den Null-Pointer gekennzeichnet wird!

Folgende vier Funktionen (aus `stdlib.h`) können in Programmen verwendet werden, um Umgebungsvariablen zu lesen, setzen, anzulegen und zu löschen:

UNIX/(DOS)
`char *getenv(const char *name);` **liest** den Wert der Umgebungsvariable `name`. Man darf den String, auf den ein Pointer zurückgegeben wird, nicht modifizieren, sondern muss erst eine Kopie anlegen!
`int putenv(const char *string);` **nimmt** `string` in das Environment **auf**. Der String muss das Format „*name=wert*“ haben. Eine existierende Variable *name* wird überschrieben. Es wird 0 (okay) oder -1 (Fehler) zurückgegeben.
`int setenv(const char *name, const char *value, int overwrite);` `/*nicht unter DOS/SVR4! */` **setzt** die Variable `name` auf den Wert `value`, d.h. ein Eintrag „`name=value`“ wird angelegt. Eine existierende Variable `name` wird nur überschrieben, falls `overwrite`$\neq$0.
`void unsetenv(const char *name);` `/*nicht unter DOS/SVR4! */` **entfernt** die Variable `name` aus dem Environment.

Beispiel: So kann man in UNIX die Identität des Benutzers ausgeben:

```
cout << "you are "<< getenv("USER") << '@'<< getenv("HOSTNAME") << endl;
```

Die Ausgabe ist z.B.: „`you are axel@vulcan`“.

4 Der Prozess-Begriff

4.1 Prozess-Konzept

Der Begriff des Prozesses ist das zentrale Konzept bei der Entwicklung und Betrachtung von Betriebssystemen. Manchmal wird aus historischen Gründen auch noch der Name „Job“ (synonym) verwendet. Die Bezeichnung „Prozess“ wurde sehr wahrscheinlich zuerst bei der Entwicklung von MULTICS (also zu Beginn der 60er Jahre) eingeführt.

Die Entwicklung des Konzepts hat ihren Ursprung im Wunsch nach **Pseudoparallelität**. Das Ziel war es ja, die CPU voll auszunutzen und schließlich mehrere Benutzer gleichzeitig zuzulassen. Abgesehen davon ist es oft rein konzeptionell sinnvoll, umfangreiche Aktivitäten in parallel ausführbare Teile zu zerlegen.

Es musste zur Verwaltung solcher gleichzeitiger Aktivitäten ein Werkzeug geschaffen werden. Es musste ein Konzept entwickelt werden, das festlegt, was genau unter einer solchen „Aktivität“ zu verstehen ist. Der Begriff „Prozess“ steht für die Grundeinheit einer „Aktivität“ im System.

Genauer ist ein Prozess der *Status* einer *Instanz* eines *Programms* – das soll heißen:

- ein Programm, das sich in der Ausführung befindet (obwohl es dazu nicht unbedingt gerade die CPU-Kontrolle besitzen muss)
- zusammen mit allen Informationen darüber, in welchem Zustand das ablaufende Programm gerade die Systemkomponenten sieht, also:
  - den Programmzähler (CPU-Register, Adresse des nächsten Maschinenbefehls)
  - andere Prozessor-Register (arithmetische, Index-Register, etc.)
  - Inhalte des von ihm benutzten Speichers und zugehörige Verwaltungsdaten (Stack und Stack-Pointer, Datensegment, Heap und Heap-Grenzen, etc.)
  - andere benutzte Ressourcen wie Dateien und I/O-Geräte (als Systemstrukturen)

Man darf auf keinen Fall **Prozess** mit **Programm** verwechseln:

Programm ist etwas **statisches**, rein abstraktes,

- eine **Arbeitsvorschrift**, ein Algorithmus (z.B. formuliert in Maschinensprache),
- anwendbar potentiell auf beliebige Eingabedaten (die nicht zum Programm gehören).

Prozess ist etwas **dynamisches**,

- eine **Aktivität** (gesteuert von einer Arbeitsvorschrift),
- eine sequentielle Folge von Einzel-Aktivitäten,
- besitzt einen „**Zustand**“ (als Funktion der Zeit), nämlich den **Kontext** des Programms (der Ausführungszustand, Register-Werte, Speicher-Werte, etc.),
- ist festgelegt auf die aktuellen Eingabedaten, an die er gekoppelt ist.

PROZESS

CODE (eig. Maschinencode)

```
int func(int val)
 {
    return 2*val;
 }

int main(int argc, char *argv[])
 {
    printf("%d\n",func(2));
 }
```

REGISTER

```
PC:    0x231abf47
Reg0:  1451
Reg1:  -2310
...    ...
```

DATENSEGMENT

```
argc:    3
argv[0]: "a.out"
argv[1]: "hallo"
...      ...
```

STACK

ret. to system
2
ret. to `main`

RESSOURCEN

```
stdin
stdout
stderr
```

Die genaue Bezeichnung war eigentlich „**sequential process**“, d.h. das System führt die Befehle des Prozesses sequentiell hintereinander aus, zu jedem Zeitpunkt höchstens einen Befehl. Es gibt also keine Gleichläufigkeit (Parallelität) innerhalb eines Prozesses. Bei Systemen mit „Threads“ (s.u.) ist diese Bezeichnung dann nicht mehr zutreffend.

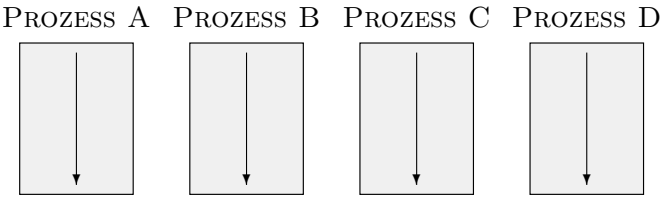
Es können durchaus zwei Prozesse gleichzeitig dasselbe Programm ausführen, mit anderen oder mit denselben Eingabedaten (z.B. fünfmal der `vi` oder `emacs`). Sie werden trotzdem als zwei voneinander verschiedene Prozesse betrachtet.

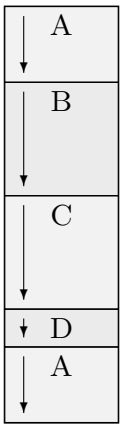
Eventuell steht dann auch der Code mehrfach im Speicher, eventuell gibt es System-Mechanismen zum Teilen von Code zwischen Prozessen. (Der Code muss dazu „reentrant“ sein, sich also nach dem Laden bei der ersten und jeden weiteren Ausführung gleich verhalten).

Die Aufgaben des Betriebssystems im Zusammenhang mit Prozessen sind folgende:

- Erzeugen eines neuen Prozesses
- Beenden und Entfernen eines alten Prozesses
- Synchronisation von Prozessen, die voneinander abhängen
- Kommunikation zwischen Prozessen

Virtuelle Sicht (Sicht des Prozesses):
Ein Prozess merkt normalerweise nicht, dass er nicht das einzige Programm im System ist. Er arbeitet mit einer „virtuellen CPU“, über die er die vollständige Kontrolle hat.

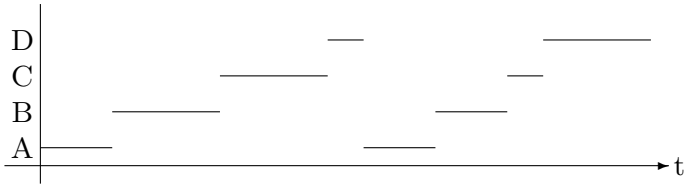




Physische Sicht (Sicht des Prozessors):
Physisch wird (bei Multiprogramming) aber natürlich die CPU in kurzen Zeitabständen zwischen den Prozessen hin- und hergeschaltet (Bild links).

Der zeitliche Verlauf der Abarbeitung mehrerer Prozesse lässt sich gut mit einem „Gantt-Diagramm“ darstellen (unten rechts).

Die Zeitscheiben sind mit Absicht von Prozess zu Prozess und auch in einer Prozesslinie unterschiedlich lang gewählt.



Beachte, dass die Zeit für das Umschalten zwischen den Prozessen in den vorangegangenen Diagrammen vernachlässigt worden ist!

Der *genaue* zeitliche Ablauf bei mehreren Prozessen kann bei mehreren Läufen stark variieren. Ein Prozess kann eventuell zwischenzeitlich nicht weiterarbeiten, weil er von einer I/O-Operation abhängt, und die Geräte haben zeitliche Schwankungen. Ein Gerät ist auch der Timer, der für die Zeitscheiben-Einteilung benötigt wird. Kleine Verschiebungen können dazu führen, dass Prozesse in unterschiedlicher Reihenfolge an ihre I/O-Anweisung (zum selben Gerät) gelangen und also in umgekehrter Reihenfolge blockieren.

Die Folge davon ist, dass Prozesse keinesfalls feste Annahmen über ihre absoluten oder relativen Laufzeiten zueinander machen dürfen. (Ausnahmen sind Echtzeit-Systeme zum Steuern und Regeln, etc.)

4.2 Prozess-Hierarchien

Üblicherweise entstehen zur regulären Laufzeit des Betriebssystems (nach dem Start und Initialisierungen) immer wieder neue Prozesse, und alte werden aus dem System entfernt.

Wenn alle System-Aktivität innerhalb irgendwelcher Prozesse erfolgt, entsteht also notwendigerweise eine Prozess-Hierarchie (das Erzeugen eines Prozesse geschieht innerhalb eines anderen Prozesses). Eine Stufe wird dabei durch das Paar erzeugender/erzeugter Prozess gebildet. Häufigere Bezeichnungen sind Elter-Prozess (parent process) und Kind-Prozess (child process).

Je nach System bestehen danach keine oder mehr oder weniger komplizierte Verbindungen zwischen Elter- und Kind-Prozess: geteilte Ressourcen, geteilter Speicher, etc.

MS-DOS: Uniprogramming

Es gibt keine parallel laufenden Prozesse. Ein Programm kann zwar ein anderes Programm aufrufen und starten – das aufrufende Programm wird aber in der Zwischenzeit (also bis das Kind terminiert) angehalten.

Die Prozess-Hierarchie ist in MS-DOS also linear, als Graph (Knoten = Prozesse, Pfeil = Erzeugungsvorgang): 

UNIX: Multiprogramming

Ein Prozess startet einen Kind-Prozess mit dem Aufruf der System-Routine `fork`.

- Dabei wird der aktuelle Prozess **verdoppelt** (identisch kopiert).
- Danach unterscheiden sich die beiden Prozesse nur in einer einzigen Sache: dem Rückgabewert von `fork` (beim Kind 0, beim Elter die Prozess-Nummer des Kinds).
- Beide Prozesse laufen danach „parallel“ weiter (direkt nach dem `fork` bekommt aber meist zunächst das Kind die CPU).
- Üblicherweise überlädt sich der neue Prozess dann mit einem neuen Programm (z.B. durch einen `exec`-Systemaufruf).

Daher gibt es in UNIX einen „Über-Prozess“, der Vorgänger aller anderen Prozesse ist. Die Prozess-Hierarchie ist verzweigt, als Graph ein Baum mit dem Start-Prozess als Wurzel.

DEC VMS: Multiprogramming
Prozesse duplizieren sich hier nicht. Es wird zunächst eine neue Prozess-Struktur im System angelegt, dann mit einem Programm verbunden, das Programm geladen, dann der neue Prozess gestartet (ähnlich auch in anderen Systemen, z.B. Amiga OS).

Windows NT: Multiprogramming Es ist sowohl eine Prozess-Erzeugung mit neuem Code, wie auch eine Prozess-Duplizierung möglich.

4.3 Prozess-Status

Prozesse können aktiv und auch schon teilweise abgearbeitet worden sein, brauchen aber nicht zu jedem Zeitpunkt lauffähig zu sein:

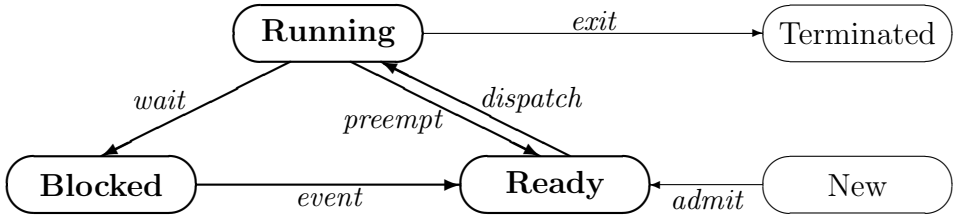
Ein Prozess kann

- auf ein Eingabedatum warten müssen,
- auf einen anderen Prozess warten müssen,
- sich freiwillig einige Zeit „schlafen legen“.

Dieser Zustand darf nicht verwechselt werden mit dem, der vorliegt, wenn ein Prozess gerade nicht die CPU-Kontrolle hat – aber wenn er sie *hätte*, direkt weiterlaufen *könnte*.

So ergeben sich die typischen *drei* möglichen Statūs eines Prozesses:

1. **Running** besitzt gerade die CPU-Kontrolle
2. **Ready / Runnable** lauffähig, aber gerade angehalten, weil ein anderer Prozess die CPU besitzt
3. **Blocked / Waiting** nicht lauffähig, wartet auf ein Ereignis von außen (I/O, Signal)



Zu den einzelnen Übergängen im Diagramm:

admit	der neu geschaffene Prozess wird eingeordnet
wait	der Prozess ruft eine Systemroutine (meist I/O) auf
event	erwartete Eingabe/erwartetes Signal trifft ein
preempt	ein anderer Prozess erhält die CPU (Scheduler-Entscheidung, s.u.)
dispatch	der Prozess erhält die CPU (Scheduler-Entscheidung, s.u.)
exit	der Prozess beendet sich selbst (ggf. auch: wird abgebrochen)

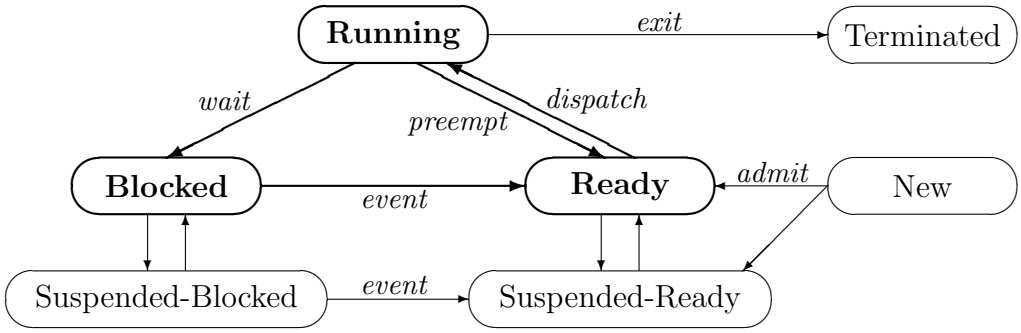
Beachte hierbei:

- Bei einem Rechnersystem mit nur einem Prozessor ist immer genau ein Prozess im Zustand *Running*. Wenn unter UNIX keine anderen Prozesse lauffähig sein sollten, wird der zu allererst erzeugte Prozess (Nummer 0) *running* und führt eine leere Endlosschleife aus (daher manchmal die Bezeichnung Idle-Prozess). Sobald ein anderer Prozess *ready* wird, wird er unterbrochen.
- Das Betriebssystem führt meist zwei Listen (Schlangen, Priority-Queues o.ä.) mit Prozess-Informationen:

  Waiting Queue für Prozesse, die gerade *blocked* sind

  Ready Queue für Prozesse, die gerade *ready* sind

- Der Übergang *wait* entsteht *aktiv im Prozess* (System-Aufruf für die Eingabe, Selbstblockade, Warten auf Signal). Die Übergänge *preempt* und *dispatch* können dagegen jederzeit vom System vorgenommen werden – an jeder beliebigen Stelle im Programm. Der Prozess selbst bemerkt davon nichts.
- Der Übergang *event* überführt den Prozess von *blocked* nach *Ready*, nicht direkt nach *running*.

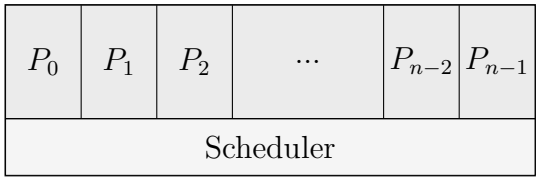


Eventuell werden zwei weitere Zustände eingeführt. Prozesse, die sehr lange blockiert sind, werden aus dem Hauptspeicher auf Platte ausgelagert und in den Zustand *Suspended-Blocked* überführt. Wenn das erwartete Ereignis eintritt, gelangen sie in den Zustand *Suspended-Ready*. Um wirklich *Ready* zu werden, müssen sie dann erst wieder von Platte in den Hauptspeicher gelesen werden.

Weitere Zustände entstehen bei Systemen mit Dual-Mode-Befehlen, wenn man Prozesse, die gerade im Systemmodus laufen, von solchen unterscheidet, die gerade im Usermodus laufen. Diese Unterscheidung macht deswegen Sinn, weil die Prozesse nach einer Unterbrechung in den vorherigen Modus zurückkehren müssen.

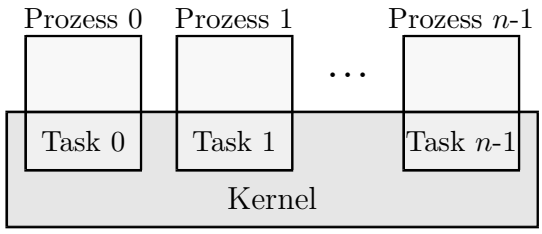
Der Teil des Betriebssystems, der die Entscheidung trifft, welcher Prozess jeweils als nächster die CPU erhält, heißt **Scheduler**. Zu seinen Aufgaben gehört es damit auch, das Betriebsmittel „CPU-Zeit“ möglichst *fair* zu vergeben.

Eine entsprechende mögliche Sicht des Betriebssystems ist die folgende: als unterste Ebene wird der Scheduler angesehen. Alles andere, also Benutzer-Programme und alle anderen Betriebssystem-Teile, liegt darüber in Form von Prozessen.



Das ist die traditionelle Betriebssystem-Sicht. Der Code zum Umschalten zwischen den Prozessen liegt also konzeptionell *außerhalb* der Prozesse.

Unix verwendet ein anderes Schema als das oben angegebene traditionelle. Der Code zum Prozess-Wechsel liegt konzeptionell in den Prozessen, d.h. der *Kernel* wird *innerhalb* der Prozessen ausgeführt. In L*i*nux wird daher auch namentlich unterschieden zwischen „Prozess“ und „Task“.



Die im System verwendete Konstruktion wird als Task bezeichnet. Prozess heißt der Teil der Task, der im Usermodus abläuft (z.B. der normale Code eines Anwenderprogramms). Der Teil, der im privilegierten Modus abläuft, besteht aus Kernel-Code und hat freien Zugriff auf alle Systemressourcen und auf andere Tasks. Der User-Teil hat keine Möglichkeit, auf diese Daten zuzugreifen, da sie außerhalb seines „logischen Adressraums“ liegen.

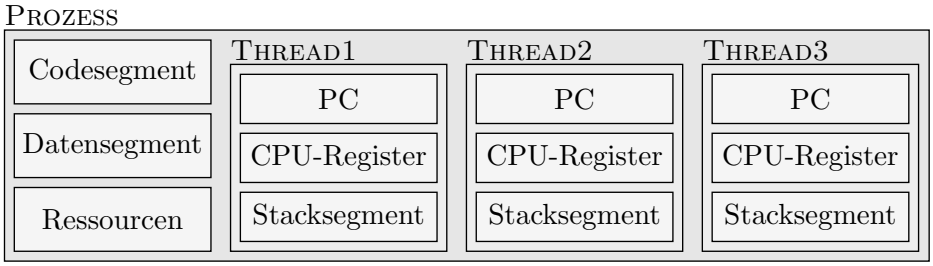
4.4 Threads

Klassischerweise ist der Prozess eigentlich die Grundeinheit für *zwei* Dinge:

- Einheit für Ressourcen-Benutzung (Speicher, Dateien)
- Einheit für CPU-Benutzung

In einigen Systemen wurde eine neue, feinere Einheit geschaffen, die sich nur auf die CPU-Benutzung bezieht. Diese Einheit heißt **lightweight process** oder häufiger **Thread**.

Meist ist ein Thread als ein *sequentieller* Strom von CPU-Aktivität *innerhalb* eines Prozesses definiert. Der Prozess bleibt die Einheit der Ressourcen-Benutzung. Mehrere Threads dürfen innerhalb eines Prozesses „leben“ und teilen sich dann die Ressourcen. Ein entsprechendes Vorgehen heißt **Multithreading**.



- Die Threads im selben Prozess *teilen* sich den *Maschinencode* und das *Datensegment*!

  Ihre Ausführung findet aber zum selben Zeitpunkt an i.Allg. *unterschiedlichen Stellen* des Codes statt. Daraus ergibt sich, dass sie eigene Kopien der CPU-Register (inklusive Programmzähler, Stack-Pointer etc.) benötigen. Außerdem sollen sie natürlich unabhängig voneinander verschachtelt Unterprogramme aufrufen können und benötigen dazu einen eigenen Stack-Bereich.

- Außerdem besitzt jeder Thread einen eigenen Status. Es kann also durchaus ein Thread innerhalb eines Prozesses ready, ein anderer durch I/O blockiert sein.

- Ein Prozess im traditionellen Sinn („heavyweight process“) ist dann ein Prozess mit genau einem Thread. Wenn man einen Prozess in mehrere Threads aufteilt, ist die Bezeichnung „sequential process“ natürlich nicht mehr zulässig. Wenn ein Prozess entsteht, entsteht automatisch ein Thread, der danach weitere Threads erzeugen kann, sodass wiederum eine baumartige Hierarchie entsteht.

- Durch die Unterteilung eines Prozesses kann beispielsweise der **Durchsatz** erhöht werden. Während ein Thread mit dem Abspeichern einer Datei beschäftigt ist (bzw. blockiert ist und auf das I/O-Ende wartet), kann ein anderer Thread bereits sinnvolle andere Dinge tun.

- Insbesondere bieten Threads Vorteile bei der **Kommunikation** zwischen mehreren Prozessen. Jeweils ein Thread kann sich asynchron um die Bereitstellung (Producer-Prozess) bzw. das Abholen (Consumer-Prozess) der auszutauschenden Daten kümmern. Der Rest der Threads in den beiden Prozessen braucht nicht unbedingt zu warten.

  Besonders geeignet für Multithreading sind Server-ähnliche Prozesse. Sie sollen gleichzeitig Anfragen von mehreren anderen Prozessen oder Rechnern beantworten können. Jede solche Bearbeitung eignet sich für einen eigenen Thread, nicht aber unbedingt für einen eigenen Prozess. Code und die meisten Daten sind allen Ausführungsströmen gemein.

- Traditionelle Multitasking-Systeme wie ältere UNIXe unterstützen kein Multithreading. Einige Systeme stellen Threads direkt auf der Kernel-Ebene zur Verfügung (OS/2, Windows NT, Sun OS, Mach; solche Threads heißen dann **Kernel-Threads**). Eine Thread-Umschaltung ist ein normaler Prozesswechsel mit allen damit verbundenen aufwendigen Operationen.

  Bei anderen Systemen findet die Unterteilung auf höherer Ebene statt, beispielsweise durch Bibliotheksroutinen, mit denen das Anwenderprogramm seine Threads selbst organisiert (**User-Level-Threads**). Das Betriebssystem sieht dabei nur den umgebenden Prozess; es finden keine vollständigen Prozessumschaltungen statt.

  User-Level-Threads sind daher meist um Größenordnungen schneller als Kernel-Threads. Das Anwenderprogramm muss sie zwar selbst verwalten; dafür kann das Scheduling aber flexibel direkt dem jeweiligen Problem angepasst werden.

Um Threads und Prozesse eindeutig auseinanderzuhalten, werden wir manchmal von *Tasks* statt Prozessen reden.

4.5 Prozess-Kontrollblöcke

Die Systemstruktur, mit der Prozesse verwaltet werden, heißt Prozess-Kontrollblock (process control block, **PCB**). In höheren Programmiersprachen wird sie durch eine **`struct`** bzw. einen **`record`** dargestellt, in objektorientierten Sprache ggf. als Klasse.

Die Sammlung der PCBs aller Prozesse erfolgt meist in Form einer Tabelle (als Array), die dann **Prozess-Tabelle** genannt wird.

Wenn ein anderer Prozess die CPU-Kontrolle erhalten soll (Zeitscheibenende o.ä.), werden alle relevanten Informationen über den Zustand des bis gerade gelaufenen Prozess (der „Kontext“

des Prozesses) in seinem PCB abgelegt. Dann wird der PCB des neuen Prozesses ausgewertet, um den Zustand vor dessen letzter Unterbrechung wiederherzustellen.

Dieser Umschaltvorgang heißt **context switch** und ist eine ziemlich „teure“ (zeitaufwendige) Operation, besonders wenn viele Register gerettet bzw. restauriert werden müssen (bei SPARC-Architektur, bei Fließkommaregistern, komplexer Speicherverwaltung etc.). Er sollte so schnell wie irgend möglich implementiert sein.

Ein PCB enthält zumindest folgende Informationen:

- den Prozess-Status (also *running*, *blocked*, *ready* o.ä.)
- den Programmzähler (das Prozessor-Register, Adresse des nächsten Befehls)
- alle anderen CPU-Register (Arithmetik, Indizierung, Stack, etc.)
- Speicher-Informationen (Adressbereiche von Code, Daten, Heap)
- Informationen über noch verwendete Ressourcen
- diverse Verwaltungsinformationen: Prozess-ID-Nummer, ID des Elter-Prozesses, Prozess-Priorität, Startzeit, verstrichene CPU-Zeit, etc.

Beispiel: PCBs unter L*i*nux

In L*i*nux ist der PCB eine C-Struktur namens `task_struct` (definiert in der Header-Datei `linux/sched.h`). Zu beachten ist hier, dass man von normalen Prozessen aus keinen (erlaubten) Zugriff auf diese Struktur hat. Es handelt sich hier um *Code des Kernels*, der hier zu Dokumentationszwecken studiert wird!

Es gibt eine globale Variable `extern struct task_struct *current` (im Kernel-Code!), die auf den PCB des jeweils aktuellen Prozesses zeigt, und mit der ein Task an den eigenen PCB gelangen kann.

Wegen der Länge der PCB-Struktur sind im Folgenden nur einige ausgewählte Member aufgeführt. Der Teil vor dem ersten ... ist ungekürzt angegeben. Er sollte nie verändert werden, da Routinen auf ihn zugreifen, die in Assembler geschrieben sind, und die sich auf die relative Position der Member in der Struktur verlassen können müssen.

```
struct task_struct
 {
   volatile long state;          // Prozess-Status
   long counter;                 // maximale Ticks bis zum Scheduling
   long priority;                // 1 .. 35 (dazu später)
   unsigned long signal;         // Bitmaske, eingetroffene Signale (⇒ max. 32)
   unsigned long blocked;        // Bitmaske, momentan blockierte Signale
   unsigned long flags;          // Statusflags, z.B. für Trace-Modus
   int errno;                    // letzter Fehlercode (durch einen Systemaufruf)
   int debugreg[8];              // Prozessor-Debuggingregister
   // ...
   struct task_struct *next_task, *prev_task;    // verkettete Liste aller Tasks
```

```
  struct task_struct *p_opptr, *p_pptr;          // Elter(n)
  int pid;                          // Prozessidentifikationsnummer
  int pgrp;                         // Prozessgruppe
  struct files_struct *files;       // offene Dateien
  struct mm_struct *mm;             // Info für Memory Management
  // ...
};
```

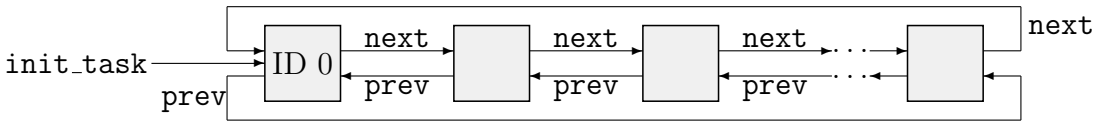
state: der Prozess-Status, **volatile** durch System-Zugriff. Dieser Eintrag kann folgende Werte annehmen (die Konstanten sind in derselben Header-Datei definiert):

TASK_RUNNING	der Task hat gerade die CPU-Kontrolle
TASK_INTERRUPTIBLE	*blocked* und durch Signale aufweckbar
TASK_UNINTERRUPTIBLE	*blocked*, nicht durch Signale aufweckbar
TASK_ZOMBIE	beendet, aber noch in der Prozesstabelle
TASK_STOPPED	Stop durch ptrace im Einzelschrittmodus
TASK_SWAPPING	der Task wartet auf eine Speicherseite

Bemerkung: Das ps-Kommando zeigt in der Spalte STAT auch die L*i*nux-spezifischen Statūs an: D=uninterruptible sleep, Z=zombie, T=trace (stopped), W=waiting (swapping).

counter: hier sind die Ticks (1 Tick=10 msec) eingetragen, die höchstens bis zum nächsten Verdrängungsvorgang vergehen werden. Der L*i*nux-Scheduler wählt den nächsten Prozess, der die CPU erhält, immer unter denen mit dem höchsten count-Eintrag aus.

next_task, prev_task: Alle Tasks sind durch diese beiden Einträge in eine doppelt verkettete Liste eingehängt (ringförmig). Es gibt eine globale Variable `extern struct task_struct init_task`, den PCB des Startprozesses (also des Idle-Prozesses), der als Einstieg in die Liste verwendet werden kann.



Diese Liste ist nicht zu verwechseln mit den oben erwähnten Warteschlangen (Ready Queue, Waiting Queue) zur Verwaltung des Scheduling!

In sched.h ist auch ein praktisches Makro definiert, das die System-Routinen benutzen können, die eine Operation auf allen Tasks durchführen und dazu die Liste durchlaufen müssen:

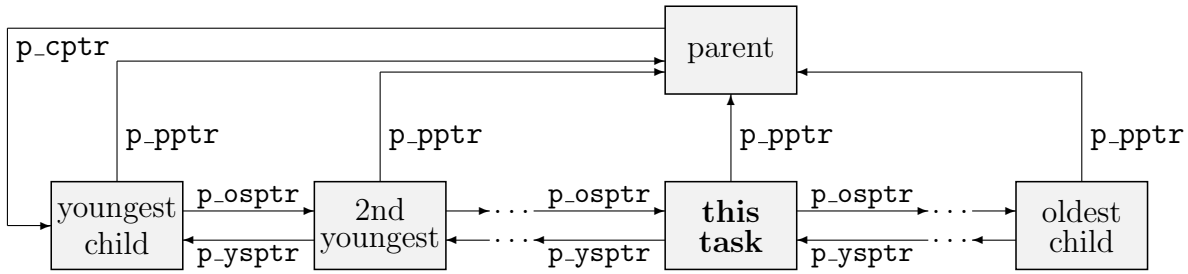
```
#define for_each_task(p) \
  for (p=&init_task ; (p=p->next_task) != &init_task ; )
```

Der Spezialfall init_task wird dabei also automatisch übersprungen.

p_pptr (process parent pointer) ist ein Zeiger auf den PCB des Elter-Prozesses. Unter UNIX kann ein Kind nach der Erzeugung einem anderen Prozess als Kind „untergeschoben“ werden. Der ursprüngliche Elter-Prozess ist immer unter p_opptr (original pptr) abgelegt.

Die Kinder eines Prozesses sind auch untereinander in einer Liste verkettet – über die Einträge `p_ysptr` (younger sibling ptr) und `p_osptr` (older sibling ptr). Den Anfang der Liste findet man mit `p_cptr` (das zuletzt erzeugte Kind).

Insgesamt hat man also folgende Beziehungen:



Alle PCBs werden unter UNIX in der Prozesstabelle abgelegt. In fast allen UNIX-Versionen wird diese Tabelle beim Booten statisch angelegt, besitzt also eine feste Maximalgröße, so auch (hauptsächlich aus historischen Gründen) bei den bisherigen L*i*nux-Versionen:

```
struct task_struct *task[NR_TASKS];
```

Die Konstante `NR_TASKS` ist in `linux/tasks.h` definiert, beispielsweise:

```
#define NR_TASKS 512
```

In älteren Versionen stand hier 128. Wenn man feststellt, dass man mit der angegebenen Maximalzahl nicht auskommt, kann man die Header-Datei ändern und muss den Kernel neu übersetzen. In späteren L*i*nux-Versionen werden die PCBs wahrscheinlich irgendwann einmal auch dynamisch angelegt werden, sodass diese Beschränkung ganz entfällt.

Die Reihenfolge der Tasks in diesem Array hat (nach einiger Zeit) nichts mit der logischen Reihenfolge (durch die Verkettung der PCBs) zu tun.

Die Einträge ab `task[2]` werden nach dem Booten dynamisch belegt und geleert. `task[1]` ist der Init-Prozess, der das Programm `init` zur Initialisierung des Systems ausführt. `task[0]`, der Idle-Prozess, ruft folgende Funktion auf (`linux/init/main.c`):

```
int cpu_idle(void *unused)
 {
   for (;;) idle();
 }
```

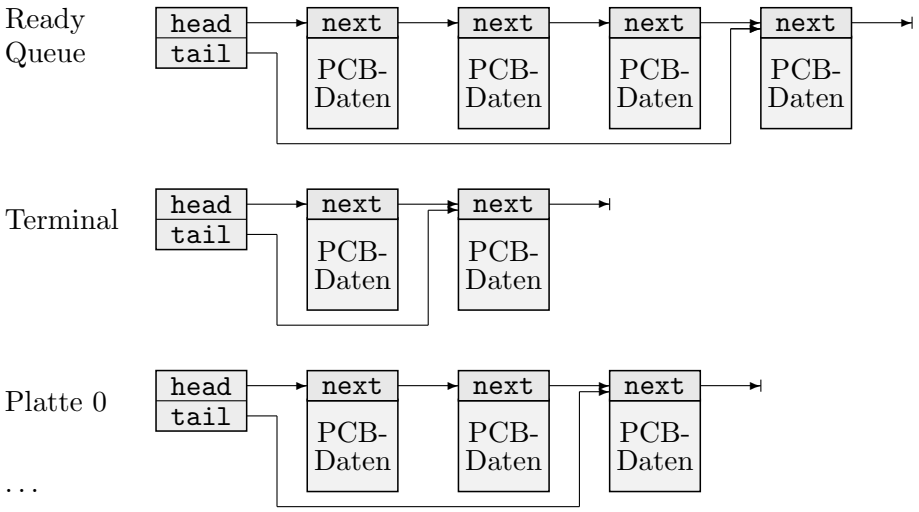
`idle` ist eine vom genauen Prozessor abhängige Maschinenroutine, die aber im wesentlichen nichts tut.

4.6 Dispatching

Die Einzelheiten der Prozess-Umschaltung, also Register retten/restaurieren, Warteschlangen verwalten, erledigt der **Dispatcher**. Es steckt keine große System-Philosophie hinter diesen rein technischen Mechanismen. (Welcher Prozess als nächstes die CPU-Kontrolle erhält, entscheidet dagegen der Scheduler.)

Die **Waiting Queue** und die **Ready Queue** verketten die *PCBs* der wartenden bzw. lauffähigen Prozesse. Damit bei Eintreten eines I/O-Ereignisses aber nicht die gesamte Waiting Queue

nach Prozessen durchsucht werden muss, die auf das entsprechende Gerät warten, gibt es meist eine Queue *pro Gerät*, die dann **Device Queue** heißt.



Beispiel: Dispatching unter L*i*nux

In L*i*nux werden nicht direkt die PCBs, sondern Zeiger auf sie eingetragen. Die verwendete Struktur ist folgende (aus `include/linux/wait.h`):

```
struct wait_queue
 {
   struct task_struct * task;
   struct wait_queue * next;
 };
```

Die Schlangen werden am einfachsten mit zwei Inline-Funktionen aus `include/linux/sched.h` bearbeitet: `add_wait_queue` und `remove_wait_queue`.

In L*i*nux erfolgt das Abmelden eines Prozesses im Task-Teil des Prozessess selbst. Ein Prozess, der auf ein Ereignis warten möchte, trägt *sich selber* in die passende Warteschlange ein! Dazu ruft er die Funktionen `interruptible_sleep_on` oder `sleep_on` auf (in `kernel/sched.c`), die folgende Funktion aufrufen:

```
static inline void __sleep_on(struct wait_queue **p, int state)
 {
   unsigned long flags;
   struct wait_queue wait = { current, NULL };

   if (!p) return;                          // keine Warteschlange angegeben
   if (current == task[0])                  // Quatsch, P0 darf nicht schlafen
      panic("task[0] trying to sleep");
   current->state = state;                  // Status eintragen
   save_flags(flags);                       // Prozessor-Flags sichern (asm)
   cli();                                   // Interrupts sperren
   __add_wait_queue(p, &wait);              // in die Schlange eintragen
   sti();                                   // Interrupts wieder zulassen
```

```
    schedule();                             // Nachfolgetask bestimmen

    // Achtung: hier verabschiedet sich der aktuelle Task!!!
    // Wenn er wieder aufgeweckt wird, geht's mit folgendem Code weiter:

    cli();                                  // Interrupts sperren
    __remove_wait_queue(p, &wait);          // aus der Schlange herausnehmen
    restore_flags(flags);                   // Prozessor-Flags restaurieren
}
```

Der Prozess wird aufgeweckt, wenn ein anderer Task `wake_up` oder `wake_up_interruptible` für dieselbe Warteschlange aufruft. Dabei könnte es sich z.B. um den Gerätetreiber eines I/O-Geräts handeln, etc.

4.7 Scheduling

Der Scheduler ist der Teil des Betriebssystems, der bei einem Prozesswechsel und mehreren lauffähigen Prozessen denjenigen unter ihnen auswählt, der die CPU als nächstes erhält.

Von der Strategie dieser Auswahl hängt ein Großteil der Performance des gesamten Systems ab. Aufgrund der Komplexität der entsprechenden Überlegungen verschieben wir die Betrachtung des Schedulers auf ein gesondertes Kapitel.

5 Shells

Ein Kommandozeileninterpreter ist die traditionelle Benutzerumgebung sowohl in UNIX wie auch in MS-DOS. Von hier aus kann man Systembefehle und Programme aufrufen und mit Parametern versehen. Unter UNIX muss eine Shell das Umgehen mit beliebig vielen Prozessen unterstützen und erleichtern.

In UNIX wird der Interpreter normalerweise als Shell bezeichnet, da er für den Benutzer wie eine Muschel um die eigentlichen Systemaufrufe gelegt ist. Der Standard-Interpreter in MS-DOS ist `COMMAND.COM` und ebenfalls ein ganz zentraler Bestandteil des Systems.

Grafische Benutzeroberflächen erleichtern zwar den Umgang mit einigen Systemschnittstellen, geübte Benutzer arbeiten aber besonders mit den ausgereiften UNIX-Shells meistens effektiver.

Die Shell ist nicht nur ein Befehlszeilen-Interpreter, sondern ein Interpreter für eine vollständige Programmiersprache, die „Skript-Sprache“ der Shell. Entsprechende Programme heißen *Shell-Skripte* und bestehen im einfachsten Fall aus einer Folge von Befehlszeilen. Die Möglichkeiten gehen aber weit über die einfacher Batch-Dateien hinaus. Vorab werden wir uns nur mit der Ausführung einzelner Kommandos befassen.

Da die Möglichkeiten bei der Standard-Shell unter MS-DOS, nämlich `COMMAND.COM` extrem eingeschränkt sind, wollen wir uns zunächst nur mit UNIX-Shells beschäftigen. DOS ist am Schluss ein eigener Abschnitt gewidmet.

5.1 Standard-Shells

Die verbreitetsten Shells sind:

`sh`	Bourne-Shell (die klassische Shell)
`csh`	für `C`-Programmierer (plus Wortspiel mit „seashell“)
`ksh`	Korn shell
`tcsh`	TC-Shell, `csh`-Variante („Cornell-Shell“)
`bash`	Bourne-Again-Shell, Standard unter L*i*nux

Die Shells sind üblicherweise in `/bin` untergebracht. Eventuell ist `/bin/sh` ein Link auf eine der anderen Shells wie `bash`.

Die meisten Shells sind (in Optionen und Skript-Sprache) kompatibel zur Bourne-Shell, nur die `csh` (und damit `tcsh`) macht eine Ausnahme. Es kann hier allerdings nicht an jeder Stelle darauf hingewiesen werden, wenn die `csh` wieder einmal eigene Wege geht. Im Ernstfall sollte man die `sh` oder `bash` benutzen.

Außer in der Skript-Sprache unterscheiden sich die Shells im Komfort:

- Bei der `bash` kann man beispielsweise mit den Cursor-Tasten Up und Down in den letzten Eingaben blättern, angefangene Dateinamen werden mit der `TAB`-Taste ergänzt, etc.
- Bei der `csh` und der `bash` kann man mit dem Ausrufezeichen auf vorhergehende Befehle zugreifen: `!!` wiederholt den letzten Befehl, `!-5` den fünftletzten, `!`*string* den letzten, der mit der Zeichenfolge *string* begann. (Es gibt noch einige weitere Möglichkeiten, die angesprochenen Befehle vor der Ausführung zu verändern.) Mit `CTRL-R` kann man einen vorangegangenen Befehl über einen Suchbegriff auswählen.

Für jeden Benutzer kann aber eine eigene Standard-Shell in `/etc/passwd` angegeben werden. Das Kommando „`finger` *user*“ gibt u.a. die entsprechende Information aus:

```
Login: axel                             Name: na, ich
Directory: /home/axel                   Shell: /bin/bash
...
```

Der Benutzer kann auf vielen Systemen hier nachträglich mit dem `chsh`-Kommando (*change shell*) eingreifen und seine Lieblingsshell eintragen:

```
chsh -s /bin/bash axel
```

Es dürfen hinter „`-s`“ nur Programme angegeben werden, die in `/etc/shells` als Shells registriert sind.

In der ersten Zeile einer Skript-Datei kann man die Wunsch-Shell angeben (mit komplettem Pfad, keine POSIX-Festlegung!), beispielsweise:

```
#! /bin/csh
```

Die Shells haben eine Option „`-c`“, die das dahinter angegebene Programm/Skript direkt zur Ausführung bringt (danach stirbt die Shell):

```
bash -c myskript
```

5.2 Befehlszeilen

Eine Befehlszeile sieht in etwa wie folgt aus:

```
command1 options1 arguments1 ; command2 options2 arguments2 ... # comment
```

Sie besteht aus einem oder mehreren Befehlen, die dann mit Semikolon voneinander getrennt sind und *nacheinander* ausgeführt werden. Üblicherweise sind also alle den ersten Teil betreffenden Prozesse beendet, bevor der zweite Teil bearbeitet wird. Wir beschränken uns daher hier zunächst auf *einen* Befehl.

`command` steht jeweils für einen eingebauten Befehl (s.u.) oder den Dateinamen eines externen Programms. In welchen Verzeichnissen die Shell nach diesen Programmen sucht, wird in der Umgebungsvariable `$PATH` festgelegt. „`echo $PATH`“ könnte folgendes ausgeben:

```
/bin:/usr/bin:/root/bin:.:/usr/local/bin:/usr/X11R6/bin:/usr/bin/TeX
```

Die Verzeichnisnamen werden mit Doppelpunkten getrennt. Hier wird nacheinander in `/bin`, in `/usr/bin` etc. gesucht.

Mit Optionen sind Steuerungs-Angaben gemeint, die mit ‘`-`’ beginnen. Argumente sind Parameter, die das Kommando benötigt, etwa Datei- und Verzeichnisnamen. Genaueres folgt am Ende des Kapitels im Abschnitt über die Auswertung von Optionen.

- Eine Kurzübersicht über die typischsten Kommandos findet sich im Anhang.
- Der einfachste Shell-Befehl ist „`:`“, der *nichts* tut. Seine Argumente werden allerdings ausgewertet.

- Wenn eine (physische) Zeile auf einen Backslash '\' endet, wird sie mit der nächsten zu einer einzigen (logischen) Zeile verbunden. Das ermöglicht eine vernünftige Formatierung bei überlangen Kommandozeilen (`echo` ist der Standard-Befehl zur Ausgabe):

```
echo gib das aus \
     und das \
     und das auch noch!
```

- Die Shell deutet dem Benutzer mit einem „Prompt" an, dass sie bereit ist, einen Befehl anzunehmen. Dieser Prompt kann vom Benutzer eingestellt werden (s.u.), voreingestellt ist meist „`$`".

  Manchmal ist am Ende einer physischen Zeile eine logische Struktur noch nicht beendet, z.B. eine Fallunterscheidung (dazu später). In diesem Fall meldet die Shell das mit einem „sekundären Prompt" (voreingestellt „`>`"). Es werden so lange solche Fortsetzungszeilen angenommen, bis die logische Struktur komplett ist:

```
$ if test -r prog.c; then
> echo prog.c existiert
> fi
$
```

  Das Aussehen der beiden Prompts wird in den Variablen `$PS1` und `$PS2` festgelegt (siehe 5.10.5).

- Die Shell arbeitet mit einigen Steuerzeichen:

`"`	Double-Quote	`*`	Wildcard	`\`	Escape, Fortsetzungszeile
`#`	Kommentarzeichen	`;`	Befehls-Sequencer	`]`	Wildcard
`$`	Variablen-Substitution	`<`	Eingabe-Umleitung	‘	Back-Quote (ASCII 96)
`&`	Hintergrundprozess, log. Und	`>`	Ausgabe-Umleitung	{	Gruppierung
’	Single-Quote (ASCII 39)	`?`	Wildcard	|	Pipes, logisches Oder
`(`	Subshell-Gruppierung	`[`	Wildcard	}	Gruppierung
`)`	Subshell-Gruppierung				

  Ihre genaue Bedeutung wird in den folgenden Abschnitten erklärt. Wenn diese Zeichen nicht in ihrer Steuerfunktion, sondern als normale ASCII-Zeichen gemeint sind, müssen sie mit einem Escape-Zeichen '\' „maskiert" werden. Beispielsweise editiert „`vi \$\*.cpp`" die Datei „`$*.cpp`" (mit Dollar und Sternchen im Dateinamen). Man sollte diese Zeichen aber besser nicht in Dateinamen verwenden.

- Einige Kommandos sind direkt in die jeweilige Shell eingebaut. Die Angabe ihrer Namen führt also nicht zum Aufruf eigener Programme und erzeugt meist auch keine neuen Prozesse. „`exit`" ist beispielsweise die Anweisung zur Beendigung der Shell.

- Eine sehr wichtige Aufgabe der Shell ist auch das **Expandieren** von Wildcards („Jokerzeichen"). Wenn man den Befehl „`ls *.cpp`" absetzt, wird *nicht* etwa das Programm „`ls`" mit dem Parameter „`*.cpp`" aufgerufen, und das Programm vergleicht alle Dateinamen im aktuellen Verzeichnis mit dem Muster „`*.cpp`".

Vielmehr durchsucht bereits die Shell das aktuelle Verzeichnis und ersetzt das „`*.cpp`“ durch eine Liste aller passenden Dateinamen. Diese fertige Liste kommt dann beim Befehl „`ls`“ an. Der Aufruf entspricht dann beispielsweise „`ls myshell.cpp test.cpp test2.cpp`“!

Wenn die Wildcards bei den Kommandos selbst ankommen sollen, muss man sie mit ‘\’ maskieren oder das Argument in Anführungszeichen einschließen (s.u.).

Es gibt folgende Arten von Wildcards:

`?`	steht für genau ein beliebiges Zeichen,
`*`	steht für beliebig viele beliebige Zeichen (evtl. 0 Stück),
`[`*charset*`]`	steht für genau ein Zeichen aus der Menge *charset*.

Die Mengenschreibweise ist definiert über

$$charset \rightarrow (char|char\texttt{-}char)^*$$

Dabei ist *char* ein beliebiges druckbares Zeichen. Erlaubt ist also beispielsweise `[aeiou]` (alle kleinen Vokale), `[a-z]` (alle Kleinbuchstaben) oder kombiniert `[a-zA-Z0-9_]` (alle in `C`-Bezeichnern zulässigen Zeichen).

„`ls test[123]?.*`“ könnte beispielsweise `test14.c` `test29.tex` `test31.pas` auflisten.

Beachte: Unter MS-DOS (in `COMMAND.COM`) müssen alle Kommandos, die korrekt auf Wildcards reagieren sollen, das Expandieren *selbst* vornehmen!

- Die Bestandteile der Befehlszeile, also Argumente und Optionen, ggf. zusätzliche Shell-Steuerzeichen, werden durch Spaces oder Tabulatoren voneinander getrennt.

  Wenn ein Argument aber Spaces enthalten soll, muss es mit Anführungszeichen ‘`"`’ eingeschlossen werden. Innerhalb dieser Anführungszeichen nimmt die Shell dennoch eine Expansion von Variablennamen vor (nicht von Wildcards)! Wenn auch das unterbunden werden soll, verwendet man Hochkommata ‘’’ (ASCII 39). Vorsicht: Die *umgekehrten* Hochkommata ‘`’ (ASCII 96) haben eine andere Bedeutung, s.u.

5.3 I/O-Umlenkung

Die drei I/O-Kanäle des neu zu startenden Prozesses kann man mit Shell-Angaben auf Dateien umlenken, aus denen dann statt von der Tastatur gelesen, bzw. in die statt auf den Bildschirm geschrieben wird (*I/O-Redirection*):

- „`<` *filename*“ lenkt die Standard-Eingabe auf die Datei *filename* um
- „`>` *filename*“ lenkt die Standard-Ausgabe auf die Datei *filename* um
- „`2>` *filename*“ lenkt den Standard-Fehlerkanal auf die Datei *filename* um

- Die 2 in „`2>`“ ist der *File-Deskriptor* des Fehlerkanals (0 und 1 funktionieren analog). Man kann auch „`<&`*n*“ bzw. „`>&`*n*“ schreiben und mit dem File-Deskriptor *n* statt mit einem Filenamen arbeiten. Einige Shells unterstützen all diese Konstruktionen leider nicht.

- Wenn es die Ausgabe-Dateien schon gibt, werden sie *überschrieben*. Mit „`>>`“ statt „`>`“ kann man dagegen an eine bestehende Datei *anhängen*.

- Da Gerätetreiber ja wie normale Dateien angesprochen werden können, sind auch sie als Quelle oder Ziel einer solchen Umlenkung erlaubt:

  ```
  ls -al > /dev/console
  ```

- Durch die Trennung von Ausgabekanal und Fehlerkanal kann man die normalen Ausgabedaten eines Prozesses in eine Datei schreiben, während seine Fehlermeldungen weiterhin auf dem Bildschirm erscheinen.

- Wenn Standardausgabe und Fehlerkanal auf dieselbe Datei umgelenkt werden sollen, kann man das *nicht* wie folgt tun:

  ```
  myprog >out.txt 2>out.txt
  ```

  Die Shell öffnet immer zuerst die Datei für die Ausgabe. Beim Versuch, die Datei auch für die Fehlermeldungen zu öffnen, scheitert sie – aber ohne einen Fehler zu melden. Die Fehlerausgaben sind verloren.

  Stattdessen *muss* man mit dem File-Deskriptor 1 arbeiten:

  ```
  myprog >out.txt 2>&1
  ```

- Man kann außerdem die Ausgabe eines Kommandos als *Argument* eines anderen verwenden. Dazu dienen die umgekehrten Hochkommas:

  ```
  echo Es ist der ‘date +%d.%m.%Y‘, ein ‘date +%A‘.
  ```

  Es wird z.B. ausgegeben: „`Es ist der 07.11.1998, ein Samstag.`“

  ```
  ls -al > ‘tty‘
  ```

  Letzteres erzwingt die Ausgabe auf dem aktuellen Terminal, auch wenn der Befehl Teil eines Skripts oder einer Befehlskette ist, deren Ausgabe insgesamt umgelenkt wird:

  ```
  (pwd ; ls -al > ‘tty‘) > out.txt
  ```

- Mit der Umlenkung `<<` kann die Standard-Eingabe eines Befehls auch direkt aus einem Teil der Befehlszeile übernommen werden:

  ```
  grep x <<@
  Adam
  Andreas
  Axel
  @
  ```

  Man wählt eine Zeichenkette, die *nicht* im gewünschten Eingabetext vorkommt, als Markierung für Anfang und Ende (hier „`@`“). (Am Dateiende wird auch gestoppt, die Anfangsmarkierung ist dennoch notwendig.) Der Befehl `grep` durchsucht seine Eingabe nach „`x`“ und gibt passende Zeilen aus (siehe Seite 104). Die Ausgabe des obigen Kommandos ist also „`Axel`“.

- Die Notationen `<&-` und `>&-` *schließen* die Standard-Ein- bzw. -Ausgabe.

5.4 Hintergrundprozesse

Gibt man hinter dem Kommando das Zeichen '`&`' an, wartet die Shell nicht auf die Beendigung des neuen Prozesses, sondern ist direkt wieder bereit, eine neue Befehlszeile anzunehmen. Der neue Prozess läuft „im Hintergrund".

Die Shell numeriert die so von ihr gestarteten Prozesse (`[1]`, `[2]`, ...) – über diese Nummern kann man sich später auf die Prozesse beziehen.

Dieses Vorgehen eignet sich, wenn der Hintergrund-Prozess seine Ausgabe in eine Datei schreibt oder ein eigenes Fenster öffnet. Die Standard-Eingabe solcher Hintergrund-Prozesse wird standardmäßig auf `/dev/null` gesetzt (es wird immer `EOF`, end-of-file, gelesen), es sei denn, sie wird ohnehin schon durch ein '`<`' umgelenkt. Wenn man die Ausgabe nicht umlenkt, teilen sich Shell und Hintergrundprozess den Ausgabekanal.

Wenn man das '`&`' vergessen hat, kann man den (Vordergrund-) Prozess noch nachträglich in den Hintergrund schieben. Mit `CTRL-Z` hält man ihn vorübergehend an, und die Shell wird wieder aktiv. Mit dem Kommando „`bg`" (BackGround) werden alle so gestoppten Prozesse von der Shell abgekoppelt und laufen weiter – so, als wären sie direkt mit '`&`' gestartet worden.

Mit „`fg`" (ForeGround) und Angabe der Nummer (als Hintergrund-Prozess) oder des Anfangs ihrer Befehlszeile können Hintergrundprozesse wieder in den Vordergrund geholt werden.

5.5 Subshells

Man kann einen Befehl oder eine Befehlsfolge in Klammern einschließen. Zu ihrer Ausführung wird dann ein eigener Shell-Prozess gestartet.

Zu beachten ist z.B., dass Environment-Änderungen in der Subshell keine Auswirkungen außerhalb der Klammern haben:

```
pwd; ( cd / ) ; pwd
```

Die Ausgabe der beiden `pwd` wird immer identisch sein, da das `cd` nur das `PATH` des Subshell-Environments beeinflusst.

5.6 Das `exec`-Kommando

Das Shell-Kommando „`exec` *command*" ersetzt die Shell durch das angegebene Kommando, d.h. die Shell wird überschrieben.

Achtung: Gibt man also „`exec ls`" ein, wird das aktuelle Verzeichnis aufgelistet, und danach wird man *ausgeloggt*, oder das Terminal-Fenster wird *geschlossen*!

Wenn man kein Kommando hinter `exec` angibt, wird die Shell durch eine neue Instanz von sich selbst ersetzt. Man kann dies aber nutzen, die Ein-/Ausgabe der Shell selbst nachträglich umzulenken (sinnvoll am Anfang von Shell-Skripten):

```
exec < /dev/ttyp3
```

Nach diesem Befehl werden die Kommandos von einem anderen Terminal gelesen. Vorsicht: Diese und die dort laufende Shell streiten sich dann um die Eingabe.

Beispiel: Man kann von einer Shell aus eine andere starten, die dann wie jedes andere Programm auch als neuer Prozess ausgeführt wird – etwa durch Eingabe von „`bash`“ innerhalb einer `csh`.

So laufen aber beide Shells, wobei `csh` nur mit dem Warten auf `bash` beschäftigt ist. Mit „`exec bash`“ wird dagegen die `csh` durch die `bash` überschrieben.

5.7 Pipes

Eine Pipe (oder Pipeline) ist eine dateiähnliche Verbindung zwischen zwei Prozessen. Ein schreibender Kanal des einen Prozesses („Producer“) wird mit einem lesenden Kanal des anderen („Consumer“) verbunden.

Allgemeine Pipes werden später eingeführt. Eine einfache Form lässt sich aber sehr leicht mit Shell-Angaben anlegen. Sie ist dann aber immer auf die Standard-Ausgabe eines Prozesses und die Standard-Eingabe eines anderen beschränkt. In der Kommandozeile setzt man dazu das Zeichen '`|`' zwischen die Angabe zum schreibenden und die zum lesenden Prozess.

Beispiel: Das Kommando „`ls -l`“ listet das aktuelle Verzeichnis im Langformat. Wenn das Verzeichnis dabei länger ist als das Terminal Zeilen hat, rollen die ersten Einträge aus dem Sichtbereich. Folgendermaßen erhält man eine seitenweise Ausgabe:

```
ls -l | more
```

Das Kommando „`more` *filename*“ zeigt die Datei *filename* seitenweise an. Ohne Dateiangabe liest `more` von seiner Standard-Eingabe – die hier ihre Zeichen von der Standard-Ausgabe von `ls` erhält.

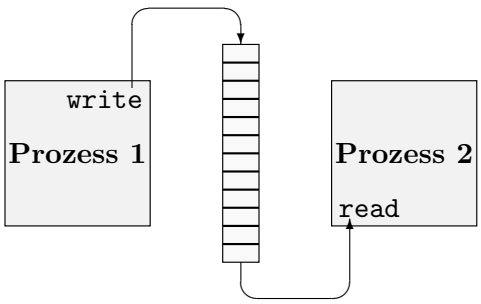
Der Effekt ist in etwa derselbe, als würde man explizit eine temporäre Datei anlegen:

```
ls -l > tempdat
more < tempdat
rm tempdat
```

Durch die Pipe bleibt es einem aber nicht nur erspart, selbst die temporäre Datei zu verwalten. Im letzten Beispiel laufen die beiden Prozesse zu `ls` und `more` *nacheinander* und die kompletten Daten lagern zwischendurch auf der Platte. Bei der Pipe von oben laufen beide Prozesse *gleichzeitig*. Die Pipeline arbeitet als **Warteschlange** (**FIFO**).

Wenn `more` versucht, Zeichen zu lesen, aber `ls` noch keine neuen Zeichen geliefert hat, wird `more` so lange *blockiert*, bis Zeichen vorliegen. Wenn `ls` Zeichen liefert, aber `more` ist noch beschäftigt und holt sie nicht ab, werden sie in einer Warteschlange zwischengespeichert.

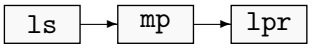
Wenn der Consumer nicht viel langsamer ist als der Producer, fallen also immer nur relativ wenig Daten an, die zwischengespeichert werden müssen. Selbst wenn man also Megabytegroße Datenmengen zwischen den Prozessen austauscht, wird immer nur wenig Plattenplatz benötigt.



Beachte: Es gibt auch in MS-DOS ein `more` und auch in `COMMAND.COM` die Möglichkeit, mit '`|`' Pipes anzulegen. Da MS-DOS aber kein Multitasking-System ist, laufen die beiden Prozesse immer *nacheinander* ab. Die Daten werden also immer zuerst *komplett* erzeugt, auf Platte zwischengelagert und dann verbraucht.

- Es können beliebig viele Pipes hintereinander in einem Shell-Kommando angegeben werden. Die Befehlszeile

  ```
  ls -l | mp | lpr
  ```

  wandelt beispielsweise die Ausgaben von `ls` in eine spezielle Form mit Titelbalken etc. um (`mp`=make pretty), bevor sie an das Druckerkommando `lpr` gehen und ausgedruckt werden.

- Der Status (Rückgabewert) einer ganzen Pipeline ist immer der Status des *letzten* vorkommenden Befehls.

- Die `csh` bietet die Möglichkeit, sowohl Standard-*Ausgabe* wie auch Standard-*Fehlerkanal* eines Befehls weiterzuleiten. Beide landen dann gemischt an der Eingabe des zweiten Befehls. Das entsprechende Shell-Symbol lautet dann „`|&`“.

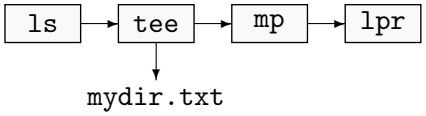
5.8 Filter

Unter UNIX ist man (wann immer es geht!) bemüht, Programme so zu schreiben wie das `mp` im Beispiel oben, also so, dass sie in Pipelines eingesetzt werden können. Die zu verarbeitenden Dateien sollten also nicht nur als Parameter angegeben werden können. Immer oder bei fehlenden Dateiangaben sollten die Programme von der Standard-Eingabe lesen bzw. auf die Standard-Ausgabe schreiben. Solche Programme bezeichnet man als **Filter**.

Wichtige UNIX-Kommandos, die als Filter arbeiten, sind folgende (genauere Informationen gibt's per `man`):

`tee:` Als Filter kopiert dieses Programm alle Zeichen unverändert von der Eingabe auf die Ausgabe. Gleichzeitig schreibt es sie aber in eine Datei, deren Name als Parameter angegeben wird:

```
ls -l | tee mydir.txt | mp | lpr
```



Hier wird die Ausgabe von `ls` also zusätzlich „abgezweigt“ und in die Datei `mydir.txt` geschrieben.

`pr:` „print“, druckt aber nicht wirklich, sondern bereitet die Datei nur für den Drucker auf. Beispielsweise kann man einen Text in mehrere Spalten formatieren:

```
ls | pr -4 | lp
```

Außerdem erhält jede so entstehende Seite einen Kopf mit Datum und Seitennummer. Das Kommando `lp` (Line Printer) schickt das Ergebnis auf den Standard-Textdrucker.

`sort:` Sortiert die Ausgabe zeilenweise, alphabetisch oder numerisch. Der Spaltenbereich, der für das Sortieren ausschlaggebend sein soll, kann zusätzlich angegeben werden.

```
ps | sort -rn
```

listet die laufenden Prozesse auf, sortiert aber in absteigender Reihenfolge (`-r`) numerisch (`-n`) nach Prozessnummer.

`tr:` „translate“, setzt Zeichencodes oder Zeichencode-Bereiche in andere um.

```
finger | tr a-z A-Z
```

gibt die aktuellen Benutzer aus, alle Kleinbuchstaben werden dabei in Großbuchstaben umgewandelt.

```
cat important.txt | tr "\012" " "
```

gibt die Datei `important.txt` aus, entfernt dabei aber alle Zeilenumbrüche (ersetzt sie durch Spaces). Der ASCII-Code von LineFeed ist $10=(12)_8$.

`grep:` Durchsucht Dateien oder die Standard-Eingabe nach Zeilen, die bestimmte Zeichenfolgen enthalten, und gibt diese Zeilen auf die Standard-Ausgabe aus. Die Zeichenfolgen werden durch **reguläre Ausdrücke** (**regular expressions**) beschrieben, einem mächtigen Mechanismus, der mit Wildcards (Jokerzeichen) bei Dateinamen vergleichbar ist. Der komplette Definitionsumfang (mit Klammerung, Oder, etc.) wird nur von der Version `egrep` unterstützt. (Der Name stammt übrigens vom `ed`-Kommando „`g/`*reg-exp*`/p`“,mit `g` für go, `p` für print.)

`grep` braucht nicht als Filter verwendet zu werden. Das erste Argument wird als regulärer Ausdruck interpretiert, der Rest als Dateinamen. Außerdem sind einfache Zeichenketten (ohne Spezialzeichen, s.u.) bereits reguläre Ausdrücke. Beispielsweise kann man also wie folgt alle `C`-Quelltexte im aktuellen Verzeichnis nach „`sqrt`“ durchsuchen lassen:

```
grep sqrt *.c
```

Die Ausgabe ist z.B.:

```
test.c: a=sqrt(x)
main.c:   printf("%f\n",sqrt(2.0));
```

Folgendes sind „atomare“ reguläre Ausdrücke im `grep`-Stil:

c	steht für die Zeichenkette *c* (*c*≠Spezialzeichen)
`.`	steht für genau ein beliebiges Zeichen
`^`	steht für den Anfang einer Zeile
`$`	steht für das Ende einer Zeile
`[`*charset*`]`	steht für genau ein Zeichen aus der Menge *charset*
`[^`*charset*`]`	steht für genau ein Zeichen, das *nicht* in *charset* vorkommt

charset ist wie bei Wildcards (Seite 99) zu verstehen. Man darf aber nicht übersehen, dass die letzte Definition `[^...]` bei den Wildcards fehlt – d.h. `[^a]` steht als Wildcard für ein Zeichen ‘`^`’ oder ein Zeichen ‘`a`’!

Wenn die Spezialzeichen `.^\$[]-` als echte Zeichen vorkommen sollen, muss ihnen ein Backslash `\` vorangestellt werden.

Größere reguläre Ausdrücke werden sukzessive aus kleineren zusammengesetzt. Wenn $\mathcal{R}_1$ und $\mathcal{R}_2$ reguläre Ausdrücke sind, sind es auch folgende:

$(\mathcal{R}_1)$	beschreibt denselben regulären Ausdruck wie $\mathcal{R}_1$ – die Klammern sind zur Gruppierung in komplexeren Ausdrücken gedacht
$\mathcal{R}_1\mathcal{R}_2$	beschreibt alle Zeichenketten, die durch *Hintereinanderschreiben* von zwei Zeichenketten entstehen, die durch $\mathcal{R}_1$ respektive $\mathcal{R}_2$ beschrieben werden
$\mathcal{R}_1\texttt{\|}\mathcal{R}_2$	beschreibt eine Zeichenkette, die durch $\mathcal{R}_1$ *oder* durch $\mathcal{R}_2$ beschrieben wird
$\mathcal{R}_1\texttt{*}$	beschreibt beliebig viele (eventuell 0) hintereinander geschriebene Zeichenketten, die alle jeweils durch $\mathcal{R}_1$ beschrieben werden
$\mathcal{R}_1\{n\}$	beschreibt n hintereinander geschriebene Zeichenketten, die alle jeweils durch $\mathcal{R}_1$ beschrieben werden

Das Jokerzeichen '`?`' entspricht also '`.`', und '`*`' entspricht „`.*`“! Die Mechanismen `|(){}` funktionieren nur mit `egrep`. In `grep` werden dafür die Klammern beispielsweise als Registerspeicher benutzt (siehe am besten den `man`-Eintrag von `ed`).

Die Zeile

```
ls | grep -v \.[ch]$
```

listet alle Dateinamen auf, die *nicht* auf `.c` oder `.h` enden.

```
ls -l | grep "^d.*"
```

listet alle Directories auf (Zeilen der `ls`-Ausgabe, die mit '`d`' beginnen).

Oben hatten wir nach „`sqrt`“ gesucht. Unser Aufruf findet dieses aber auch als Unterstring z.B. von „`anothersqrtfunction`“. Wir müssen also ausschließen, dass direkt vor oder hinter dem „`sqrt`“ Buchstaben, Ziffern oder Underscores stehen:

```
grep [^a-zA-Z0-9_]sqrt[^a-zA-Z0-9_] *.c
```

So darf allerdings `sqrt` nicht am Zeilenanfang oder -Ende stehen. Wenn wir die entsprechenden regulären Ausdrücke einbauen, erhalten wir folgendes:

```
egrep "(^|[^a-zA-Z0-9_])sqrt([^a-zA-Z0-9_]|$)"
```

sed: Stream EDitor. Er kann diverse komplexe Transformationen an Textströmen vornehmen. Seine Aktionen werden wiederum mit Hilfe von regulären Ausdrücken beschrieben. Er ist an der Zeileneditor `ed` angelehnt, der auch vom Editor `vi` aus verwendet wird. Entsprechend ist seine Notation sehr `vi`-ähnlich.

Beispielsweise ersetzt man im `vi` mit „`:%s/float/double/g`“ alle im Text vorkommenden Textstücke „`float`“ durch „`double`“ (`%`=im ganzen Text, `s`=substitute, `g`=general, also mehr als einmal pro Zeile). Entsprechend sieht das mit dem `sed` aus (`-e`=expression):

```
sed -e "s/float/double/g" program.c > program2.c
```

So kann man Zeilenumbrüche durch „<CR>“ besonders kenntlich machen:

```
ls | sed -e "s/$/<CR>/"
```

Beispiel: Hier erzeugt der `sed` ein Stück C-Quelltext. Es wird ein Array aus Strings definiert, das die Namen der Dateien im aktuellen Verzeichnis enthält:

```
ls -m | ( echo "char *names[]={" ; \
  sed -e "s/^/ \"/g" -e "s/$/\"/g" -e "s/, /\",\"/g" -e "s/\"\"$//g" ; \
  echo "};" )
```

Ausgabe ist z.B. folgende:

```
char *names[]={
"1.txt","2.txt","3.txt"
};
```

Das „`ls -m`“ erzeugt zunächst folgende Ausgabe:

```
1.txt, 2.txt, 3.txt
```

Der `sed` wendet darauf vier Ersetzungsoperationen an:

`s/^/ \"/g`	fügt Space/Anführungszeichen am Zeilenanfang ein
`s/$/\"/g`	hängt Anführungszeichen ans Zeilenende an
`s/, /\",\"/g`	ersetzt Komma durch `","`
`s/\"\"$//g`	löscht zwei Anführungszeichen am Zeilenende

5.9 system

Es gibt eine ANSI-C-Bibliotheksroutine `system` (definiert in `stdlib.h`), mit der man aus eigenen Programmen leicht die Standard-Shell verwenden kann, um Programme komfortabel aufzurufen (auch unter DOS):

ANSI-C

```
int system(const char *string);
```

interpretiert mit Hilfe der Standard-Shell `string` als Befehlszeile und führt sie aus

Der übergebene Parameter ist einfach eine komplette Befehlszeile, wie sie am Terminal eingegeben werden kann, mit allen Services wie ‘`<`’, ‘`>`’, ‘`|`’ und ‘`&`’. Die Funktion wartet auf die Beendigung des Kommandos und liefert seinen `exit`-Code zurück. Ein möglicher Aufruf ist:

```
system("ps -al > proclist.txt");
```

Unter UNIX wird die für den Benutzer eingestellte Standard-Shell verwendet, unter MS-DOS die in der `CONFIG.SYS`-Zeile „`SHELL=...`“ festgelegte.

Die Funktion ist zwar einfach zu handhaben; man muss aber beachten, dass nicht nur das eigentliche Programm geladen und gestartet wird, sondern zwischendurch auch die Shell!

5.10 Shell-Skripte

Die Shell arbeitet nicht nur interaktiv zeilenweise, sondern kann auch zur Interpretation ganzer Programme eingesetzt werden, deren Bestandteile Shell-Kommandos sind. Die einfachste Form solcher „Shell-Skripte“ ist eine Folge von Befehlszeilen, als wären sie so in der Shell nacheinander angegeben worden.

- Die Shell stellt aber zusätzlich eigene Variablen und Kontrollstrukturen wie `if`, `while` und `case` zur Verfügung – und in deren Behandlung unterscheiden sich die verschiedenen Shells oft deutlich.

- In sehr vielen Fällen ist es – auch beim Programmieren von System-Kommandos – gar nicht notwendig, eine höhere Programmiersprache wie `C` zu bemühen. Durch die einfachen Mechanismen zur Kombination mehrerer Programme (I/O-Umlenkung, Pipelines, Argument-Einsetzung, etc.) zu einem Befehl sind Shell-Skripte oft schneller geschrieben, übersichtlicher und wartungsfreundlicher.

- Man kann viele Bücher über die effiziente Programmierung von Shell-Skripten füllen. Wir beschränken uns in unserem Rahmen aber auf die notwendigsten Dinge. Die meisten Angaben beziehen sich auf *alle* Shells; einige Spezialitäten nur *einzelner* Shells werden außer acht gelassen.

- Physisch ist ein Shell-Skript einfach eine ASCII-Datei. Wenn ihr Zugriffsflag `x` (execute) gesetzt ist, kann sie – wie ausführbare Programme im Maschinenformat – durch Angabe ihres Namens in der Shell ausgeführt werden. Ansonsten kann man sie mit „`sh` *skript*“ starten.

- In der ersten Zeile kann man angeben, mit welcher Shell das Skript ausgeführt werden soll, beispielsweise

  ```
  #!/opt/gnu/bin/bash
  ```

  Der volle Pfad ist erforderlich. Ohne diese Angabe wird die Standard-Shell des Benutzers verwendet. Wenn man aber beispielsweise die spezielle Syntax der `csh` verwendet, die zu den anderen Shells inkompatibel ist, sollte man diese Angabe nicht vergessen. Wir werden sie sonst im Skript zur Abkürzung oft weglassen.

- Die meisten Shells führen bei ihrem Start ein spezielles Shell-Skript aus (*run commands*), das Kommandos zur Initialisierung enthält. Im `HOME`-Directory des Benutzers sucht die `bash` nach `.bashrc`, die `csh` nach `.cshrc`, etc.

  Sinnvolle Aktionen in den Run-Commands sind z.B.:

  - das Setzen von Umgebungsvariablen, nur für diese Shell,
  - das Setzen von Alias-Namen für Kommandos (z.B. „`alias ll="ls -l"`“).

  Wenn die Shell als Login-Shell benutzt wird (meist durch die Option `-l`, z.B. von einem `getty` aus), wird außerdem zuvor noch ein Login-Skript ausgeführt, das je nach System `.profile` oder `.login` heißt und ebenfalls im Home-Directory liegt. Wenn es kein solches gibt, wird ein systemweites Standard-Skript wie `/etc/profile` verwendet.

Sinnvolle Aktionen in `profile` sind z.B.:

- das Setzen der `umask` (wie „`umask 022`“)
- das Setzen von `$PATH` für alle Shells (Suchpfad für Kommandos)
- das Setzen von Umgebungsvariablen für bestimmte Programme (für alle Shells gültig)

5.10.1 Eigene Variablen

Die Shell arbeitet immer nur mit einem Typ von Variablen, nämlich mit Zeichenketten (Strings). Entsprechende Befehle können diese aber natürlich auch als numerische Werte interpretieren.

- Solche Variablen legt man einfach durch ihre (erste) Belegung mit einem Wert an, beispielsweise

  ```
  myvar=hello
  ```

  Achtung: Die `csh` erlaubt solche Zuweisungen nur in Skripten, nicht in der interaktiven Eingabe. Bei ihr muss man „`set myvar=hello`“ schreiben, was wiederum bei den anderen Shells nicht funktioniert.

- In den Variablennamen sind Buchstaben, Underscores und (außer an der ersten Stelle) Ziffern erlaubt. Zwischen großen und kleinen Buchstaben wird unterschieden.

- Wenn in späteren Shell-Befehlen der aktuelle *Wert* dieser Variablen *eingesetzt* werden soll, muss man ein Dollarzeichen voranstellen:

  ```
  echo $myvar, world!
  ```

  Die Ausgabe ist „`hello, world!`“.

- Eine Variablen-Verwendung kann auch die erste Angabe einer Befehlszeile sein. Der Inhalt der Variablen wird dann als Name des auszuführenden Befehls interpretiert:

  ```
  myvar="echo hello"
  $myvar world!
  ```

- Wenn man Variable verwendet, die vorher nicht belegt worden sind, ist das kein Fehler, sondern die Variablen werden dann angelegt und sind *leer*:

  ```
  echo myvar2=$myvar2
  ```

  liefert in diesem Fall die Ausgabe „`myvar2=`“.

- Mit `read` kann man Werte für Variable von der Standard-Eingabe einlesen:

  ```
  read var1 var2 var3
  ```

  Hier werden die *Worte* einer Eingabezeile nacheinander in die drei Variablen gefüllt. Die letzte Variable erhält immer den Rest der Eingabzeile, also eventuell mehr als ein Wort.

- Mit `unset` kann man eine Variable löschen.

5.10.2 Array-Variablen

Allein die `bash` stellt zusätzlich **Arrays** von Variablen zur Verfügung. Elemente werden mit ganzen Zahlen ≥ 0 indiziert und in `C`-Schreibweise (also wie *arrname*`[`*index*`]`) angesprochen.

- Ein Array entsteht beim ersten Ansprechen eines Elements, also

  ```
  arr[4]=four
  index=4
  echo ${arr[$index]}
  ```

- Nicht gesetzte Array-Elemente sind wie undefinierte Variable *leer*.

- Das Beschreiben eines ganzen Arrays geschieht mit runden Klammern:

  ```
  array=(zero one two three four)
  ```

  Zum lesenden Ansprechen eines ganzen Arrays (aller belegten Elemente) verwendet man als Index '`*`':

  ```
  echo ${array[*]}
  ```

  Damit ist folgende Konstruktion möglich (zu `for` weiter unten):

  ```
  for element in ${array[*]} ...
  ```

- Arrays können nicht exportiert (s.u.) werden. Sie können mit **`unset`** gelöscht werden.

5.10.3 Exportieren von Variablen

Die in einem Shell-Skript neu definierten Variablen sind weder nach „außen“ sichtbar (in einem Skript, das dieses Skript aufgerufen hat, hinter dem Aufruf), noch nach „innen“ (in Shell-Skripten, die dieses Skript aufruft).

```
# Skript1:            # Skript2:
MYVAR=‘date‘          echo $MYVAR
Skript2
```

Diese Skripte-Kombination gibt nach dem Aufruf `Skript2` *nichts* aus, da die Variable `MYVAR` in `Skript2` leer ist.

Variable können nie in Richtung Elter-Prozess bekanntgemacht werden. Sollen sie aber an Kinder weitergegeben werden, können sie „exportiert“ werden. (Sie werden dann in das Environment aufgenommen, das an Kind-Prozesse vererbt wird.)

In den meisten Shells geschieht das mit dem Kommando „`export` *varname*“ (bei der `csh` muss man mit `setenv` arbeiten). Ein geändertes `Skript1` sieht so aus:

```
# Skript1:
MYVAR=‘date‘
export MYVAR
Skript2
```

Bei der `bash` kann man das mit einer Zuweisung verbinden:

```
export MYVAR=‘date‘
```

Das Exportieren geschieht in der `csh` mit `setenv`:

```
setenv MYVAR ‘date‘
```

5.10.4 Aufrufparameter

Genau wie bei ausführbaren Programmen kann man auch an Shell-Skripte Parameter über ihre Aufrufzeile übergeben. Sie werden immer als Zeichenketten (nie numerisch) interpretiert und entsprechen den Bestandteilen der Aufrufzeile, die durch Spaces oder Tabulatoren voneinander abgegrenzt werden. (Zur Syntax bei Optionen mit ‘`-`’ siehe weiter unten.)

Innerhalb des Skripts werden die Parameter mit ihrer *Nummer* angesprochen: `$0` ist der Name des Programms, `$1` der erste angegebene Parameter, etc. – daher heißen diese Variablen auch **positionale Parameter**. Die Anzahl der Argumente steht in `$#`. Alle positionalen Parameter – als *ein* String verkettet – lassen sich als `$*` ansprechen.

Beispiel: Das folgende kurze Skript „`zgv`“ ermöglicht das Anschauen eines PostScript-Files, das aus Platzgründen aber in gepackter Form (`*.gz`) auf der Platte vorliegt. Ein Aufruf wäre z.B. „`zgv chap1.ps.gz`“.

```
#!/bin/sh
gunzip -c $1 | ghostview -
```

Hinter dem ‘`$`’ ist nur *eine Ziffer* erlaubt. Das Skript kann also maximal *neun* Argumente ansprechen. Genauer allerdings sieht es immer ein „Fenster“ der Parameterliste, das 9 Parameter breit ist. Mit dem Shell-Kommando **`shift`** kann man dieses Fenster nach rechts verschieben und so doch alle Argumente erreichen.

Beispiel: Folgendes Skript gibt einfach alle seine Argumente aus, und zwar immer in Neuner-Gruppen zeilenweise:

```
while test $# -ne 0
do
   echo $1 $2 $3 $4 $5 $6 $7 $8 $9
   if test $# -gt 9; then shift 9; else shift $#; fi
done
```

Offensichtlich steht `$#` für die Anzahl der Variablen *ab Fensterbeginn*. Sie verkleinert sich während dieser Schleife. Analog ist `$*` nach einem `shift` zu interpretieren.

Mit dem Kommando `set` kann man die Parameter-Variablen im Skript verändern:

```
set Abracadabra dreimal schwarzer Kater
```

Danach ist `$1`=`Abracadabra`, `$2`=`dreimal`, etc. Man kann hier geschickt die Ausgabe eines anderen Kommandos einbauen:

```
set ‘date +"%d %m %Y"‘
```

Danach enthält `$1` den aktuellen Tag, `$2` den Monat und `$3` das Jahr.

```
set - `ls *.c`
```

Es findet Expandierung statt, d.h. hier werden die Variablen ab `$1` mit den Namen aller `C`-Quellcodes im aktuellen Verzeichnis gefüllt. Das '`-`' steht hier, um Komplikationen zu vermeiden, wenn der erste Dateiname mit '`-`' beginnt – er würde dann als Option von `set` interpretiert! Ansonsten wird unser '`-`' von `set` ignoriert.

Die verwandten Variablen `$$`, `$!` und `$?` werden später im Kapitel über Prozesse behandelt.

5.10.5 Umgebungsvariablen

Die Shell setzt außerdem die Einträge aus dem Environment des Prozesses in echte Shell-Variable um. Dementsprechend gibt es Variable `$HOME`, `$PATH`, `$DISPLAY`, etc., die man wie eigene Variable lesen und schreiben kann.

```
cat mydoc.ps | $PRINTER
DISPLAY=wmpi03:0.0
```

Wenn die Variablen geändert werden, hat das aber zunächst keine Auswirkung auf das echte UNIX-Environment (insbesondere auf Prozesse, die nicht von dieser Shell aus gestartet werden)! Dazu müssen sie exportiert werden.

```
export DISPLAY=wmpi03:0.0
```

Zwei Umgebungsvariablen, die direkt mit der Shell (`sh` und Kompatible) zusammenhängen, sind folgende:

`PS1`	primärer Prompt-String, also Ausgabe der Shell bei einer Kommando-Aufforderung (z.B. „$“)
`PS2`	sekundärer Prompt-String, der ausgegeben wird, wenn eine Fortsetzungszeile angefordert wird

Man kann diverse Steuerzeichen in die Prompt-Strings einbauen (am besten in der `man`-Seite der Shell nachschauen), bei der `bash` z.B.:

`\d`	Datum	`\u`	Username
`\H`	Hostname	`\w`	Working Directory
`\t`	Zeit	`\#`	Kommando-Nummer

In der `csh` heißt die erste Variable `prompt`; es gibt keine Fortsetzungszeilen.

Die `bash` hat einige zusätzliche nützliche Variable, u.a.:

`OLDPWD`	aktuelles Verzeichnis *vor* dem letzten `cd`
`PPID`	Nummer des Elter-Prozesses der `bash`
`RANDOM`	liefert eine Zufallszahl zwischen 0 und 32767 (eine Zuweisung an `RANDOM` setzt einen neuen Startwert)
`SECONDS`	Sekunden seit dem Start der `bash` (darf durch eine Zuweisung zurückgesetzt werden)
`TMOUT`	Anzahl von Sekunden, die der Benutzer inaktiv sein darf, bevor sich die `bash` selbsttätig beendet

5.10.6 Variablen-Substitutionen

Die simpelste Substitution in der Shell ist die, dass für „`$`*var*“ immer der aktuelle *Inhalt* der Variablen *var* eingesetzt wird. Durch geschweifte Klammern und zusätzliche Angaben kann man dieses Verhalten modifizieren:

`${`*var*`}`	verhält sich wie `$`*var*
`${`*var*`-`*text*`}`	falls *var* leer sein sollte (z.B. noch nicht definiert wurde), wird der Text *text* eingesetzt
`${`*var*`=`*text*`}`	falls `var` leer war, erhält es den Wert *text* vor der Verwendung
`${`*var*`:?`*error*`}`	falls `var` leer ist, wird die Shell mit der Fehlermeldung *error* abgebrochen. Wenn auch *error* leer ist, erscheint stattdessen eine Standardmeldung („`parameter null or not set`“ o.ä.).

5.10.7 Listen und Gruppierungen

Ein Shell-Skript ist genauer per Definition eine sequentielle Aufeinanderfolge von sogenannten „Listen“.

Eine **Liste** ist ein einzelnes Kommando oder eine Kommandofolge, die mit ‘`;`’ oder ‘`|`’ gebildet wird. Bei ‘`;`’ werden die Befehle unabhängig voneinander nacheinander ausgeführt, bei ‘`|`’ dagegen parallel mit einer Pipeline-Verbindung.

Listen müssen entweder mit dem Zeilenende oder mit einem ‘`;`’ enden (wenn sie in größere Strukturen eingebaut werden). Der Status der gesamten Liste ist jeweils der Status des zuletzt ausgeführten Kommandos.

Für eine Liste *list* ist die „**Gruppierung**“ `(list)` wieder eine Liste. Die enthaltenen Kommandos werden aber in einer eigenen Subshell ausgeführt. Änderungen an Shell-Variablen (z.B. `$PWD`) haben also keine Wirkungen nach außerhalb der Klammern:

```
pwd; (cd /usr/bin; pwd; ); pwd
```

Das Wechseln des Verzeichnisses geschieht nur in der Subshell. Hinter den Klammern ist man wieder im selben Verzeichnis wie zuvor.

Wenn man statt runder Klammern geschweifte setzt, werden die Kommandos nur gruppiert, ohne eine Subshell zu starten:

```
pwd ; { cd /usr/bin ; pwd; } ; pwd
```

Hiernach ist man unter Garantie in `/usr/bin`.

5.10.8 Fallunterscheidungen mit `if`

Die `csh` hält sich bei `if` eher an die `C`-Syntax und fällt hier ein wenig aus dem Rahmen. Bei den anderen Shells sieht ein `if`-Statement wie folgt aus:

if *list* **then** *list* (**elif** *list* **then** *list*)* [**else** *list*] **fi**

Der *-Teil kann wie üblich beliebig oft wiederholt werden, der []-Teil ist optional. Die Bedeutung der Konstruktion ist analog zu höheren Programmiersprachen zu verstehen, wobei **elif** der Kombination **else**/**if** entspricht.

Listen werden hier als Bedingungen für Verzweigungen aufgefasst, und zwar über ihren Status (Rückgabewert). Achtung: Ein Wert von 0 entspricht dabei **true** (kein Fehler aufgetreten), ein anderer Wert **false**!

Beispielsweise sind folgende einfache Konstruktionen möglich:

if $list_1$ **then** $list_2$ **fi**
if $list_1$ **then** $list_2$ **else** $list_3$ **fi**
if $list_1$ **then** $list_2$ **elif** $list_3$ **then** $list_4$ **fi**
if $list_1$ **then** $list_2$ **elif** $list_3$ **then** $list_4$ **else** $list_5$ **fi**

Diese Konstruktionen sind wieder Listen und haben als Status den Status der zuletzt ausgeführten Liste.

Beispiel: Sehr oft reagiert man mit **if** auf Erfolgs- (0) bzw. Misserfolgsmeldungen ($\neq 0$) von Programmen, beispielsweise:

```
if `gcc $1.c -o $1`
then
   echo Compilation erfolgreich
else
   echo Fehler bei der Compilation
fi
```

Es gibt Dummy-Befehle `true` und `false`, die lediglich den Wert 0 bzw. 1 zurückliefern und in Bedingungen verwendet werden können.

Beispiele:

```
if true; then echo "alles klar"; else echo "sehr seltsam"; fi
```

Man beachte, dass alle die Listen entweder an einem Zeilenende oder mit einem ‘;’ enden müssen! Äquivalent sähe die obige Zeile, in mehrere aufgeteilt, wie folgt aus:

```
if true
then
  echo "alles klar"
else
  echo "sehr seltsam"
fi
```

5.10.9 Listen mit `&&` und `||`

Man hat zusätzlich die Möglichkeit, kurze Fallunterscheidung zu bilden – in Form von Listen mit „`&&`“ bzw. „`||`“:

$command_1$ `&&` $command_2$

Das ist ein Analogon zur Kurzschlussauswertung des logischen Und in `C`, d.h. es ist eine abkürzende Schreibweise für Folgendes:

if *command*$_1$; **then** *command*$_2$; **fi**

Das Gegenstück dazu ist folgendes:

command$_1$ || *command*$_2$

Vorsicht: Das ist kein *direktes* Analogon zum || von C. command$_2$ wird genau dann ausgeführt, wenn command$_1$ einen Rückgabewert **ungleich** 0 hat, also ein Fehler aufgetreten ist.

Beispiel: Mit folgendem Skript kann man ghostview eine PostScript-Datei anzeigen lassen, die gepackt auf der Platte abgelegt ist:

```
< $1 && (gunzip -c $1 | ghostview - &)
```

Der erste Teil testet nur, ob die anzuzeigende Datei überhaupt existiert (sonst gibt er eine Fehlermeldung aus). Durch das „&&“ wird nur dann die Subshell in Klammern überhaupt ausgeführt. Ansonsten würde unerwünschterweise ghostview gestartet, das vergeblich auf Daten von der Eingabe warten würde (leeres Fenster)!

Der Nachteil ist, dass man so im Dokument nicht zurückblättern kann (da die Daten in der Pipeline nur einmal gelesen werden können). Alternativ kann man wie folgt mit einer temporären Datei arbeiten:

```
test $# -eq 1 && ( \
   tempfile=‘mktemp baba.XXXXXX‘ ; \
   test $? -eq 0 && ( ( gunzip -c $1 > $tempfile && ghostview $tempfile) ; \
   rm -R $tempfile ) )
```

Nur wenn mit mktemp eine Temporärdatei angelegt werden konnte ($? liefert den Return-Code des echten Kommandos, also den von mktemp), werden gunzip und ghostview ausgeführt (und die Temporärdatei wieder entsorgt).

5.10.10 Bedingungen

Es ist natürlich sehr wichtig zu wissen, wie man überhaupt sinnvolle Bedingungen konstruieren kann, die in **if** etc. verwendet werden können.

Bei der bash kann man eine Liste durch ((*expression*)) erzeugen. Der *expression*-Teil wird wie ein arithmetischer Ausdruck ausgewertet. Der Status der Liste ist 0, wenn der Ausdruck *nicht* 0 ist, sonst 1. Das entspricht in etwa der C-Semantik von Bedingungen:

```
if ((1+2)); then echo "ungleich 0"; else echo "gleich 0"; fi
```

Bei *allen* Shells kann man den UNIX-Befehl test einsetzen (daher sollte man eigene Programme oder Skripte unter UNIX bekanntlicherweise niemals „test“ nennen). Es kann diverse arithmetische, String- und Filesystem-bezogene Bedingungen testen. Ein Aufruf lautet immer „test *expr*“, es wird immer 0 für wahr und 1 für falsch zurückgeliefert.

Unten folgt eine Auflistung der wichtigsten Möglichkeiten für den Ausdruck *expr*.

Einige wichtige File-Tests:

-d *file*	gibt es *file*, und ist es ein Directory?
-f *file*	gibt es *file*, und ist es ein normales File?
-e *file*	gibt es *file* überhaupt? (nicht auf allen Systemen)
-r *file*	gibt es *file*, und ist es lesbar?

`-w` *file*	gibt es *file*, und ist es beschreibbar?
`-x` *file*	gibt es *file*, und ist es ausführbar?
file$_1$ `-nt` *file*2	ist *file*$_1$ neuer als *file*$_2$ (Newer-Than)?
file$_1$ `-ot` *file*2	ist *file*$_1$ älter als *file*$_2$ (Older-Than)?

String-Tests:

`-z` *string*	Test auf leeren String (Zero-Length)
`-n` *string*	Test auf nicht-leeren String (Non-Zero-Length)
string	wie `-n`
string$_1$ `=` *string*$_2$	Test auf String-Gleichheit
string$_1$ `!=` *string*$_2$	Test auf String-Ungleichheit

Logische Verknüpfungen:

`!` *expr*	logische Negation von *expr*
expr$_1$ `-a` *expr*$_2$	logische Und-Verknüpfung zweier Ausdrücke
expr$_1$ `-o` *expr*$_2$	logische Oder-Verknüpfung zweier Ausdrücke

Arithmetische Vergleiche:

arg$_1$ ∘ *arg*$_2$

Die Ausdrücke *arg*$_1$ und *arg*$_2$ werden als ganze Zahlen interpretiert und mit einem binären Vergleichsoperator ∘ verglichen.

Für ein *arg* kann insbesondere die Konstruktion „`-l` *string*“ verwendet werden (Stringlänge).

∘	Vergleich
`-eq`	$=$
`-ne`	$\neq$
`-lt`	$<$
`-le`	$\leq$
`-gt`	$>$
`-ge`	$\geq$

Beispiel: Das folgende Skript gibt eine Kurzinformation über den Typ der Datei, deren Namen als Parameter übergeben wird:

```
if test $# -ge 1 -a -e $1 ; then
   echo -n $1":        "
   if test -d $1 ; then echo directory
   elif test -b $1 -o -c $1 ; then echo special file
   elif test -L $1 ; then echo symbolic link
   elif test -p $1 ; then echo named pipe
   elif test -f $1 ; then echo regular file
   else echo unknown
   fi
fi
```

5.10.11 Mehrfach-Fallunterscheidung

Außer mit mehreren `else` und `elif` kann man dies mit einer Konstruktion realisieren, die **`switch`**/**`case`** in `C` bzw. **`case`**...**`of`** in `PASCAL` ähnelt:

`case` *word* **`in`** (*pattern*(|*pattern*)* **`)`** *list* `;;`)* **`esac`**

Es wird nacheinander eine Übereinstimmung des Worts *word* mit einem der angegebenen Muster *pattern* gesucht. Die Muster sind reguläre Ausdrücke im Dateinamen-Stil. Das *list* hinter der ersten Übereinstimmung wird ausgeführt. Wenn man einen Default-Fall einbauen möchte (keine Übereinstimmung mit den anderen Fällen), verwendet man als *letzten* Fall einfach `*`.

Beispiele: Folgendes Skript gibt Informationen über die Anzahl übergebener Parameter aus:

```
case $# in
   0) echo kein Parameter angegeben ;;
   1) echo ein Parameter angegeben ;;
   *) echo mehr als ein Parameter angegeben ;;
esac
```

Das nächste Skript wertet seine Parameter als Optionen und Argumente aus. Als Optionen sind „`-a`“, „`-e`“, „`-i`“, „`-o`“ und „`-u`“ erlaubt, als Argumente `C-` oder `C++`-Quelltexte:

```
for arg do
   case $arg in
      -[aeiou])   echo Option $arg entdeckt ;;
      -*)         echo unbekannte Option: $arg ;;
      *.c|*.cpp)  echo Quelltext $arg gefunden ;;
      *)          echo mit dem Argument $arg kann ich nichts anfangen! ;;
   esac
done
```

5.10.12 Schleifen

Es gibt mehrere Schleifenformen. Die einfachste davon ist die mit **`while`**:

`while` $list_1$ **`do`** $list_2$ **`done`**

Es wird so lange $\texttt{list}_2$ ausgeführt, wie $\texttt{list}_1$ einen Okay-Status (also 0) liefert. Die Schleife ist abweisend, es wird also als erstes einmal $\texttt{list}_1$ ausgeführt. Der Status der Konstruktion ist der Status von $\texttt{list}_2$ im letzten Durchlauf oder 0, falls die Schleife gar nicht durchlaufen wird.

Beispiel: Folgende Schleife läuft ewig:

```
while true
do
   date
   sleep 10
done
```

Die **`until`**-Schleife ist die Umkehrung der **`while`**-Schleife:

`until` $list_1$ **`do`** $list_2$ **`done`**

Diese Schleife läuft, *bis* $\texttt{list}_1$ okay (0) liefert (d.h. *solange* $\texttt{list}_1$ „nicht-okay“ liefert).

Eine ganz andere Schleifen-Konstruktion ist folgende:

`for` *name* [**`in`** *wordlist*] **`do`** *list* **`done`**

wordlist ist eine Liste von Worten, die mit Space voneinander getrennt sind, und die von der Shell wie Argumente erweitert werden (also beispielsweise „`*.cpp *.c`“). Es entsteht eine möglicherweise längere Wortliste als Ergebnis.

Für jedes Wort aus der erweiterten Wortliste wird der Schleifenrumpf *list* durchlaufen, wobei die Shell-Variable *name* so lange als Wert dieses Wort hat.

Wenn man die **in**-Angabe auslässt, durchläuft *name* automatisch die positionalen Parameter.

Beispiel:

```
for filename do >$filename; done
```

Dieses Skript legt leere Dateien an, deren Namen in der Aufrufzeile angegeben werden. (Vorsicht: Die `csh` erlaubt keine leeren Befehle wie hier zwischen **do** und **>**!)

Beispiel: Das folgende Skript listet das aktuelle Verzeichnis rekursiv auf. Inhalte von Unterverzeichnissen werden um vier Stellen nach rechts eingerückt.

Das Skript ruft sich dazu selbst rekursiv auf. Da es in die Unterverzeichnisse hineinwechselt, übergibt es sich selbst als ersten Parameter seine eigene Position. Der dritte Parameter ist der String, der zum Einrücken benutzt wird (hier: vier Spaces).

```
if test $# -lt 1 ; then self=‘pwd‘/$0 ; fi
if test $# -ge 2 ; then cd $2 ; fi

for file in * ; do
  echo -n "$3$spc$file"
  if test -d $file
  then
     echo " (dir)"
     "$self" "$self" "$file" "$3    "
  else
     echo
  fi
done
```

Aus allen Schleifen kann man – in Anlehnung an `C` – mit **break** ganz herausspringen, bzw. mit **continue** der *nächsten* Schleifendurchgang erzwingen.

5.10.13 Funktionen

Bei der `bash` kann man sogar Unterprogramme in einem Skript definieren:

function *name* `()` { *list* }

Das Wort „`function`“ darf weggelassen werden. Der Aufruf erfolgt wie bei normalen Kommandos über den Namen der Funktion (dazu muss die Definition vorher im Skript stehen). Es können beim Aufruf Argumente angegeben werden, die in der Funktion als die positionalen Parameter `$1` bis `$9` erreichbar sind.

Es ist auch ein rekursiver Aufruf erlaubt. Lokale Variable werden mit „`local`“ (bei der ersten Verwendung) gekennzeichnet. Bei Rekursion existieren die Parameter und lokalen Variablen in mehreren Instanzen.

5.11 Optionen und Argumente

Programmen, wie auch Shell-Skripten kann man ja bereits durch die Aufrufzeile Parameter mitgeben. Die Zeile wird durch Spaces oder Tabulatoren in Bestandteile zerlegt. Wenn ein sol-

cher Bestandteil mit einem '`-`' (unter MS-DOS mit einem '`/`') beginnt, nennt man ihn **Option**, sonst **Argument**.

Wie oben beschrieben, werden die Bestandteile der Aufrufzeile von Shell-Skripten aus über die Variablen `$1` bis `$9` (ggf. mit `shift`) angesprochen. Der Mechanismus in `C` und `C++` zur Übergabe der Optionen und Argumente an ein Programm ist systemunabhängig (siehe 3.3).

5.11.1 Format der Optionen

Es gibt, weder unter MS-DOS, noch unter UNIX, ein vorgeschriebenes Format für die Optionen und deren Positionierung. Üblicherweise sieht der Aufruf eines Programms wie folgt aus:

name option cmdarg**

Das * steht für eine beliebig häufige Wiederholung, eventuell auch 0-mal. Es ist

name: der Name des auszuführenden Programms, evtl. inklusive Pfad.

option: eine Steuerungs-Angabe an das Programm, eingeleitet mit '`-`' unter UNIX bzw. mit '`/`' unter MS-DOS.
Optionen *modifizieren* das Verhalten des Kommandos und sind keine für die Ausführung essentiellen Angaben. Die *Reihenfolge* der Optionen in der Aufrufzeile sollte *keine* Rolle spielen.
Beispiel: „`ls -l`“, dabei schaltet das „`-l`“ das Langformat ein.

cmdarg: ein Argument des Aufrufs.
Argumente beschreiben meist Daten (Dateinamen), auf denen das Kommando *operieren* soll. Hier spielt die *Reihenfolge* sehr wohl eine Rolle – mehrere Argumente werden anhand ihrer *Position* unterschieden.
Beispiel: „`cp from to`“, Position 1: die Quelle des Kopiervorgangs, Position 2: das Ziel.

Ein Beispiel mit Optionen *und* Argumenten ist „`cp -f from to`“. Das '`f`' steht für `force` und erlaubt das Überschreiben der Datei `to`, wenn es sie bereits gibt.

Das genaue Format und die Semantik der Optionen ist nicht festgelegt. Man sollte sich aber an die unten aufgeführten Konventionen halten, damit beispielsweise das Utility `getopt` funktioniert, das einem bei der Auswertung (von Shell-Skripten aus) viel Arbeit abnimmt:

option:
`-`*noargletter noargletter**
`-`*argletter optarg*(`,`*optarg*)*

noargletter: ein einzelnes Zeichen; für Optionen ohne zusätzliche Angaben.
Beispiel: „`ls -l -a`“.
Mehrere Zeichen können hinter einem einzigen '`-`' zusammengefasst werden.
Beispiel: „`ls -al`“.

argletter: ein einzelnes Zeichen; für Optionen mit zusätzlichen Angaben.
Beispiel: `gcc hello.c -o hello`, hier ist *optarg*=„`hello`“.
Es können nicht mehrere Optionen zusammengefasst, mehrere Options-Argumente dagegen mit Kommas getrennt angegeben werden.

cmdarg:
: irgendeine Angabe, die *nicht* mit '-' beginnt, beispielsweise der Pfad einer Datei. Es gibt eine Ausnahme: „-“ allein steht für die Standard-Eingabe als Argument.

Die Optionen müssen meist nicht unbedingt vor den Argumenten stehen, sollten es aber, damit `getopts` funktioniert. Optional kann man mit „`--`“ das Ende der Options-Angaben markieren.

Manche Programme haben auch Klartext-Optionen mit mehr als einem Buchstaben, z.B. startet „`bash -noprofile`“ die Bash ohne Lesen der Konfigurationsdateien. Diese Optionen müssen dann meist *vor* den einfachen Optionen stehen. Viele GNU-Programme sind daher zu Klartext-Optionen mit *zwei* Minuszeichen übergegangen, z.B. „`gcc --version`“.

5.11.2 Auswertung in `C`

In `C` und `C++` muss man die Auswertung der Argumente selbst vornehmen. Nur in bestimmten Zusammenhängen (z.B. beim `X`-Toolkit) gibt es Hilfsfunktionen.

Beispiel: Folgendes `C++`-Programm hat die möglichen argumentlosen Optionen `a` bis `z`, die Optionen `A` bis `Z` mit Argumenten, außerdem beliebig viele Argumente. Der große Programmblock wertet alle Parameter aus. Der Schluss ist nur zum Testen gedacht und gibt die gefundenen Ergebnisse aus.

```
#include <iostream.h>
#include <ctype.h>
#include <vector.h>

int main(int argc, char *argv[])
 {
   bool noargopts[26];
   char *argopts[26];
   vector<char *> args;

   for (int i=0;i<26;++i) { noargopts[i]=false; argopts[i]=0; }

   for (int i=1;i<argc;++i)
    {
      char *P=argv[i],c;
      if ( P[0]!='-' || P[1]==0 )
         args.push_back(P);
      else
         while ((c=*++P)!=0)
            if (islower(c))
               noargopts[c-'a']=true;
            else if (isupper(c))
             {
               if (argopts[c-'A']!=0)
                  cerr << "Warning: option " << c << " used twice\n";
               else if (++i==argc)
                  cerr << "Warning: option " << c <<" needs an argument\n";
               else argopts[c-'A']=argv[i];
```

```
            }
           else cerr << "Unknown option: " << c << endl;
    }

   cout << "argumentless options: ";
   for (int i=0;i<26;++i)
      if (noargopts[i]) cout << (char)(i+'a');
   cout << endl;
   cout << "options with arguments:";
   for (int i=0;i<26;++i)
      if (argopts[i]!=0)
         cout << endl << (char)(i+'A') << ": " << argopts[i];
   cout << endl;
   cout << "arguments:\n";
   for (vector<char *>::iterator i=args.begin();i!=args.end();++i)
      cout << *i << endl;
}
```

Ein Aufruf „`testarg -a hello -bc -E world !`“ liefert folgende Ausgabe:

```
argumentless options: abc
options with arguments:
E: world
arguments:
hello
!
```

5.11.3 Auswertung in Shell-Skripten

Für Shell-Skripte gibt es das Kommando `getopts` (z.B. in `/usr/bin/getopt` oder direkt in die Shells eingebaut), das das Auswerten (*Parsen*) der Kommando-Zeile übernimmt:

```
getopts optstring name [arg]
```

Der String `optstring` enthält alle Buchstaben, die als Optionen erlaubt sein sollen. Optionen, die Argumente haben können sollen, müssen darin von einem Doppelpunkt gefolgt werden. Etwa definiert „`a:bc`“ drei Optionen, davon `a` *mit*, `b` und `c` *ohne* Argument.

`getopt` muss mehrfach aufgerufen werden. Jedesmal untersucht es die *nächste* Options-Angabe und setzt drei Shell-Variable entsprechend:

`$name`	der Name der Option, also ein Buchstabe (‘`?`’ bei unbekannter Option)
`$OPTARG`	das Options-Argument (leer, wenn keines angegeben wurde)
`$OPTIND`	der Index der Option innerhalb der Kommandozeile (ab 1)

Wenn alle Argumente bearbeitet wurden, gibt `getopts` 1 zurück, sonst 0. Diesen Status kann man direkt in einer `while`-Schleife verwenden.

Normalerweise werden die positionalen Parameter untersucht. Wenn allerdings `args` angegeben sind, werden diese geparst. `OPTIND` muss vor einem weiteren Parse-Vorgang (mit einem neuen String) explizit auf 1 zurückgesetzt werden.

Beispiel: Das folgende Skript `prn` ist eine Schnittstelle für das Drucker-Kommando `lpr` und den Prettifier `mp` unter Solaris:

```
prn [-n username] [-h heading] [-t type] [-1] files ...
```

`mp` erzeugt PostScript-Output mit einer Kopfzeile (Dateityp, Benutzername, Seitennummer) und einer Fußzeile (Dateiname oder Titel), die jeweils grau unterlegt sind. Unser Skript benutzt `sed`, um den PostScript-Output zu verändern. Wenn ein anderes Programm statt `mp` verwendet werden soll, müssen die `sed`-Aufrufe entsprechend geändert werden.

Mit den Optionen `-n`, `-h`, `-t` setzt man den Namen (name), den Titel (heading) bzw. den Dateityp (type). Fehlen solche Angaben, werden standardmäßig der Username, der Dateiname bzw. das Ergebnis von `file` verwendet. `file` ermittelt die (wahrscheinliche) Dateiart (Skript, Quelltext).

`-1` ist eine Option ohne Parameter und bewirkt, dass nur eine Seite pro Blatt ausgedruckt wird und nicht zwei nebeneinander, wie voreingestellt ist.

Außerdem überprüft das Skript *immer* mit `file` die Dateiart, um zu verhindern, dass versehentlich Binärdateien ausgegeben werden.

```
#!/bin/sh

user=‘finger \‘whoami\‘ | grep "life:" | tail -1 | sed -e "s/^.*life: //"‘
head=
type=
landscape="-l"

while getopts n:h:t:1 optname
do
  case $optname in
    n) user=$OPTARG;;
    h) head=$OPTARG;;
    t) type=$OPTARG;;
    1) landscape="";;
    ?) echo "Usage: prt [-n username] [-h head] [-t type] [-1] files"
       exit 2;;
  esac
done

shift ‘expr $OPTIND - 1‘
for file in $*
do
  if test -f $file; then
    typestring=‘file $file‘
    thistype=

    if test ‘echo $typestring | egrep ript | wc -l‘ -ne 0
    then thistype="Shell script"
    elif test ‘echo $typestring | egrep rogram | wc -l‘ -ne 0
    then thistype="Listing"
    elif test ‘echo $typestring | egrep commands | wc -l‘ -ne 0
```

```
    then thistype="Commands"
    elif test `echo $typestring | egrep text | wc -l` -ne 0
    then thistype="Text"
    fi

    if test -n "$thistype"
    then
      if test -n "$type"; then thistype=$type; fi
      if test -n "$head"; then thishead=$head; else thishead=$file; fi
      mp $landscape -s $thishead $file \
         | sed -e "s/User ([^)]*) def/User ($user) def/" \
               -e "s/MailFor ([^)]*) def/MailFor ($thistype for ) def/" \
         | lpr
    else
      echo $file "doesn't seem to be printable"
    fi
  else echo "can't open $file"
  fi
done
```

5.12 COMMAND.COM

Das ist der Standard-Befehlsinterpreter unter MS-DOS, der auch unter Windows noch verwendbar ist. Viele Erscheinungen, die wir bei UNIX-Shells kennengelernt haben, gibt es auch bei ihm, wenn auch in eingeschränkter Form (z.B. Shell-Skripte und Umgebungsvariable). Er ist erst nach den UNIX-Shells entstanden, war mit ihnen verglichen zunächst sehr primitiv und ist nur langsam um minimale Kontrollstrukturen erweitert worden. Man gelangt schnell an seine Grenzen, beispielsweise bei Installations-Skripten, und ist gezwungen, eine höhere Programmiersprache zu benutzen.

- Die standardmäßig zu verwendende Shell wird in `CONFIG.SYS` mit der Umgebungsvariable `SHELL` festgelegt, z.B.

  ```
  SHELL=C:\DOS\COMMAND.COM C:\ /P
  ```

  `C:\` legt hier das aktuelle Verzeichnis beim Start fest. Die Variable `COMSPEC` legt den Pfad der Datei `COMMAND.COM fest` – für den Fall, dass Teile neugeladen werden müssen (Standard: Hauptverzeichnis des Boot-Mediums).

- Der Aufbau einer Befehlszeile ist ähnlich wie bei UNIX. Zu beachten ist vor allem, dass das einleitende Zeichen einer Option hier '`/`' ist (statt '`-`'). Zum Einsetzen von Werten von Umgebungsvariablen wird nicht '`$`' vor den Namen gesetzt. Vielmehr wird der Name in Prozentzeichen eingeschlossen: „`%TEMP%`". Kommentare werden nicht mit '`#`', sondern mit „`REM`" (remark) eingeleitet. Da MS-DOS nur Dateinamen mit Großbuchstaben unterstützt, wird nicht zwischen großen und kleinen Buchstaben unterschieden, auch nicht bei eingebauten Befehlen.

- I/O-Umlenkungen (mit `< > >>`) sind analog zu UNIX möglich. Es gibt keinen Fehlerkanal unter DOS, sodass Dinge wie „`2>`" entfallen.

Wie erwähnt, gibt es auch Pipes mit `|` (auch Ketten von Pipes). Die beteiligten Prozesse laufen ohne Multitasking aber *nacheinander* ab. Die Daten werden komplett erzeugt, als temporäre Datei zwischengelagert, dann komplett gelesen und gelöscht.

- Wie bei UNIX-Shells sind einige Kommandos in `COMMAND` fest eingebaut, beispielsweise `ECHO`, `EXIT`, `IF`, usw. Die Umgebungsvariable `PATH` (durch den *Befehl* `PATH` veränderbar) legt fest, wo ansonsten nach Kommandos (als Dateien) gesucht wird.

- Man kann jederzeit ein neues `COMMAND` starten – einfach durch Absetzen des Befehls „`COMMAND`“. Wie bei den UNIX-Shells kann man mit „`COMMAND /C` *command*“ auch direkt einen Befehl ausführen lassen, nach dessen Ausführung `COMMAND` endet.

  Man sollte aber daran denken, dass niemals zwei Interpreter parallel laufen. Unter DOS ist eben kein Multitasking möglich – daher sprechen wir hier auch von „**Kopien**“ statt von Prozessen. Nach dem Neuaufruf von `COMMAND` ist die aufrufende Kopie inaktiv. Wenn die neue Kopie beendet ist, wird sie entfernt, und die alte wird aktiv.

- Es gibt flüchtige und permanente Kopien von `COMMAND`. Letztere entstehen durch Angabe der Option `/P`. Nach dem Booten von MS-DOS wird üblicherweise eine permanente Kopie gestartet (wie oben). Danach gestartete weitere Kopien sind dagegen üblicherweise flüchtig, d.h. sie werden bei ihrer Beendigung vollständig aus dem Speicher entfernt.

- Eine permanente Kopie verhält sich wie folgt: `COMMAND` besteht aus einem residenten Teil, der nach dem Start von DOS permanent im Speicher bleibt, und einem nicht-residenten Teil. Er wird an das Ende des konventionellen Speichers gelegt. Durch den Start eines Programms kann er dabei also durchaus überschrieben werden – um den unter DOS so knappen Hauptspeicher zu schonen. Wenn das passiert ist, lädt der residente Teil nach Beendigung des Programms automatisch den nicht-residenten Teil neu.

- Das Analogon zu Shell-Skripten heißt hier Batch-Dateien, was ihrer Natur auch wesentlich näher kommt. Die Programmiermöglichkeiten sind gegenüber UNIX wesentlich eingeschränkt (s.u.). Batch-Dateien müssen die Endung `.BAT` haben. Sie lassen sich dann durch Angabe ihres Namens (ohne Endung) aus `COMMAND` heraus starten.

- Standardmäßig führt ein permanentes `COMMAND` bei seinem Start die Datei `AUTOEXEC.BAT` aus. Wenn man `COMMAND` mit der Option „`/K` *batchfile*“ (als letzte Option!) aufruft, wird stattdessen die dort angegebene Datei verwendet. Wenn diese Option fehlt und es kein `AUTOEXEC.BAT` gibt, fragt `COMMAND` automatisch mit den Befehlen `DATE` und `TIME` nach der Systemzeit (ein etwas anachronistisches Verhalten).

- Die Befehlszeilen aus den Batch-Dateien werden vor ihrer Ausführung auf dem Bildschirm ausgegeben, es sei denn sie beginnen mit einem ‘`@`’. Das Kommando „`ECHO OFF`“ schaltet den Echo-Effekt ab.

- Batch-Dateien werden nicht in gesonderten Kopien von `COMMAND` ausgeführt, sondern in der aktuellen, also so, als würden die Befehle interaktiv eingegeben.

- Rückgabewerte aus Batch-Dateien gibt es indirekt durch die Variable `ERRORLEVEL`, die am Schluss der Datei entsprechend gesetzt werden kann (0=okay, größere Werte=schlimmere

Fehler). Mit „`IF ERRORLEVEL`“ (s.u.) kann ein aufrufendes `COMMAND` auf diese Werte reagieren.

- Auch unter DOS gibt es Umgebungsvariablen. Leider wird der dafür verwendete Speicher nicht dynamisch angelegt. Man muss beim Start von `COMMAND` seine Maximalgröße angeben, beispielsweise 1 KByte mit „`COMMAND /E:1024`“. Voreingestellt ist 256 Bytes, was bei Verwendung einiger Compiler oder TEX viel zu wenig ist. Mehr als 32 KByte sind nicht erlaubt.

  Einige wichtige Variablen sind:

`%PATH%`	die Liste der Pfade, in denen nach Befehlen gesucht wird (durch ‘`;`’ getrennt statt durch ‘`:`’ wie unter UNIX)
`%PROMPT%`	das Format der Eingabeaufforderung
`%DIRCMD%`	Standard-Optionen für den `DIR`-Befehl
`%COMSPEC%`	die Datei, aus der `COMMAND.COM` Teile seiner selbst nachlädt
`%TEMP%`	Verzeichnis, in dem temporäre Dateien angelegt werden sollen
`%WINDIR%`	Installationsverzeichnis von Windows.

  Die Variablen werden mit dem Kommando „`SET` *name*`=`*value*“ verändert. Wenn *value* leer ist, wird die Variable ganz gelöscht. Der Befehl `SET` alleine zeigt die aktuellen Werte aller Variablen an.

  Neu gestartete Kopien von `COMMAND` erben die Umgebungsvariablen von der aufrufenden Kopie. Da Batch-Dateien nicht in gesonderten Kopien laufen, bleiben dort gesetzte Variableninhalte über den Aufruf hinaus erhalten. Insbesondere nehmen temporär verwendete Variablen weiterhin Speicher im knappen Umgebungsbereich in Anspruch. Man sollte sie am Ende der Batch-Datei mit „`SET` *name*`=`“ löschen.

- Es können auch Parameter an Batch-Dateien übergeben werden. Sie werden über `%0` bis `%9` angesprochen (kein Prozent hinter der Ziffer!). Es gibt auch ein `SHIFT`, das das sichtbare Fenster aber immer um 1 nach rechts verschiebt.

Einige wichtige Befehle für Batch-Dateien sind folgende:

`GOTO` *marke*`:`
: springt an eine andere Stelle der Batch-Datei, die mit dem Label `:`*marke* (am Anfang der Zeile) versehen ist.

`IF` *condition command*`:`
: Der **`IF`**-Befehl ist nicht besonders flexibel.

  Die Bedingung *condition* kann eine der folgenden Formen annehmen:

`ERRORLEVEL` *number*	„Return-Code“ des letzten Befehls $\geq$ *number*
string`==`*string*	String-Gleichheit, Strings z.B. Umgebungsvariablen
`EXIST` *dateiname*	Datei existent

  Jede der Formen darf mit einem führenden **`NOT`** negiert werden.

  Man kann keine „Blöcke“ von Befehlen markieren wie bei **`if`**...**`fi`** unter UNIX. Wenn die Bedingung erfüllt ist, wird der eine (!) Befehl *command* ausgeführt. Dieser Befehl kann ein `GOTO` sein, sodass man über Umwege mehrere Befehle ausführen lassen kann.

Es gibt keine `while`-Schleife in `COMMAND`. Entsprechende Dinge muss man mit `IF` und `GOTO` konstruieren.

`FOR %%`*var* `IN (`*group*`) DO` *command*`:`
: Das ist die einzige Schleifenkonstruktion von `COMMAND`. *group* steht für eine Liste von Strings, die die Wildcards `?` und `*` enthalten dürfen, und die als Dateinamen interpretiert werden. Die Variable *var* durchläuft diese Liste, analog zu UNIX.

  Es kann wirklich nur ein Befehl als Schleifenrumpf angegeben werden. Man kann auch mit `GOTO`s nicht mehrere Befehle ausführen lassen oder gar Schleifen ineinanderschachteln.

`CALL` *batchfile*`:`
: Vorsicht: Wenn man in einer Batchdatei eine andere einfach mit Namen aufruft, wird der Rest der aufrufenden Batchdatei ignoriert! Das liegt daran, dass keine gesonderte Kopie von `COMMAND` aufgerufen wird.

  Wenn man das möchte, muss man den Befehl „`CALL` *batchfile*“ verwenden. Nach der Abarbeitung von *batchfile* wird in der ursprünglichen Datei weitergearbeitet. Die aufgerufene Datei sollte auf jeden Fall mit `EXIT` enden, da `COMMAND` sich sonst leicht aufhängt.

6 Prozesse unter UNIX

Im Abschnitt über Shells haben wir bereits gesehen, dass schon beim Aufruf eines Programms von einer Shell aus ein neuer Prozess kreiert wird. Mit '`&`' kann man Prozesse kreieren, die im „Hintergrund“ weiterlaufen, also parallel zur Shell.

In diesem Kapitel beschäftigen wir uns mit den grundlegenden Mechanismen zur Erzeugung und Vernichtung von Prozessen, hauptsächlich von eigenen Programmen (meist in `C`) aus.

6.1 Prozessnummern

6.1.1 Process ID (PID)

Wie erwähnt, erhalten Prozesse unter UNIX eine Nummer (PID, *process identification*), die sie im System eindeutig identifiziert. Der `C`-Datentyp, der solche Prozessnummern aufnehmen kann, heißt `pid_t` (definiert in `sys/types.h`); oft ist er einfach gleich **`int`**.

Der erste echte Prozess nach dem Booten hat die PID 1, danach wird üblicherweise aufsteigend numeriert. Frei gewordene Prozessnummern werden meist nicht wiederverwendet.

Im Zusammenhang mit Prozessen handhabt die Shell einige Variable, die ähnlich wie Parameter-Variable aussehen:

`$$`	PID des aktuellen Prozesses
`$!`	PID des zuletzt gestarteten Hintergrundprozesses
`$?`	Return-Code des zuletzt ausgeführten Befehls

Wenn noch kein Befehl bzw. Hintergrundprozess ausgeführt wurde, sind die Variablen `$?` bzw. `$!` *leer*.

Beispiel: Mit dem Kommando „`wait` *pid*“ kann man in der Shell auf die Beendigung des Prozesses mit der Nummer *pid* warten. Der Prozess muss dabei als (Hintergrund-)Prozess von dieser Shell erzeugt worden sein, beispielsweise:

```
ghostview skript.ps &
pid=$!
sleep 1        #  damit ghostview definitiv schon läuft, wenn gewartet wird
...            #  sinnvolle Befehle während ghostview
wait $pid
if test $? -ne 0 ; then echo Rückgabewert war $? ; fi
```

6.1.2 Parent Process ID (PPID)

Jeder Prozess (außer 0) hat einen „Elter“, also den Prozess, der ihn erzeugt hat. Dessen Nummer wird, vom Kind aus betrachtet, mit PPID (parent process id) bezeichnet.

In `C`-Programmen erhält man seine eigene Prozessnummer über den Systemaufruf `getpid` (siehe auch Seite 51), die seines Elters mit `getppid` (beide deklariert in `unistd.h`):

	UNIX
`pid_t getpid(void);`	gibt die eigene Prozess-ID zurück
`pid_t getppid(void);`	gibt die Prozess-ID des Elter-Prozesses zurück

Beispiel:

```
#include <iostream.h>
#include <unistd.h>

int main()
 {
   cout << "Ich bin " << getpid() << '\n'
        << "und mein Elter heißt " << getppid() << endl;

   return 0;
 }
```

6.1.3 Process Group ID (PGID)

Prozesse gehören außerdem immer einer **Prozessgruppe** an, in der logisch zusammengehörige Prozesse zusammengefasst sind. Beispielsweise werden „Signale“ (s.u.), die durch das Drücken von `CTRL-C` ausgelöst werden, an alle Prozesse einer Gruppe geschickt.

Wenn ein Prozess einen Kind-Prozess erzeugt, gehört dieses zunächst zur gleichen Prozessgruppe wie der Elter. Es kann sich aber aus dieser Gruppe absondern – durch Tod oder Wechseln in eine andere Gruppe.

Prozessgruppen haben genau wie Prozesse eine Nummer (PGID, *process group ID*), und zwar immer die PID eines Prozesses, der der Gruppe angehört hat. Dieser Prozess heißt **Gruppenführer**. Wenn der Führer die Gruppe verlässt (z.B. stirbt), behält sie dennoch seine Nummer. Sie wird erst aufgelöst, wenn alle Mitglieder sie verlassen haben.

Folgende zwei Funktionen lesen bzw. schreiben die Gruppen-ID:

UNIX
`pid_t getpgid(pid_t pid);` gibt die Prozessgruppen-ID des Prozesses mit der PID pid zurück
`int setpgid(pid_t pid, pid_t pgid);` setzt die PGID des Prozesses `pid` auf `pgid`. Dabei muss `pid` den aktuellen Prozess oder eines seiner Kinder bezeichnen. Für `pid`=0 wird die Nummer des aktuellen Prozesses eingesetzt. Für `pgid`=0 wird die Nummer des wechselnden Prozesses eingesetzt.

Wenn der aktuelle Prozess nicht schon Gruppenführer ist, legt `setpgid(0,0)` also eine neue Prozessgruppe mit ihm als Führer an.

Es gibt einen weiteren verwandten Begriff, den wir hier nicht ausführlich besprechen werden. Ganze Prozessgruppen können wiederum zusammengefasst werden, zu sogenannten **Sessions**.

Das Analogon zum Gruppenführer ist der Session-Leader. Üblicherweise laufen alle Prozesse, die ein Benutzer zwischen Login und Logout erzeugt, innerhalb einer Session und werden bei einem Logout kollektiv vernichtet. Außerdem können Terminal-Ein- und -Ausgabe an Sessions gebunden werden.

Eine Session wird durch eine Session-ID (SID) identifiziert. Die entsprechenden Systemaufrufe sind `getsid` und `setsid`.

6.2 Prozess-Informationen

6.2.1 Der Befehl `ps`

Mit dem Kommando `ps` (Prozess-Status) kann man sich einen tabellarischen Überblick über die im Moment im System befindlichen Prozesse anzeigen lassen:

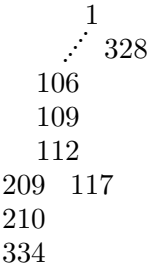
```
UID   PID  PPID PRI  STA TTY TIME COMMAND
 ...
500   109   106   0  S    1  0:00 xinit /usr/X11R6/bin/fvwm95 -- -bpp 16 :0
500   112   109   1  S    1  0:02 fvwm95
500   117   112   0  S    1  0:00 /usr/X11R6/lib/X11/fvwm95/FvwmTaskBar
500   209   112   0  S    1  0:00 xterm -ls
500   210   209  17  S   p2  0:00 -bash
500   328     1   3  S   p2  0:00 xdvi.bin
500   334   210  10  R   p2  0:00 ps l
```

Das genaue Ausgabeformat von `ps` kann mit Optionen beeinflusst werden, variiert aber leider von System zu System sehr stark (am besten jeweils mit `man ps` erkunden). Für die Ausgabe oben wurde `ps -l` verwendet, was üblicherweise das „Langformat“ mit zusätzlichen Informationen einschaltet.

`ps` bietet nur eine Momentaufnahme. Wenn man kontinuierlich über die Prozesslast des Rechners informiert werden möchte, kann man den Befehl `top` verwenden. Es gibt ein Utility `pstree`, das die Elter-Kind-Beziehungen ASCII-grafisch in einem Baum anzeigt.

In der Tabelle erscheinen unter anderem die Nummer des jeweiligen Prozesses (PID), sowie die Nummer des Elter-Prozesses, der ihn erzeugt hat (PPID, parent process id).

Der aktuelle Prozess (also der, der das `ps`-Kommando ausführt) erscheint hier als letztes mit der Nummer 334. Als Elter ist der Prozess 210 angegeben, die Shell (`bash`), die das Kommando `ps` gestartet hat. Diese Shell wiederum hat den Prozess ihres Terminal-Fensters `xterm` als Elter, etc. Der Über-Prozess 1 erscheint selbst nicht in der Tabelle. `xdvi` ist als Hintergrundprozess gestartet und (über einen Zwischenprozess) abgekoppelt worden. Daher wurde ist er PID 1 untergeschoben, der jetzt als Ersatzelter fungiert.



In der Spalte `STAT`) erhält man Informationen über den Prozess-Status. Der führende Buchstabe bedeutet dabei `R`=runnable (also *running* oder *ready*), `S`=sleeping (*blocked*), `T`=traced (gestoppt, z.B. im Einzelschrittmodus).

Die anderen Einträge von oben bedeuten:

`TTY`	die „Nummer“ des Terminals (TeleTYpe), das den Prozess besitzt
`TIME`	die Zeit, die der Prozess bereits die CPU besaß
`COMMAND`	die Kommandozeile, die den Prozess kreiert hat

6.2.2 Das Verzeichnis `/proc`

Es stellt sich noch die Frage, wie ein Programm wie `ps` überhaupt an die Informationen über andere Prozesse gelangen soll. Das ist von System zu System sehr verschieden.

In einigen UNIXen gibt es ein Pseudo-Filesystem, das als „`/proc`“ in den Dateibaum eingehängt ist. (Bei L*i*nux muss dazu z.B. der Kernel mit „`/proc` filesystem support“ compiliert worden sein.) Es liefert in Form von Pseudo-Dateien, die nie physisch existieren, Informationen über laufende Prozesse und andere Systemaktivitäten. Es handelt sich um Kernel-Schnittstellen, die nur wie Dateien angesprochen werden (siehe auch das Kapitel über Filesysteme).

Für jeden laufenden Prozess gibt es ein Pseudo-Directory, das als Namen seine PID hat. Unter L*i*nux enthält es u.a. folgende Pseudo-Dateien:

`cmdline`	die Aufrufzeile, die den Prozess erzeugt hat
`cwd`	Link zum aktuellen Verzeichnis des Prozesses
`environ`	das komplette Environment des Prozesses – die Einträge sind mit Nullbytes (nicht Zeilenvorschüben) getrennt
`exe`	Link auf die ausgeführte Binärdatei
`fd`	Unterverzeichnis, das alle offenen Dateien des Prozesses enthält
`maps`	Informationen über Speicherseiten des Prozesses
`root`	Link auf das Root-Verzeichnis des Prozesses (momentan immer auf „`/`“)
`stat`	diverse Informationen (PID, Kommandoname in Klammern, Status, PPID, PGID, SID, TTY und mehr)
`status`	diverse zeilenweise formatierte Prozess-Informationen

Natürlich hat nicht jeder Prozess Zugriff auf alle diese Daten! Jeder Benutzer darf aber z.B. Kommandozeile und Status auslesen.

Beispiel: Folgendes Skript `pinfo` (in dieser Form nur für L*i*nux!) gibt für die in der Aufrufzeile angegebenen PIDs ausführliche Informationen zurück, die es aus den Pseudo-Dateien `cmdline` und vor allem `status` bezieht:

```
#!/bin/sh
for pid ; do
   if test -d /proc/$pid ; then
      echo; echo -n CMDLINE:
      cat /proc/$pid/cmdline | tr \\000 \\040
      echo
      cat /proc/$pid/status
   else echo pinfo: PID $pid does not exist
   fi
done
```

Das `tr` hinter dem „`cat cmdline`“ wandelt die Nullbytes zwischen den Worten der Aufrufzeile in Spaces (ASCII $32=(40)_8$) um.

Die Ausgabe könnte (leicht abgekürzt) etwa so aussehen:

```
CMDLINE:xdvi.bin -name xdvi -s 4 -bg #e8e8e8 -expert -gamma 1.2 -paper a4 opsys
Name:   xdvi.bin
State:  S (sleeping)
Pid:    208
PPid:   1
Uid:    500     500     500     500
Gid:    100     100     100     100
VmSize:     2900 kB
...
SigIgn: 80000006
```

Beispiel: Das folgende Shell-Skript `minips` (nur L*i*nux!) ist nur zur Demonstration gedacht. Es ist ein „kleines“ `ps`, das PID, PPID und das Aufruf-Kommando ausgibt. Es muss die PPID aus der Datei `stat` herausoperieren und ist durch die häufigen `sed`-Aufrufe leider recht langsam (`set` ist wegen möglicher Spaces im Kommandonamen nicht verwendbar). Es schreibt seine Ausgabe zunächst in ein temporäres File, das am Schluss mit `sort` nach PIDs sortiert ausgegeben wird.

```
#!/bin/sh
tempfile=‘mktemp /tmp/ps.XXXXXX‘
if test -z $tempfile; then exit; fi
echo "  PID  PPID COMMAND"
exec >$tempfile
olddir=‘pwd‘
cd /proc

for pid in * ; do
   if test  -d $pid  -a  $pid -gt 0  2>/dev/null ; then
   ppid=‘cat $pid/stat | sed -e "s/.*) *[^ ]* *//" -e "s/ .*//"‘
   cmdline=‘cat $pid/cmdline‘
   printf "%5d %5d %s\n" $pid $ppid "$cmdline"
   fi
done

cd $olddir
sort $tempfile >/dev/tty
rm $tempfile
```

Unser Skript gibt *alle* Prozesse aus, also auch den Init-Prozess PID 1 und einige andere, die von `ps` underdrückt werden.

Wenn wir einen nicht-numerischen Directory-Namen erwischen (der also keinem Prozess entspricht, z.B. `cpuinfo`), meldet `test` beim Vergleich „≥ 0“ einen Fehler. Wir unterdrücken die Meldungen, indem wir den Fehlerkanal nach `/dev/null` umleiten. (`/dev/null` ist ein Gerätetreiber, der beim Lesen nur End-Of-File meldet und alle auf ihn geschriebenen Zeichen ignoriert, Genaueres später.)

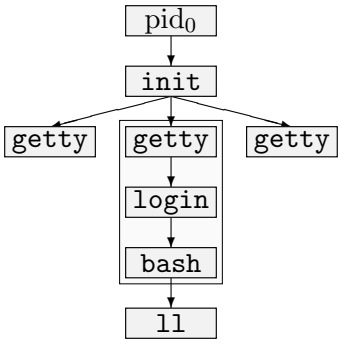
Außerdem gibt es in `/proc` allgemeine Informationen über:

`cpuinfo`	CPU und Systemarchitektur
`devices`	Geräte und Gerätenummern
`filesystems`	verfügbare Dateisysteme
`interrupts`	Interrupts und Interrupts-Nummern
`ioports`	die Belegung des I/O-Speichers
`loadavg`	die Systemauslastung der letzten 1/5/15 Minuten
`meminfo`	freien und belegten Hauptspeicher
`pci`	alle erkannten PCI-Geräte
`version`	die L*i*nux-(Kernel-)Version

Diese Pseudo-Dateien sind für alle Benutzer lesbar, sodass man sie sich also einfach mit `cat` anschauen kann.

6.3 Prozesse beim Systemstart

Während der Initialisierung erzeugt der `init`-Prozess (PID 1) für jedes angeschlossene Terminal einen `getty`-Prozess und teilt ihm einen Gerätetreiber (Tastatur-Eingaben/Konsolen-Ausgaben) zu, beispielsweise `/dev/tty1` usw. Die dazu notwendigen Angaben sind in der Datei `/etc/inittab` festgelegt (s.u.). (Das aktuelle eigene TTY kann man mit dem Befehl `tty` abfragen.)



Dann gibt `getty` die Begrüßungsmeldung aus `/etc/issue` aus, erzeugt die Login-Meldung bei der Passwort-Eingabe, nimmt den Benutzernamen an und *ersetzt* sich durch das `login`-Kommando (z.B. `/usr/bin/login`).

Dieses nimmt das Passwort an und verschlüsselt es mit der Funktion `crypt` (dazu später genauer), holt das korrekte verschlüsselte Passwort mit der Funktion `getpwnam` ab und vergleicht die beiden. Bei Übereinstimmung ermöglicht es das Einloggen des Benutzers.

Dann erledigt es noch einige Einstellungen (Home-Verzeichnis aufsuchen, Superuser-Rechte aufgeben, auf neue Mail hinweisen) und *überlädt* sich schließlich ebenfalls, nämlich durch die für den Benutzer als Standard festgelegte Shell (im Bild ist das die `bash`). Das `ll` im Bild entsteht durch Benutzereingabe dieses Kommandos „`ll`“.

Die drei Programme `getty`, `login` und die Shell laufen also nicht gleichzeitig, sondern im selben Prozess. Der aktuelle Code wird jeweils durch den des nächsten Programms überschrieben. Gleichzeitig laufen aber dieser Prozess, der Init-Prozess und ggf. das Shell-Kommando `ll`.

Wenn das Einloggen schiefläuft (unbekannter Benutzer oder falsches Passwort), verstirbt der Prozess bereits beim `login`. Ansonsten wird nach dem Ausloggen des Benutzers die Shell beendet.

In beiden Fällen wird dem Init-Prozess der Tod eines direkten Kindes gemeldet, und er startet normalerweise erneut ein `getty`. In `/etc/inittab` ist als Aktion für jedes `getty` daher „`respawn`“ angegeben. „`once`“ würde einmaliges Starten bedeuten.

L*i*nux legt standardmäßig sechs virtuelle Textkonsolen an, die man mit `ALT-F1` bis `ALT-F6` anwählen kann. Entsprechend startet es auch 6 `getty`-Prozesse, die `/dev/tty1` bis `/dev/tty6` benutzen:

```
1:123:respawn:/sbin/mingetty --noclear tty1
2:123:respawn:/sbin/mingetty tty2
...
```

Das Format von `/etc/inittab` wird später besprochen.

6.4 Signale

Bei Signalen im UNIX-Sinn handelt es sich um eine Software-Entsprechung zu den schon besprochenen Interrupt-Mechanismen. System-Mechanismen „schicken“ solche Signale an Prozesse, die dann darauf reagieren müssen. Das Eintreffen eines Signals ist ein *asynchrones* Ereignis, das zu jedem Zeitpunkt eintreten kann. Der Prozess wird aus seiner normalen Befehlsverarbeitung herausgerissen.

Signale sind die simpelste Art von Kommunikation zwischen Prozessen. Die einzige Information, die transportiert wird, ist allerdings die *Nummer* des Signals.

Sie werden unter UNIX durch kleine positive ganze Zahlen repräsentiert, die allerdings nicht auf allen Systemen gleich sind. L*i*nux hat 30 unterschiedliche Signale.

Beispielsweise gibt es das Signal `INT` (bzw. `SIGINT`), das meistens die Nummer 2 hat und durch ein `CTRL-C` ausgelöst wird. Es wird den Prozessen geschickt, die mit dem Terminal gekoppelt sind, auf dem das `CTRL-C` gedrückt wurde (dazu später genauer).

Signale werden ausgelöst:

- in Fehlerfällen (fehlerhafter Speicherzugriff, Division durch 0)
- wenn ein Prozess über bestimmte Ereignisse benachrichtigt werden will (Eintreffen von I/O-Daten, Timer-Ablauf, etc.)
- durch Benutzer-Eingriff (`CTRL-C`, `CTRL-Z`, `CTRL-^`)
- durch den System-Aufruf `kill`

Die Namen aller erlaubten Signale erhält man üblicherweise mit dem Aufruf „`kill -l`“ (list). Jedes Signal hat eine (von System zu System variierende!) Nummer, die man alternativ angeben kann (z.B. meist „`kill -9`“ statt „`kill -KILL`“).

Einige wichtige Signale sind:

`SIGHUP` (Hangup), falls die Verbindung zum Terminal verlorengeht

`SIGALRM` (Alarm), erhält der Prozess nach einem `alarm`-Aufruf nach Ablauf der eingestellten Zeitspanne

`SIGINT` (Interrupt), falls die Abbruchtaste gedrückt wird, meist `CTRL-C` oder `BREAK` o.ä.

`SIGQUIT` (Quit), falls die „Quit-Taste“ gedrückt wird, meist `CTRL-^`; entspricht `SIGINT`, legt aber noch einen Core-Dump an (falls das nicht systemweit abgeschaltet ist)

SIGTERM (Terminate), kann nur durch den `kill`-Systemaufruf zum Prozess gelangen; der Prozess erhält noch Gelegenheit, „aufzuräumen“

SIGKILL (Kill), unbedingter Prozessabbruch; kann nur durch den `kill`-Systemaufruf zum Prozess gelangen; keine Gelegenheit mehr, „aufzuräumen“; kann niemals abgeschaltet werden

SIGWINCH (Window Change), ein anderes Fenster wurde aktiviert

Die Signale unter Linux-x86 sind in der folgenden Tabelle aufgeführt:

Signale unter Linux							
0	–	8	SIGFPE	16	SIGSTKFLT	24	SIGXCPU
1	SIGHUP	9	SIGKILL	17	SIGCHLD	25	SIGXFSZ
2	SIGINT	10	SIGUSR1	18	SIGCONT	26	SIGVTALRM
3	SIGQUIT	11	SIGSEGV	19	SIGSTOP	27	SIGPROF
4	SIGILL	12	SIGUSR2	20	SIGTSTP	28	SIGWINCH
5	SIGTRAP	13	SIGPIPE	21	SIGTTIN	29	SIGIO
6	SIGABRT	14	SIGALRM	22	SIGTTOU	30	SIGPWR
7	SIGBUS	15	SIGTERM	23	SIGURG	31	SIGUNUSED

Beispiel: Unter UNIX kann man Prozesse zwangsweise blockieren oder ganz entfernen, indem man ihnen spezielle Signale schickt. Das geschieht von der Shell aus mit dem Kommando `kill` (das letztendlich auf den Systemaufruf `kill` zurückgeführt wird).

pid soll die Prozess-Nummer sein, die man mit `ps` erfragen kann. Z.B. sind folgende Befehle sinnvoll:

`kill -STOP` *pid*	Ein laufender Prozess (running oder ready) wird gestoppt, d.h. er wird blocked, bis er ein Signal erhält.
`kill -CONT` *pid*	Ein mit `STOP` angehaltener Prozess wird wieder ready.
`kill -TERM` *pid*	Ein Prozess (gleich, in welchem Zustand) wird terminiert und aus dem System entfernt.

Ein `CTRL-Z` in der Shell schickt dem von dort aus gestarteten Vordergrund-Prozess ein `STOP`-Signal, und die Shell wird wieder bereit, Befehle anzunehmen. Durch den Befehl `bg`, der den Prozess „in den Hintergrund schiebt“, schickt ihm die Shell ein `CONT`-Signal.

Wenn ein Prozess angehalten wurde, reagiert er nicht auf irgendwelche Eingaben (bei fensterorientierten Programmen z.B. auch auf Mausklicks, nötiger Grafik-Neuaufbau, etc.). Diese „Ereignisse“ werden aber vom System in einer Warteschlange gespeichert, und der Prozess arbeitet sie nacheinander ab, sobald er weiterläuft.

Ein angehaltener Prozess erhält ein ‘`T`’ in der `STAT`-Spalte bei `ps`. Der Buchstabe kommt dadurch zustande, dass dieser Zustand u.a. für die Einzelschritt-Abarbeitung (tracing) verwendet werden kann.

- Ein Prozess hat nicht die Erlaubnis, beliebigen anderen Prozessen Signale zu schicken – das können nur Super-User-Prozesse. Normalerweise sind nur Prozesse desselben Besitzers und Prozesse in derselben Prozess-Gruppe erreichbar.

- Ein Prozess kann durch Systemaufrufe explizit angeben, was beim Eintreffen von Signalen passieren soll. Diese Mechanismen lernen wir etwas später kennen.
- Es ist nicht in POSIX definiert, wann Signale den Ziel-Prozess erreichen müssen. Es bleibt der Implementation freigestellt, bei einem `signal`-Aufruf sofort den angesprochenen Prozess zu aktivieren und dessen Signal-Reaktion auszulösen.

  Üblicherweise lösen die Signale aber nicht sofort den Handler im Ziel-Prozess aus. Beim Dispatching und am Ende jedes Systemaufrufs wird nach inzwischen eingetroffenen Signalen geschaut. Wenn der Prozess gerade nicht aktiv ist, erreicht ihn ein Signal, sobald der Dispatcher ihn das nächste Mal aktiviert. Wenn er aktiv ist, hat er sich das Signal entweder direkt selbst geschickt, oder der Kernel hat es indirekt generiert, ausgelöst durch einen System-Aufruf des Prozesses, und am Ende des Aufrufs wird auf das Signal reagiert.

Beispiel: Das folgende Skript `timeout` lässt einem angegebenen Befehl nur eine gewisse Zahl von Sekunden – danach wird er zwangsweise abgebrochen. Ein möglicher Aufruf ist „`timeout 5 xlogo`“.

```
timeout=$1
if test $timeout -gt 0 2>/dev/null ; then
   shift 1
   $* &
   sleep $timeout
   kill -TERM $!
fi
```

Beispiel: Das folgende Skript zeigt alle PostScript-Dateien im aktuellen Verzeichnis an, jeweils mit einem eigenen `ghostview`-Prozess. Es wartet auf eine Eingabe und bricht dann alle erzeugten Prozesse wieder ab. Die Fehler-Ausgabe von `kill` wird weggeworfen, damit man nicht für nicht mehr existente Prozesse (per Hand geschlossene Fenster) Meldungen erhält.

```
for file in *.ps; do
   ghostview $file &
   pids="$pids $!"
done
echo RETURN zum Abbrechen
read
for pid in $pids; do
   kill $pid 2>/dev/null
done
```

Häufig wird mit Bitmasken gearbeitet, in denen ein gesetztes Bit für ein zu sendendes, empfangenes bzw. zu sperrendes Signal steht. Wenn (wie üblich) für diese Masken gerade ein Maschinenwort verwendet wird, ist die Anzahl möglicher Signale von Natur aus beschränkt – auf 32-Bit-Systemen (L*i*nux-x86) z.B. auf 32. Durch diese Arbeitsweise sieht ein Prozess außerdem nur ein Signal, obwohl schnell hintereinander (bevor er einmal reagieren kann) dasselbe Signal mehrfach geschickt wurde.

Eine Bitmaske im Prozess-Kontrollblock stellt meist die bereits eingetroffenen Signale, eine weitere die momentan blockierten Signale dar, unter L*i*nux beispielsweise:

```
struct task_struct          // Linux (sched.h)
 {
   ...
   unsigned long signal;    // 32 Bits für eingetroffene Signale
   unsigned long blocked;   // 32 Bits für blockierte Signale
   ...
 };
```

Beispiel: Der folgende Code aus dem L*i*nux-Kernel (`exit.c`) ist für das Verschicken eines Signals Nummer `sig` an den durch `p` dargestellten Prozess zuständig. `priv` gibt an, dass der Kernel selbst das Signal schickt (ansonsten muss geprüft werden, ob der sendende Prozess dazu berechtigt ist).

```
int send_sig(unsigned long sig, struct task_struct *p, int priv)
 {
   unsigned long flags;

   if ( p==NULL || sig>32 ) return -EINVAL;     // ungültige Signalnummer

   if (!priv && ((sig != SIGCONT) || (current->session != p->session)) &&
      (current->euid^p->suid) && (current->euid^p->uid) &&
      (current->uid^p->suid) && (current->uid^p->uid) &&
      !suser())
         return -EPERM;              // Prozess nicht berechtigt

   if (sig==0) return 0;             // 0 testet nur die Berechtigung
   if (!p->sig) return 0;            // keine Handler ⇒ das ist ein Zombie!

   save_flags(flags); cli();         // jetzt bitte keine Interrupts...

   if ( sig == SIGKILL || sig == SIGCONT )
    {
      if (p->state == TASK_STOPPED) wake_up_process(p);
       p->exit_code = 0;
       p->signal &= ~( (1<<(SIGSTOP-1)) | (1<<(SIGTSTP-1))
                     | (1<<(SIGTTIN-1)) | (1<<(SIGTTOU-1)) );
    }

   if ( sig == SIGSTOP || sig == SIGTSTP || sig == SIGTTIN || sig == SIGTTOU )
      p->signal &= ~(1<<(SIGCONT-1));

   restore_flags(flags);
   generate(sig,p);                  // eigentliches Erzeugen, ggf. Wecken, etc.
   return 0;
 }
```

6.5 Beendigung von Prozessen

- Ein Prozess *endet* normalerweise, wenn er selbst den Systemaufruf `_exit` tätigt. Das geschieht von C-Programmen aus

- durch Aufruf der Funktion `exit(int ret_value)`
- durch ein **return** `ret_value` im Hauptprogramm `main`

Wenn `main` ohne ein **return** endet, erzeugen die meisten Compiler Code, der den Wert 0 zurückliefert.

In beiden Fällen wird der Abschlusscode aktiv, der vom Linker automatisch hinzugebunden wird. Er gibt allen noch belegten Speicher frei, schließt alle noch offenen Dateien, etc. Erst *dann* ruft er `_exit` auf.

Der Rückgabewert `ret_val` dient zur Kommunikation mit dem erzeugenden Prozess. Üblicherweise wartet dieser (spätestens kurz vor seinem eigenen Ende) mit den `wait`-Funktionen auf seine Kinder und kann sich an dieser Stelle die zurückgegebenen Werte abholen (dazu später). Aus technischen Gründen machen nur Werte zwischen 0 und 255 Sinn.

- Wenn ein Prozess `_exit` aufgerufen hat, wird er *nicht sofort* aus dem System entfernt. Der erzeugende Prozess möchte ja im allgemeinen seinen Rückgabewert abholen. Dem Elter wird das Signal `SIGCHLD` geschickt, die Standard-Reaktion darauf ist allerdings Nichtstun.

  Deshalb stirbt das Kind erst endgültig, wenn der Elter durch ein `wait` von seinem Tod Kenntnis genommen hat. In der Zwischenzeit vegetiert er noch als sogenannter „**Zombie**"-Prozess dahin. Seine Ressourcen werden freigegeben, sein Rückgabewert wird dagegen aufbewahrt, bis sein Elter ihn mit einem `wait` abfragt. Erst dann wird das Kind aus der Prozesstabelle entfernt.

  Der Elter-Prozess kann die Erzeugung von Zombies ganz verhindern, indem er das Signal `SIGCHLD` ignoriert. Rückgabewerte gehen dann aber verloren!

- Wenn dagegen ein Elter-Prozess stirbt, werden alle seine verwaisten Kinder vom `init`-Prozess 1 adoptiert (dazu muss der Kernel bei jedem Prozess-Tod alle aktiven Prozesse überprüfen). Die Information, wer der ursprüngliche Elter war, wird aber aufgehoben.

- Ein *abnormaler* Programmabbruch wird mit der Bibliotheksfunktion `abort` eingeleitet, die das Signal `SIGABRT` verschickt. Noch offene Dateien werden dabei geschlossen. Es kann kein Rückgabewert an den Elter-Prozess geliefert werden. Der Kernel erzeugt einen künstlichen Rückgabewert, der die Information über den Abbruch beinhaltet.

- Außerdem kann ein Prozess von einem Elter-Prozess zwangsweise beendet werden, indem ihm mit dem Systemaufruf `kill` ein entsprechendes Signal geschickt wird:

	UNIX
int kill**(**pid_t pid**, int** sig**);** schickt dem Prozess `pid` das Signal `sig`	

Die Signale entsprechen dabei denen beim Kommando `kill`. Wenn der aktuelle Prozess keine Superuser-Rechte hat, können nur Prozesse des selben Besitzers gekillt werden.

Beachte: Wenn man als `pid` eine 0 angibt, wird nicht etwa der Urprozess Nummer 0 gekillt, sondern alle Prozesse der eigenen Prozessgruppe! Mit `pid`=-1 kann der Super-User *alle* Prozesse überhaupt (außer 0 und 1) killen.

Folgende ANSI-C-Bibliotheksfunktion schickt dem eigenen Prozess ein Signal (anwendbar auch auf Ein-Prozess-Systemen):

ANSI-C
`int raise(int sig);` schickt dem *aktuellen* Prozess das Signal `sig`; d.h. es ist äquivalent zu `kill(getpid(),sig)`

- Es gibt eine `C`-Funktion `atexit` (in `stdlib.h`), mit der man „Handler-Routinen“ angeben kann, die automatisch bei einem regulären Programmende (`exit` oder **`return`**) aufgerufen werden:

UNIX/DOS
`int atexit(void (*function)(void));` hängt einen Handler in die Liste der Exit-Handler ein

  Der Parameter ist also ein Pointer auf eine Funktion mit folgendem Prototyp:

```
void handler_function(void);
```

  Dort kann man beispielsweise Abschlusswerte in Dateien schreiben, Operationen im Zusammenhang mit dynamischem Speicher vornehmen, etc.

6.6 `fork`

In UNIX werden *alle* Prozesse durch Verdopplung mit dem Systemaufruf `fork` („Gabelung“) erzeugt. Der alte und der neu entstandene Prozess unterscheiden sich (fast) allein durch den Rückgabewert des `fork`-Aufrufs.

UNIX
`pid_t fork(void);` legt einen Kind-Prozess als Kopie des aktuellen Prozesses an. Der Elter-Prozess erhält als Rückgabewert die PID des Kinds, das Kind erhält 0 zurück. Im Fehlerfall erhält der Elter eine -1.

`fork` hat zwei Hauptanwendungen:

- Beide Prozesse verbleiben **im selben Programm**. Sie teilen sich also den Code, befinden sich aber (meist) an unterschiedlichen Stellen. Server-Prozesse, die gleichzeitig mehrere Dienste für mehrere Clients ausführen sollen, arbeiten meist so.
- Einer der Prozesse überlädt sich nach dem `fork` durch `exec` mit dem Code eines **neuen Programms**. Dann ging es nur darum, ein anderes Programm aufzurufen. Das ist der typische Fall einer Shell, in der ein Kommando abgesetzt wird.

Beispiel: Im folgenden Programm konkurrieren Elter und Kind um die Ausgabe:

```
#include <iostream.h>
#include <sys/types.h>
#include <unistd.h>
```

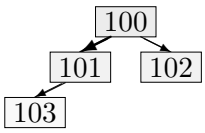
```
int main()
 {
   int i;
   pid_t pid=fork();

   if (pid)                        // ich bin der Elter-Prozeß
      for (i=0;i<10;++i)
       {
         cout << i << endl;
         sleep(1);
       }
   else                            // ich bin der Kind-Prozeß
      for (i=100;i<110;++i)
       {
         cout << i << endl;
         sleep(1);
       }
 }
```

Die Ausgabe ist (in etwa) „`0 100 1 101`...“. Jedenfalls ist zu erkennen, dass jeder der beiden Prozesse eine *eigene Kopie* der Variablen `i` hat.

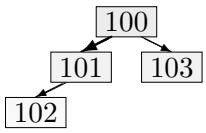
Nach der Verzweigung laufen beide Prozesse parallel weiter. Welcher der beiden als nächstes die CPU erhält, ist nicht festgelegt.

Beispiel:
Mit mehreren `fork`s ergibt sich schnell ein ganzer Baum von Prozessen. Das Codestück rechts erzeugt nämlich nicht nur *zwei* neue Prozesse, sondern *drei*.

```
int main()
 {
   fork();
   fork();
   cout << getpid() << endl;
 }
```



Elter- und Kind-Prozess des ersten `fork` spalten sich danach noch einmal auf. Wenn die PIDs ab 100 verteilt würden, ergäbe sich einer der dargestellten Bäume.



Beispiel: Das folgende Programm ruft `xlogo` auf – ein simples X-Window-Programm, das ein Fenster mit einem großen ‘`X`’-Symbol anzeigt.

```
int main()
 {
   if (fork()==0)
      execlp("xlogo", "xlogo", 0);
 }
```

Genaueres zu den `exec`-Aufrufen und ihren Parametern folgt im nächsten Abschnitt. Unser Programm wartet nicht auf die Beendigung des `xlogo`-Prozesses. Wenn es beendet ist, wird er dem Prozess 1 als Kind untergeschoben (wie wir mit `ps` überprüfen können).

- Das Codesegment ist auf den meisten Systemen gar nicht beschreibbar. Es wird daher fast nie kopiert, sondern von beiden Prozessen gelesen.

Unter L*i*nux wird beim Duplizieren *nicht wirklich* das gesamte Datensegment und der Heap *kopiert*. Meistens wird ja im Kind doch sofort ein `exec` aufgerufen, und diese Daten werden überschrieben!

Es werden spezielle Speicherverwaltungs-Mechanismen verwendet: „*copy-on-write*“-Speicherseiten (COW). Elter und Kind verwenden *denselben* Speicher, der als *read only* markiert wird. Wenn das erste Mal ein Schreibvorgang stattfinden soll, fängt der Kern diesen Fehler ab, legt für das Kind eine Kopie an und markiert beide Kopien wieder als beschreibbar.

Meistens muss also nur die Seitentabelle kopiert werden, und ein `fork` bedeutet keinen übermäßigen Zeitaufwand.

Es gibt meist einen weiteren Systemaufruf `vfork`, der auf allen Systemen das COW-Verhalten von oben erzwingt. Unter L*i*nux ist `vfork` einfach mit `fork` identisch.

Vorsicht: Manchmal erlaubt es `vfork` dem Kind, direkt und ununterbrochen abzulaufen, bis es ein `exec` ausführt oder beendet wird. Man will es dem Kind möglichst schnell ermöglichen, sich zu überladen. Bei unvorsichtiger Programmierung kann man aber so das System aufhängen.

- Die beiden Prozesse haben nach dem `fork` Zugriff auf dieselben Betriebsmittel. Dateien bleiben in beiden Kopien geöffnet, und damit bleiben insbesondere Zugriffskanäle auf I/O-Geräte bestehen! Wenn klar ist, dass eine Kopie bestimmte Dateien nicht mehr benötigt, sollte man daran denken, sie dort zu schließen.

  Die File-Deskriptoren beziehen sich auf dieselben Einträge in der systeminternen Dateitabelle. Beide Prozesse dürfen also schreibend auf eine Datei zugreifen, ohne dass zu befürchten wäre, dass sie sich ihre Ausgaben gegenseitig überschreiben würden. Ihre Ausgaben werden vielmehr gemischt in der Datei ankommen.

  Mit dem Systemaufruf „`fcntl(fd,F_SETFD,1)`“ (Funktion deklariert in `fcntl.h`) kann man eine Datei so manipulieren, dass sie automatisch von einer Prozess-Kopie geschlossen wird, wenn ein `exec` ausgeführt wird (durch das „close-on-exec“-Flag). Die Kopie führt danach ja ein anderes Programm aus und könnte mit den alten File-Deskriptoren nichts mehr anfangen.

Eine Shell muss mit Verdopplung arbeiten, jedesmal, wenn sie auf ein Benutzer-Kommando hin ein Programm starten soll. Der neue Prozess überlädt dann aber sofort den alten Shell-Code mit dem Code des Programms. Dazu bedient es sich eines der `exec`-Aufrufe (s.u.).

In einer *Shell* wird meistens sequentiell gearbeitet, d.h. der Benutzer arbeitet meistens nur mit dem gerade gestarteten Programm, und auch die Shell braucht in dieser Zeit nicht aktiv zu sein.

- Deshalb ist das normale Verhalten einer Shell, zu warten, bis der zuletzt gestartete Prozess beendet ist. Das geschieht mit einem `wait`- oder `waitpid`-Aufruf.
- Wenn der Benutzer ein ‘`&`’ hinter das Kommando setzt, laufen Shell und neues Programm echt parallel weiter. Das ist aber das Normalverhalten des `fork`-Aufrufs. Hier braucht also nicht gewartet zu werden.

- Ganz am Ende der Shell sollte aber auf jeden Fall auf eventuell noch laufende Kind-Prozesse gewartet werden, da sie sonst im System bleiben. Wenn der Benutzer so ein Verhalten wünscht, sollte er das Kommando „`nohup`“ (no hangup) verwenden, das einen Prozess von seinem Elter (und dem zugehörigen Terminal) abkoppelt.

Beispiel: Wir wollen einige einfache Mechanismen der Shell nachbilden. Unsere erste Version ist natürlich noch sehr minimal.

Sie nimmt nacheinander Kommandozeilen an, bis `exit` eingegeben wird. Optionen, Parameter, Umlenkung etc. werden einfach ignoriert. Es wird nur versucht, das Programm zu starten, das dem ersten eingegebenen Wort entspricht. Danach wartet die Shell auf dessen Beendigung, also das normale Shell-Verhalten ohne '`&`'.

```
#include <unistd.h>                          // für fork, execlp, sleep
#include <sys/wait.h>                        // für waitpid
#include <sys/types.h>                       // für pid_t
#include <iostream.h>                        // C++-I/O
#include <string.h>                          // für strtok und strcmp

int main()
 {
   for (;;)
    {
      char buffer[256];
      cout << "nullsh: ";
      cin.getline(buffer,256);
      if (*buffer!=0)
       {
         char *command=strtok(buffer," ");
         if (strcmp(command,"exit")==0) break;

         pid_t pid=fork();                   // Prozess duplizieren
         if (pid==0)                         // "ich bin das Kind"
          {
            execlp(command,command,0);       // überlädt den Prozess!
            cerr << "command not found!\n";  // hier kommt man nur hin, wenn
            _exit(0);                        // execlp schieflief!
          }
         else                                // "ich bin der Elter"
            waitpid(pid,0,0);                // auf den Tod des Kinds warten
       }
    }
 }
```

Das Programm kann man z.B. mit `g++ nullsh.cpp -o nullsh` übersetzen. Die Shell gibt als Prompt „`nullsh:`“ aus. Mit `strtok` isoliert man das erste „Wort“ der Eingabe (führende Spaces werden gestrichen) als Kommando.

Vorsicht: `strtok` verändert die Eingabezeile durch Einbau von Nullbytes. Um aber nicht durch komplizierten eigenen Parse-Code vom eigentlichen Problem abzulenken, verwenden wir die Funktion dennoch.

Die Eingabe „`exit`“ bricht das Programm ab.

Ansonsten wird ein Kind erzeugt, das versucht, das Programm mit dem eingegebenen Namen zu starten. Bei Erfolg wird die Kind-Shell ersetzt durch das neu geladene Programm. Bei Misserfolg landet das Kind dagegen in der Zeile hinter `execlp` und gibt einen Fehler aus. Wir brechen das Kind dann mit `_exit` ab. Wir dürfen nicht `exit` verwenden, da das die Ressourcen auch des Elter-Prozesses freigeben würde!

Beispielsweise kann man „`ls`“ oder „`date`“ als Kommandos ausprobieren. Programme, die Parameter unbedingt benötigen, liefern so natürlich nur Fehlermeldungen. Eine „richtige“ Shell sollte natürlich die Eingabezeile entsprechend auswerten und die Parameter im `execlp`-Aufruf angeben.

Nach der Eingabe von `xv` erhält man mit dem `ps`-Kommando (aus einer anderen Shell heraus) eine Ausgabe der folgenden Art:

```
UID   PID  PPID PRI  STA TIME COMMAND
  0   210   209  15  S   0:00 -bash      ← die zweite Shell
  0   288   119   1  S   0:00 nullsh     ← unsere Shell
  0   289   288  20  R   0:01 xv         ← xv aus unserer Shell
  0   290   210  10  R   0:00 ps l       ← aktueller Prozess
```

Unsere Shell „schläft“ (Status S), weil sie im `waitpid`-Systemaufruf hängt. Die andere Shell (`bash`) hat denselben Status, da sie gerade auf eine Eingabe wartet. `xv` und `ps` sind *ready* bzw. *running*.

Einige Befehle (wie `cd`) sind unter UNIX nicht als eigene Programme abgelegt, sondern direkt in die Shells eingebaut – genau, wie wir es auch mit `exit` getan haben. `cd`, `if`, `while`, etc. führen deshalb bei uns zu einer Fehlerausgabe.

Wenn man diese Shell (z.B. mit „`nullsh`“) ineinander verschachtelt aufruft, startet sie also Kopien von sich selbst. Man muss dann zum Abbruch mehrere Male „`exit`“ eingeben!

Vorsicht: Wir fangen hier keine System-Signale ab! Wenn wir während der Ausgabe eines Befehls `CTRL-C` drücken, brechen wir nicht *ihn* ab, sondern unsere Shell (die ihren Unterprozess mit killt).

Der Übersicht halber ist hier ein Fall ausgelassen. `fork` kann auch den Wert -1 zurückgeben, um anzudeuten, dass der Kind-Prozess wegen Speichermangel nicht angelegt werden konnte. Die Fallunterscheidung sollte also eigentlich in etwa so aussehen (`errno.h` muss eingebunden werden):

```
switch (pid)
 {
   case 0:                                        // ich bin das Kind
      ...
   case -1:                                       // Elter, aber kinderlos
      fprintf(stderr,"ERROR: %s\n",sys_errlist[errno]);
      break;
   default:                                       // Elter, jetzt mit Kind
      ...
 }
```

6.7 Warten auf Prozesse

6.7.1 Die `wait`-Funktionen

Es gibt mehrere Funktionen (deklariert in `sys/wait.h`), mit denen ein Elter-Prozess auf das Ende seiner Kinder warten kann:

	UNIX
`pid_t wait(int *status);` wartet auf das Ende *irgendeines* Kindes.	
`pid_t waitpid(pid_t pid, int *status, int options);` wartet üblicherweise auf das Ende eines *bestimmten* Kinds, nämlich auf das mit der Prozessnummer `pid` (> 0) (mit `pid=-1` verhält sich `waitpid` wie `wait`)	

Mit Bitmasken in `options` kann man das Verhalten der Funktion `waitpid` modifizieren. Setzt man z.B. `WNOHANG`, kehrt die Funktion sofort zurück, wenn nicht zuvor schon ein Kind gestorben ist, ohne dass auf es gewartet wurde.

`waitpid` stammt aus `POSIX`, ist aber weit verbreitet. Aus BSD stammen Funktionen `wait3` und `wait4`, die es aber in relativ wenigen UNIXen gibt.

Wenn auf einen Kind-Prozess gewartet werden soll, den es schon gar nicht mehr gibt, kehren alle Funktionen sofort zurück. Der Elter-Prozess ist während des Wartens im blocked-Zustand. Er kann allerdings mit Signalen daraus aufgeweckt werden.

Beide Funktionen liefern als Rückgabewert die Nummer des gestorbenen Kinds, bzw. die des zuerst gestorbenen Kinds zurück (oder 0, wenn niemand gestorben ist). Informationen über diesen Prozess liefern sie in dem Integerwort `s` zurück, auf das `status` zeigt. Mit Makros aus `sys/wait.h` kann man dieses Wort auswerten:

`WEXITSTATUS(s)`	die untersten 8 Bit des Returncodes des Kindes, d.h. des Werts, der bei seinem `exit()` oder `return` (in `main`) angegeben wurde
`if (WIFEXITED(s)) ...`	falls das Kind normal beendet wurde
`if (WIFSIGNALED(s)) ...`	falls das Kind durch ein unbeantwortetes Signal gestorben ist
`WTERMSIG(s)`	Nummer des Signals, dessentwegen das Kind gestorben ist (falls `WIFSIGNALED`$\neq 0$)

(Weitere Bits zeigen an, ob ein gestopptes Kind gekillt wurde, ob ein Coredump erzeugt wurde, etc.)

Beispiel: Wir rufen wiederum `xlogo` auf, warten diesmal aber korrekt auf dessen Beendigung, d.h. darauf, dass das Fenster geschlossen wird oder jemand einen `kill`-Befehl absetzt:

```
int main()
 {
   pid_t pid=fork();

   if (pid==0)
      execlp("xlogo", "xlogo", 0);
```

```
  else
   {
     int s;
     waitpid(pid,&s,0);
     if (WIFEXITED(s))
        cout << "normal beendet, Rückgabewert " << WEXITSTATUS(s) << endl;
     else if (WIFSIGNALED(s))
        cout << "beendet durch Signal Nummer " << WTERMSIG(s) << endl;
     else
        cout << "Überraschung...\n";
   }
}
```

6.7.2 Zombies

Wenn man auf seine Kinder nicht wartet, werden sie bekanntlicherweise nach ihrem Tode zu Zombies, solange man noch selbst am Leben ist:

Beispiel: Hier wartet der Elter-Prozess einmal *nicht*:

```
int main()
 {
   if (fork()==0)
      execlp("xlogo","xlogo",0);
   else
      for (;;) sleep(10);
 }
```

Der Elter-Prozess führt hier der Einfachheit halber einen endlosen Schlaf mit kurzen Unterbrechungen aus. Wenn wir das `xlogo`-Fenster schließen, erzeugen wir einen Zombie, wie ein `ps`-Aufruf zeigt (einige Systeme zeigen statt „`<zombie>`“ ein „`<defunct>`“):

```
 PID TTY STAT TIME COMMAND
...
3485  p3 Z    0:00 (xlogo <zombie>)
```

Wenn man sich nicht um seine Kinder kümmern, aber Zombies vermeiden möchte, kann man einfach zwei `fork`s ineinander schachteln:

```
int main()
 {
   pid_t pid=fork();

   if (pid==0)
    {
      if (fork()==0)
         execlp("xlogo","xlogo",0);
    }
   else
    {
      waitpid(pid,0,0);
```

```
      for (;;) sleep(10);
    }
  }
```

Der ursprüngliche Prozess ist nun *Großelter* von `xlogo`. Der Elter stirbt direkt nach dem zweiten `fork`. Der `xlogo`-Prozess ist also verwaist und wird von PID 1 adoptiert. PID 1 wartet immer auf seine Kinder, auch auf untergeschobene. Daher kann so `xlogo` nie ein Zombie werden.

Vorsicht: Der Großelter muss allerdings auf sein eigenes Kind warten, da sonst *dieses* zum Zombie wird! Das kann allerdings vor den eigentlichen Aktivitäten direkt hinter dem `fork` geschehen, da das Kind ja nur kurz den Enkel in die Welt setzen muss.

6.7.3 Hintergrundprozesse in der Shell

Wir modifizieren unsere Shell von oben so, dass sie einfach nicht auf die Beendigung der erzeugten Prozesse wartet. Sie verhält sich also in etwa so wie normale Shells, wenn hinter jedem Befehl ein '`&`' angegeben wird.

Wir entfernen einfach folgendes:

```
else waitpid(pid,0,0);
```

Zum Testen können wir ein Kommando „`forever`“ (in `C` oder als Shell-Skript) schreiben, das im Sekundentakt zählt:

```
int main()
 {
   for (int i=0;;++i)
    {
      cout << i << endl;
      sleep(1);
    }
 }
```

```
#!/bin/sh
count=1
while true; do
   echo $count
   count=‘expr $count + 1‘
   sleep 1
done
```

Wenn wir von unserer `nullsh2` aus `forever` starten, erhalten wir dennoch wieder einen Shell-Prompt. Wir können dann noch beliebig viele `forever` starten, deren Ausgaben gemischt auf dem Bildschirm erscheinen.

Dadurch, dass wir im Moment *gar nicht* auf unsere Kinder warten, handeln wir uns zwei Unannehmlichkeiten ein:

- Wenn wir mit `exit` die Shell beenden, laufen die `forever`-Prozesse weiter! Sie sind vom `init`-Prozess adoptiert worden. Sie lassen sich auch nicht mehr mit `CTRL-C` abbrechen und müssen mit `kill` getötet werden.
- Wenn wir andere Prozesse, die vor der Shell beendet sind (`ls`, `pwd` o.ä.) starten, werden diese nach dem Tod zu Zombies, was wir mit dem `ps`-Kommando (in einem anderen Fenster) nachprüfen können:

  ```
  844  p3 Z    0:00 (ls <zombie>)
  848  p3 Z    0:00 (pwd <zombie>)
  ```

  Erst, wenn wir unsere Shell beenden, werden alle Kinder mitbeseitigt, und die Zombies verschwinden.

Zweiteres verhindern wir, indem wir nach jeder Eingabe die Prozesse mit `wait` „verabschieden“, die in der Zwischenzeit beendet wurden.

```
int s;
while (waitpid(-1,&s,WNOHANG)>0);
```

Bis die nächste Eingabe verarbeitet ist, kann es so allerdings kurzzeitig zu Zombies kommen (siehe die Bemerkung unten).

Ersteres ist etwas schwieriger zu handhaben. Wir müssen uns alle Prozessnummern unserer Kinder merken, um sie am Schluss der Shell mitzubeseitigen. Wir verwenden dazu die Schablone `vector` aus der `C++`-STL. Die *Änderungen* an der Shell sind folgende:

```
...
#include <sys/signal.h>   // je nach System evtl. signal.h
#include <vector.h>

int main()
 {
   vector<pid_t> children;

   for (;;)
    {
         ...
         int pid=fork();
         if (pid==0)
         ...
         else children.push_back(pid);
     }

     int s;
     while (waitpid(-1,&s,WNOHANG)>0);
   }

   vector<pid_t>::iterator it;
   for ( it=children.begin() ; it!=children.end() ; ++it )
      kill(*it,SIGTERM);

 }
```

Bemerkung: Wir können, um Zombies *ganz* zu verhindern, *nicht* das Verfahren mit den geschachtelten `fork`s verwenden. Der Großelter würde dann ja den *Enkelprozess* in den Vektor aufnehmen wollen – aber dessen PID kennt nur das zwischengeschobene Kind, das es seinem Vater nur schwer mitteilen kann! Der Exit-Status beispielsweise darf nur Werte bis 255 haben, die PIDs werden aber meist größer sein. Die Kommunikation müsste über kompliziertere Wege gehen, die wir noch nicht kennen.

6.8 `exec`

Der Systemaufruf, der in den meisten Fällen dem `fork`-Aufruf folgt, ist ein Aufruf der `exec`-Familie. Wir haben oben schon `execlp` gebraucht, bisher aber immer die Aufruf-Argumente

aus der Kommandozeile ignoriert. Die einzelnen `exec`-Versionen unterscheiden sich gerade im Wesentlichen in der Art der Übergabe dieser Aufruf-Parameter. `execve` ist der System-Aufruf, der Rest sind Bibliotheksfunktionen, die darauf zurückgeführt werden:

UNIX/DOS

```
int execv(const char *path, const char *argv[]);
int execvp(const char *file, const char *argv[]);
int execve(const char *path, char *const argv[], char *const envp[]);
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg,...);   //,char *const envp[]);
```

Die Buchstaben haben folgende Bedeutung (Einzelheiten weiter unten):

`l`	list	die Argumente werden einzeln in der Parameterliste von `exec` übergeben
`v`	vector	die Argumente werden als ein „Vektor“-Argument (Array) übergeben
`p`	path	das Programm wird in den in `$PATH` angegebenen Verzeichnissen gesucht
`e`	environment	das Environment wird als Array übergeben und nicht vererbt

- Der Rückgabewert ist *immer* -1 und bedeutet, dass ein Fehler aufgetreten ist. Anderenfalls kehren die Funktionen nicht zurück, da ja der aktuelle Code überladen wird.

- Das erste Argument ist das auszuführende **Kommando**. Die bezeichnete Datei muss als ausführbar gekennzeichnet sein – ein binäres Executable oder eine Skript-Datei.

  `execvp`, `execlp`:

  Die ‘`p`’-Versionen suchen das Kommando automatisch in den Directories, die in der Umgebungsvariable `$PATH` angegeben sind, es sei denn, der Name ist ein absoluter Filename, beginnt also mit einem ‘`/`’. Sie verhalten sich bei der Kommandosuche also wie die Shell.

  `execv`, `execve`, `execl`, `execle`:

  Die Versionen ohne ‘`p`’ interpretieren den Befehlsnamen direkt als Dateinamen und beachten `$PATH` nicht.

- Die **Kommandozeilen-Argumente** können auf zwei Weisen übergeben werden:

  `execv`, `execve`, `execvp`:

  Die Versionen mit ‘`v`’ (vector) nehmen sie als Array von `char*` an, also analog zu `main` in `C`-Programmen. Die Anzahl wird aber nicht als Parameter angegeben, daher muss zusätzlich als letzter String im Array der Null-Pointer stehen.

  Ein Aufruf für „`ls -a ..`“ wäre in diesem Stil:

  ```
  char *argv[3]={ "ls", "-a", "..", 0 };
  execvp( "ls", argv );
  ```

  `execl`, `execle`, `execlp`:

  Die Versionen mit ‘`l`’ (list) nehmen die Argumente direkt über die Parameterliste an

und arbeiten mit dem C-Mechanismus „...“ für variabel viele Parameter. Der letzte angegebene Pointer muss der Null-Pointer sein.

Ein Aufruf für „`ls -a ..`“ in diesem Stil ist:

```
execlp( "ls" , "ls" , "-a" , ".." , 0 );
```

- Auch das **Environment** kann auf zwei verschiedene Weisen an das Programm weitergegeben werden:

  execl, execv, execlp, execvp:

  Die Funktionen ohne ‘e’ übergeben das Environment unverändert. Das ist fast immer der gewünschte Effekt.

  execle, execve:

  Die Versionen mit ‘e’ ermöglichen zusätzlich die Übergabe des neuen Environment als ein Array-Parameter. Es gibt keine Versionen mit ‘e’ *und* ‘p’, d.h. hier muss das Kommando immer mit komplettem Pfad angegeben werden.

  Beispielsweise definiert man mit folgendem Aufruf für den neuen Prozess die Variable DISPLAY:

  ```
  char *envp[]= { "DISPLAY=hostname:0.0", 0 };
  execle( "/usr/bin/env" , "env" , 0 , envp );
  perror("execle-Test");
  ```

  Vorsicht: Das Environment, das `env` ausgibt, besteht danach *nur* aus dieser *einen Variablen*!

  Beachte: Bei `execle` folgt *hinter* dem Null-Pointer der Argumente das Array mit den Environment-Angaben. Diese Semantik lässt sich nicht im C-Prototypen angeben und ist oben nur als Kommentar angedeutet.

Beispiel: Die oben schon besprochene Bibliotheksfunktion `system` ist natürlich mit Hilfe eines `exec`-Systemaufrufs implementiert. Das könnte beispielsweise so aussehen:

```
int system(const char *command)
 {
   pid_t pid;
   int s;
   switch (pid=fork())
    {
      case -1: return -1;               // fork hat nicht geklappt
      case  0: execl("/bin/sh","sh","-c",command,0);
               _exit(127);              // Fehlerwert, falls exec nicht klappt
      default: while (wait(&s)!=pid);   // Elter-Prozess wartet
               return WEXITSTATUS(s);   // und gibt den exit-Code weiter
    }
 }
```

Wir rufen also einfach die Standardshell `/bin/sh` mit der Kommando-Option `-c` auf. Um genau dasselbe zu tun wie das `system` aus der Bibliothek, müssten wir auch noch Ein- und Ausgabekanäle der beiden Prozesse entkoppeln (was wir aber erst später behandeln).

Beispiel: Wir bauen nun unsere Shell dahingehend aus, dass die Worte, die in der Aufrufzeile hinter dem Programmnamen stehen, als Parameter erkannt und dem Kommando beim `exec`-Aufruf übergeben werden.

- Wir haben es mit *beliebig vielen* Argumenten zu tun, weswegen nur ein `execv`-Aufruf in Frage kommt. Das Durchsuchen der in `$PATH` angegebenen Directories wollen wir dem System überlassen, und die Umgebungsvariablen nicht ändern. Wir verwenden also `execvp`.
- Die `C++`-Klasse `String` stellt leider keine Funktion zur Verfügung, die eine Zeichenkette in ihre Worte aufspaltet. Außerdem benötigt der `execvp`-Aufruf als Parameter ein Array von `char*`. Wir arbeiten daher weiterhin mit `char*`.
- Die Funktion `strtok`, die die Kommandozeile in Worte zerlegt, *verändert* diese Zeile durch Beendigung der Worte mit dem Nullbyte. Wir können diesen Vorgang also nur *einmal* durchführen – zwei Durchgänge (erst zählen, dann Anlegen des Arrays) ist nicht möglich.

  Um mit variabel vielen Parametern arbeiten zu können, bemühen wir wieder die `C++`-Schablone `vector`. Wenn wir die Anzahl der Parameter kennen, legen wir ein passend großes Array von `char*` an und füllen den Vektor mit einem `copy`-Aufruf in dieses Array um.
- Hier bietet es sich an, die Interpretation des '`&`'-Zeichens einzubauen. Wenn ein Argument damit beginnt, beenden wir die Interpretation der Kommandozeile (der Einfachheit halber anders als bei üblichen Shells!) und merken uns (in `bg`), dass das Kommando im Hintergrund ausgeführt werden soll (wir müssen uns also wieder die Nummer des Kindprozesses merken). Ansonsten warten wir mit `waitpid` auf das Ende des Kinds.

  Beachte: In unserer Version muss ein *Wort* der Eingabe mit '`&`' beginnen. Übliche Shells finden das Zeichen auch ohne umgebende Spaces. Wir nehmen das aber in Kauf, um `strtok` verwenden zu können.

Es folgt der vollständige Quelltext (nur ohne `#include`-Angaben):

```
int main()
 {
   vector<pid_t> children;

   for (;;)
    {
      char buffer[256];
      cout << "mysh: ";
      cin.getline(buffer,256);

      if (*buffer!=0)
       {
         char *command=strtok(buffer," ");
         if (strcmp(command,"exit")==0) break;

         vector<char *> arg_vec;
```

```
        char *s;
        bool bg=false;
        while ((s=strtok(0," "))!=0)
         {
           if (s[0]=='&') { bg=true; break; }
           arg_vec.push_back(s);
         }

        int argc=1+arg_vec.size();
        char **argv=new(char *)[argc+1];
        argv[0]=command;
        copy(arg_vec.begin(),arg_vec.end(),argv+1);
        argv[argc]=0;

        pid_t pid=fork();
        if (pid==0)
         {
           execvp(command,argv);
           cerr << "command not found!\n";
           _exit(0);
         }
        else if (!bg) waitpid(pid,0,0);
        else children.push_back(pid);

        int st;
        while (waitpid(-1,&st,WNOHANG)>0);
      }
    }

  vector<pid_t>::iterator it;
  for ( it=children.begin() ; it!=children.end() ; ++it )
     kill(*it,SIGTERM);
}
```

Wir können nun z.B. innerhalb unserer eigenen Shell mit „`forever &`“ einen endlosen Prozess im Hintergrund starten. Mit „`ps`“ finden wir seine Nummer heraus und können ihn mit einem passenden `kill`-Kommando beenden. Das `waitpid` am Ende der **`for`**-Schleife verhindert, dass er zum Zombie wird.

Auch Kommandos wie „`vi` *filename*“ lassen sich nun innerhalb unserer Shell problemlos starten. Was natürlich *nicht* funktionieren kann, sind *Wildcards*: „`ls *.tex`“ liefert einen Fehler von `ls`; es sei denn, es gibt wirklich eine Datei mit dem Namen „`*.tex`“!

6.9 Reaktion auf Signale

Ein Prozess kann selbst bestimmen, was beim Eintreffen von Signalen geschehen soll, und zwar für jedes mögliche Signal getrennt. Er kann

- das Signal ganz **ignorieren**,

- das Signal **blockieren** (es wird dann aber zwischengespeichert und ausgelöst, wenn die Blockade aufgehoben wird),
- eine eigene **Handler-Routine** angeben, die automatisch beim Eintreffen des Signals aufgerufen wird,
- **sterben** (in seinem Rückgabe-Status wird das auslösende Signal angegeben, und der Elter-Prozess kann darauf reagieren).

6.9.1 `signal` und `sigaction`

Ohne spezielle Maßnahmen des Prozesses ist kein Signal blockiert, und alle Signale werden mit Standard-Reaktionen des Kernels beantwortet: `SIGSTOP` blockiert den Prozess, `SIGTERM` bricht ihn ab, etc.

Mit dem Systemaufruf `sigaction` kann sehr genau eingestellt werden, wie auf ein Signal reagiert werden soll. Diesen Aufruf werden wir weiter unten kurz behandeln. Die einfachste, klassische Funktion (inzwischen auch Teil von ANSI-C) heißt aber `signal` (deklariert in `signal.h`):

ANSI-C
`void (*signal(int sig, void (*func)(int) ))(int);` legt die Reaktion auf das Signal `sig` fest; `func` gibt eine Standard-Reaktion oder einen eigenen Signal-Handler an

Die Definition wird verständlicher, wenn man einen Zwischentyp einführt:

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int sig, sighandler_t func);
```

Die Handler-Routinen haben den Prototyp **`void handler(int)`**. Als **`int`**-Parameter erhalten sie die Nummer des auslösenden Signals, sodass man eine Routine für mehrere Signale verwenden kann. Der Typ `sighandler_t` stellt einen Pointer auf solche Funktionen dar.

Der Funktion `signal` wird die Nummer des Signals übergeben, für das die Systemreaktion verändert werden soll, und einen Pointer auf eine Handler-Routine. `signal` gibt einen Pointer auf die bisher gültige Handler-Routine zurück, damit man sie ggf. später restaurieren kann.

Anstelle eines Pointers kann man für `func` zwei spezielle Werte übergeben:

`SIG_ERR`	(definiert z.B. als `(sighandler_t)-1`) ein Fehler soll ausgelöst werden (führt zum Prozess-Abbruch)
`SIG_DFL`	(definiert z.B. als `(sighandler_t)0`) die Default-Reaktion soll wieder eingestellt werden
`SIG_IGN`	(definiert z.B. als `(sighandler_t)1`) das Signal soll ignoriert werden

Beachte: Die Reaktion auf die Signale `SIGKILL` und `SIGSTOP` kann aus naheliegenden Gründen nicht verändert werden. `signal` liefert dann `SIG_ERR` als Rückgabewert.

Falls die Handler-Routine zurückkehrt (und nicht den Prozess z.B. mit `exit` beendet), kehrt sie an die Stelle im Prozess zurück, wo dieser durch das Signal unterbrochen wurde. Vorsicht:

Dabei werden Warte-Operationen abgebrochen, beispielsweise `sleep` oder das Warten auf das Zustandekommen von I/O!

Beachte: Der Name `signal` für die Funktion ist sehr schlecht gewählt, da er suggeriert, dass hiermit ein Signal *ausgelöst* wird. Ein besserer Name wäre `catch` o.ä. Entsprechend sollte eher der Systemaufruf `kill` den Namen `signal` tragen!

Beispiel: Wenn wir folgende Zeile in unser `forever.cpp` einbauen, können wir es nicht mehr mit `CTRL-C` abbrechen:

```
signal(SIGINT,SIG_IGN);
```

Die Zeile

```
signal(SIGINT,ctrlchandler);
```

gibt dagegen eine Handler-Routine an, die „`CTRL-C`“ ausgibt und den Prozess beendet:

```
void ctrlchandler(int s)
 {
   cout << "CTRL-C -- aborting" << endl;
   exit(0);
 }
```

Von einer anderen Shell aus können wir diese Routine auch durch ein „`kill -INT` *pid*“ auslösen.

Beachte: Wenn man mit System-Ressourcen arbeitet, die nicht (wie `malloc`-Speicher) automatisch beim Prozess-Ende freigegeben werden (FIFOs, Message-Queues, etc.), sollte man auf jeden Fall `SIGINT`, `SIGQUIT` und `SIGTERM` abfangen und diese Ressourcen wieder freigeben!

Vorsicht: Es ist von UNIX zu UNIX unterschiedlich, wie sich Signale verhalten, wenn sie einmal ausgelöst wurden. Unter System V und Linux wird beim Eintreten eines Signals automatisch wieder die Default-Reaktion eingestellt. In der Handler-Routine wird daher oft wiederum `signal` aufgerufen. Wenn zwei gleiche Signale schnell hintereinander eintreffen, kann es aber passieren, dass das erste durch den Handler behandelt wird, das zweite an die Default-Einstellungen gerät! Bei BSD-UNIX bleibt immer die alte Einstellung erhalten.

Es gibt folgende weitere POSIX-kompatible Systemaufrufe bzw. Bibliotheksfunktionen zu Signalen und ganzen Signal-Mengen:

UNIX
`int sigemptyset(sigset_t *set);` leert die angegebene Signal-Menge
`int sigfillset(sigset_t *set);` nimmt alle Signale in die Signal-Menge auf
`int sigaddset(sigset_t *set, int signum);` nimmt das Signal `signum` in die Signal-Menge auf
`int sigdelset(sigset_t *set, int signum);` löscht das Signal `signum` aus der Signal-Menge
`int sigismember(const sigset_t *set, int signum);` testet, ob Signal `signum` in der Signal-Menge enthalten ist

UNIX
`int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);` ändert die Reaktion auf das Signal `signum` über eine `sigaction`-Struktur
`int sigprocmask(int how, const sigset_t *set,sigset_t *oldset);` abhängig von `how` gibt die Signal-Menge `set` an, welche Signale blockieren, zusätzlich blockieren oder nicht mehr blockieren sollen
`int sigpending(sigset_t *set);` schreibt die schon eingetroffenen, aber noch nicht behandelten Signale in die Signal-Menge `set`
`int sigsuspend(const sigset_t *mask);` blockiert den Prozess, bis ein Signal eintrifft, wobei ausnahmsweise `mask` als Signalmaske verwendet wird

`sigset_t` ist normalerweise derselbe Typ, der intern für die Signal-Bitmasken im PCB verwendet wird, beispielsweise **`unsigned long`**.

Einige Funktionen arbeiten mit einer `sigaction`-Struktur (systemabhängig, hier aus L*i*nux):

```
struct sigaction
 {
   sighandler_t sa_handler;
   sigset_t sa_mask;
   unsigned long sa_flags;
   void (*sa_restorer)(void);
 };
```

Mit der Struktur `sigaction` kann die Reaktion „feiner“ eingestellt werden (Aufrufe der Funktion `signal` werden intern auf `sigaction`-Aufrufe zurückgeführt):

- In `sa_mask` können Signale angegeben werden, die während der Signal-Behandlung unterdrückt werden sollen.
- In `sa_flags` kann u.a. angegeben werden, ob der Signal-Handler nach Eintreffen eines Signals weiterhin eingestellt bleiben soll (`SA_RESTART`, die normale BSD-Reaktion) oder ob auf die Default-Reaktion zurückgegangen werden soll (`SA_ONESHOT`, das Verhalten von `signal`). Durch `SA_RESTART` wird außerdem in bestimmte System-Funktionen zurückgekehrt, während deren Ausführung das Signal eintraf.

6.9.2 `alarm`

Ein nützlicher Systemaufruf in diesem Zusammenhang ist `alarm`:

UNIX
`unsigned int alarm(unsigned int seconds);` Anmelden eines Timer-Requests zum Empfang eine `SIGALRM`-Signals nach `seconds` Sekunden

Er bittet das System darum, dem aufrufenden Prozess nach Ablauf von `seconds` Sekunden das Signal `SIGALRM` zu schicken. Der Prozess läuft bis dahin normal weiter. Die Länge der tatsächliche abgelaufenen Zeit bis zum Signal kann je nach Systemzustand etwas variieren.

Ein *weiterer* `alarm`-Aufruf *vor* Erhalten des Signals macht den vorangegangenen Aufruf ungültig. Ein Prozess kann nur einen `alarm`-Timer zur selben Zeit beanspruchen. `alarm(0)` nimmt einen vorangegangenen Request zurück.

Beachte: In einigen Systemen (auch L*i*nux) gibt es zwei aus BSD stammende Aufrufe `getitimer` und `setitimer`, die sich ebenfalls mit Timern beschäftigen (dazu später). `alarm` und diese Aufrufe benutzen aber denselben Timer-Mechanismus und überschreiben sich ihre Requests.

Beispiel: Wir implementieren eine Funktion `gets_to` (`gets` mit Timeout), die wie `gets` einen String einliest, aber nach einer angegebenen Anzahl von Sekunden abbricht. Bei einem Timeout wird der Null-Pointer zurückgegeben, auch wenn bereits einige Zeichen eingegeben wurden.

```
#include <unistd.h>
#include <signal.h>
#include <stdio.h>

void sigalrm_handler(int sig) { puts("<TIMEOUT>"); }

char *gets_to(char *buf, int size, unsigned int sec)
 {
   void (*oldhandler)(int)=signal(SIGALRM,sigalrm_handler);
   alarm(sec);
   buf=fgets(buf,size,stdin);
   alarm(0);
   signal(SIGALRM,oldhandler);
   return buf;
 }

int main()
 {
   char buffer[256];
   if (gets_to(buffer,256,5)==0) puts("Chance verpasst!");
 }
```

Beachte, dass die unterbrochene Aktion – der `fgets`-Aufruf – durch das Signal *abgebrochen* wird. Der Handler kehrt also *dahinter* nach `gets_to` zurück. Da `fgets` dann nicht abgeschlossen wurde, gibt es den Nullpointer zurück.

Um in anderen Zusammenhängen feststellen zu können, *ob* ein Alarm aufgetreten ist, könnte man mit einem Flag arbeiten, das in der Handler-Routine gesetzt wird.

Beispiel: Das folgende Programm `timeout` führt ein Kommando aus, das ihm als Parameter übergeben wird, bricht es aber nach einer Maximalzahl von Sekunden per `SIGALRM` ab und gibt eine entsprechende Meldung nach `stderr` aus. Seine Aufrufsyntax ist:

`timeout` *maxsec command* [*argument(s)*...]

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/types.h>

int main(int argc, char *argv[])
 {
   if (argc>2)
    {
      pid_t pid=fork();
      if (pid==0)
       {
         alarm(atoi(argv[1]));
         execvp(argv[2],argv+2);
       }
      else
       {
         int i,status;
         waitpid(pid,&status,0);
         if (WIFSIGNALED(status) && WTERMSIG(status)==SIGALRM)
          {
            fprintf(stderr,"*** timed out: ");
            for (i=2;i<argc;++i) fprintf(stderr,"%s ",argv[i]);
            fputc('\n',stderr);
          }
       }
    }
 }
```

`timeout` lässt sich beispielsweise dazu verwenden, mehrere Rechner im Netz abzufragen, von denen sich aber einige längere Zeit nicht melden könnten:

```
timeout 10 finger @wzyx99
```

6.9.3 pause

Mit folgendem Systemaufruf kann man auf irgendein Signal warten:

	UNIX
`int pause(void);`	
	blockiert den aktuellen Prozess, bis er ein Signal erhält; liefert immer -1 zurück (mit `errno=EINTR`)

Das genaue Verhalten hängt von der jeweiligen Signal-Einstellung ab. Wenn das Signal normalerweise den Prozess abbricht oder ignoriert wird, passiert das auch jetzt. Nur wenn das Signal von einem Handler bedient wird und dieser zurückkehrt, kommt der Prozess zum `pause` zurück.

Ein Server-Prozess kann sich gut mit `pause` deaktivieren. Er kann dann z.B. durch ein selbstdefiniertes Signal von einem Client aufgeweckt werden und sich dann darüber informieren, was von ihm verlangt wird.

6.9.4 sleep

Wir hatten `sleep` schon verwendet, das den aktuellen Prozess für eine angegebene Zahl von Sekunden blockiert – oder bis es durch ein Signal unterbrochen wird:

UNIX/DOS
`unsigned int sleep(unsigned int seconds);` legt den Prozess `seconds` Sekunden schlafen; gibt die Anzahl Sekunden zurück, die noch zu warten sind ($\neq 0$, falls der Aufruf durch ein Signal unterbrochen wurde)

Es handelt sich nicht um einen Systemaufruf, sondern um eine Bibliotheksfunktion, da sich die Funktionalität direkt auf `pause` und `alarm` zurückführen lässt:

```
alarm(3);
pause();
```

Hier sollte natürlich besser noch ein `SIGALRM`-Handler dazugeschaltet werden. Meist ist `sleep` über `alarm` implementiert, weswegen man die beiden in Programmen niemals gemischt verwenden sollte!

Beispiel: Wir implementieren `sleep` selbst. Die Anzahl der noch zu wartenden Sekunden ermitteln wir dabei durch einen `time`-Aufruf vor und einen nach dem `pause`:

```
static void _sleep_sigalrm_handler(int sig) { }

unsigned int sleep(unsigned int seconds)
 {
   void (*old_handler)(int)=signal(SIGALRM,_sleep_sigalrm_handler);
   time_t t=time(0);
   alarm(seconds);
   pause();
   signal(SIGALRM,old_handler);
   return seconds-(time(0)-t);
 }
```

Wir testen unsere Routine mit folgenden Hauptprogramm:

```
void sigint_handler(int sig) { }

int main()
 {
   pid_t pid=fork();

   if (pid==0)
    {
      signal(SIGINT,sigint_handler);
      printf("%d\n",sleep(3));
    }
   else
    {
      sleep(1);
```

```
      kill(pid,SIGINT);
    }
  }
```

Wie erwartet, wird „2“ ausgegeben, da der Elter-Prozess nach einer Sekunde sein Kind unterbricht. Ohne den `SIGINT`-Handler würde das Kind ganz abgebrochen und käme nie zur `printf`-Ausgabe.

6.9.5 Feinere Timer

Die internen Timer haben eine feinere Auflösung als nur Sekunden. Es gibt System-Funktionen, die im Mikrosekunden-Bereich einstellbar sind (deklariert in `sys/time.h`). Diese Timer heißen *Intervall-Timer*, da sie kontinuierlich Signale liefern können.

UNIX
`getitimer(int which, struct itimerval *value);` füllt `*value` mit Informationen über den Timer `which`

`int setitimer(int which, const struct itimerval *value,`
`struct itimerval *ovalue);`
aktiviert den Timer `which`, mit den Instruktionen in `*value`, liefert die alten Einstellungen in `*ovalue`

Vorsicht: Eventuell wird diese Genauigkeit von der Hardware gar nicht erreicht. Außerdem kann die Zeit, bis das Signal tatsächlich beim Prozess eintrifft, je nach System-Situation stark variieren.

Es gibt drei Timer, die in `which` angegeben werden können:

Timer	Signal	Bedeutung
`ITIMER_REAL`	`SIGALRM`	Echtzeit-Timer
`ITIMER_VIRTUAL`	`SIGVTALRM`	virtueller Timer (zählt nur die Zeit, die der Prozess die CPU besitzt)
`ITIMER_PROF`	`SIGPROF`	Profiling-Timer (zählt die User- und die System-Zeit, die den Prozess betreffen)

Die beiden verwendeten Strukturen sind wie folgt definiert (siehe `sys/time.h`):

```
struct itimerval
 {
   struct timeval it_interval;    // nächster Wert
   struct timeval it_value;       // Startwert
 };

struct timeval
 {
   long tv_sec, tv_usec;          // Sekunden, Mikrosekunden
 };
```

In `it_value` muss die Startzeit angegeben werden, die der Timer sukzessive *dekrementiert*. Wenn er bei 0 angekommen ist, wird das Signal ausgelöst und der Timer auf `it_interval`

gesetzt. Auf diese Weise kann man sich kontinuierlich Signale schicken lassen. Wenn der Timer (durch einen der beiden Werte) auf 0 gesetzt wird, stoppt er. Mit `it_interval`=0 erhält man also genau ein Signal.

Beispiel: Wir lassen uns 5 Sekunden lang im Halb-Sekundenabstand ein `SIGALRM` schicken:

```
void sigalrm_handler(int sig) { }

int main()
 {
   struct sigaction sigact={ sigalrm_handler, 0, SA_RESTART,0 };
   struct itimerval iv={ 0,500000,0,500000 }, ov;

   sigaction(SIGALRM,&sigact,&sigact);
   setitimer(ITIMER_REAL,&iv,&ov);

   for (int timer_dec=10;timer_dec>0;--timer_dec)
    {
      cout << timer_dec << ' ' << flush;
      pause();
    }

   iv.it_value.tv_sec=iv.it_value.tv_usec=0;
   setitimer(ITIMER_REAL,&iv,&ov);

   cout << endl;
 }
```

Unser Handler tut *nichts*; wir dürfen das Signal aber nicht ignorieren.

6.9.6 Signale in der Shell

Beispiel: Bisher wird in unserer Shell (Seite 148) durch ein `CTRL-C` die *Shell selbst* abgebrochen, nicht nur das von ihr gerade ausgerufene Programm. *Alle* Prozesse, die mit dem entsprechenden Terminal gekoppelt sind, erhalten das `INT`-Signal.

Wir verhindern das, indem wir vor der Eingabe das Signal `INT` ignorieren lassen.

```
for (;;)
 {
   signal(SIGINT,SIG_IGN);
   ...
         signal(SIGINT,SIG_DFL);
         execvp(command,argv);
         ...
```

Das Signalverhalten wird vom Kindprozess geerbt. Nach `exec` werden alle Signale wieder auf `SIG_DFL` gesetzt, es sei denn, sie standen auf `SIG_IGN`. Damit `CTRL-C` für das Kind wieder durchgelassen wird, stellen wir die Signalbehandlung vor dem `exec` daher wieder auf den Standard zurück.

Beispiel: Bei unserem Vorgehen werden auch Kinder abgebrochen, die wir mit '`&`' in den Hintergrund geschoben haben (alle Prozesse der Prozessgruppe erhalten das Signal). Es ist nicht ratsam, `SIGINT` im Kind weiterhin zu ignorieren (dann funktioniert das `kill`-Kommando nicht).

Am besten koppeln wir Hintergrundprozesse von der Shell ab, indem wir sie (vor dem `exec`) zum Führer einer neuen Gruppe machen:

```
if (bg) setpgid(0,0);
execvp(command,argv); ...
```

Mit einem Signal können wir auch ein unerwünschtes Verhalten verhindern, das wir bei unseren bisherigen Versionen erhalten können. Wenn beim Start von Hintergrundprozessen der Elter-Prozess nach dem `fork` zuerst an die Reihe kommt, gibt er das nächste Prompt aus. Wenn das Kind sein Kommando nicht ausführen kann, meldet es *danach* erst „`command not found`“, und diese Meldung erscheint hinter dem nächsten Prompt.

Wir lassen daher das Kind bei Hintergrundprozessen ein Signal `SIGUSR1` an den Elter zurückschicken, der dieses erwartet, bevor er den nächsten Prompt ausgibt. Wenn gerade zwischen dem `signal` und `execvp` ein Task-Switching stattfindet, kommt der Prompt doch wieder zu früh. So etwas können wir mit unseren bisherigen Mitteln noch nicht verhindern.

```
void sigusr_handler(int) { }
...

 for (;;)
    {
      signal(SIGUSR1,sigusr_handler);
      ...
      pid_t pid=fork();
         if (pid==0)
          {
            if (bg) { setpgid(0,0); kill(getppid(),SIGUSR1); }
            else signal(SIGINT,SIG_DFL);
            execvp(command,argv);
            ...
          }
         else if (!bg) waitpid(pid,0,0);
         else { pause(); children.push_back(pid); }
...
```

In den meisten Shells gibt es Möglichkeiten, auf Signale zu reagieren, die die *Shell* selbst erhält. Bei der Bourne-Shell (und Kompatiblen) ist das das Kommando `trap`, bei der `csh` heißt es `onintr`. Bei ersterer gilt:

`trap command signal(s)`

Bei Eintreffen eines der angegebenen Signale wird ab jetzt das Kommando `command` ausgeführt, z.B.

```
trap "echo \<CTRL-C\>" SIGINT
```

Das Unterbrechungssignal löst ab jetzt nur die Ausgabe „`<CTRL-C>`“ aus (nicht nur in Skripten).

`trap command EXIT`

Bei der Beendigung der Shell wird `command` ausgeführt – sowohl beim regulären Ende eines Shell-Skripts, wie auch bei Abbruch durch ein Signal (außer `SIGKILL`), z.B.:

```
trap "echo ciao!" EXIT
```

`trap command DEBUG`

So wird `command` *nach jedem Befehl* ausgeführt.

`trap signal(s)`

Ohne Kommando setzt `trap` die Reaktionen auf die angegebenen Signale (oder `EXIT` oder `DEBUG`) auf die Standard-Einstellungen zurück.

6.10 I/O-Umlenkung

Nach einem `fork` teilen sich Elter- und Kindprozess zunächst Ressourcen wie geöffnete Dateien. Mit den Systemaufrufen `dup` und `dup2` kann man File-Deskriptoren modifizieren, d.h. zum Beispiel `stdin` und `stdout` des Kindprozesses mit anderen Dateien koppeln.

UNIX/DOS
`int dup(int old_fd);` kopiert den angegebenen File-Deskriptor, verwendet die kleinstmögliche Zahl als Deskriptor der Kopie und gibt ihn zurück (-1 für Fehler).
`int dup2(int old_fd, int new_fd);` kopiert den angegebenen File-Deskriptor, verwendet `new_fd` als Deskriptor der Kopie (schließt ggf. vorher `new_fd`), gibt `new_fd` (oder -1) zurück.

Genauer gesagt, es wird *keine Kopie* der internen Datei-Verwaltungsstruktur angelegt (Schreib-/Lese-Position, Flags, etc.). Vielmehr existieren nach dem `dup`-Aufruf in der Tabelle offener Dateien des Prozesses *zwei identische Pointer* auf dieselbe Verwaltungsstruktur. Änderungen über den einen Deskriptor spiegeln sich also bei Zugriff über den anderen Deskriptor wieder. Damit die Datei geschlossen wird, muss für *beide* Deskriptoren ein `close` ausgeführt werden.

`dup2` überschreibt dabei den Filedeskriptor `new_fd`. Falls er bereits in Gebrauch ist, wird die vorher damit bezeichnete Datei geschlossen. Gute Kandidaten für `new_fd` sind also 0, 1 und 2.

Beispielsweise kann man wie folgt die Standardausgabe umlenken:

```
int fd=open("new_file.txt",O_CREAT|O_TRUNC|O_WRONLY,0644);
if (fd<0) ...                    // File kann nicht geöffnet werden
else if (dup2(fd,1)<0) ...       // Umlenkung klappt nicht
else close(fd);
```

Nach diesem Codestück gehen alle Ausgaben, die sonst nach `stdout`. Man beachte, dass `dup` nur mit den Low-Level-Dateioperationen `open`, `close`, etc. (deklariert in `fcntl.h`) und nicht mit `fopen`, etc. arbeitet!

Beispiel: Die Mechanismen für die I/O-Redirection mit '`<`' und '`>`' in der Shell bedienen sich `dup`-Aufrufen.

Wir ergänzen unsere Shell (zuletzt Seite 158) erneut, sodass sie nun '`<`'- und '`>`'-Zeichen beachtet und die direkt oder als eigenes Wort folgende Angabe als Dateinamen interpretiert.

```
...
char *s, *redir[2];
redir[0]=redir[1]=0;
while ((s=strtok(0," "))!=0)
 {
   if (s[0]=='&') { bg=true; break; }
   if (s[0]=='<' || s[0]=='>')
    {
      int redir_num=(int)(s[0]=='>');
      if (s[1]==0)
       {
         if ((s=strtok(0," "))==0) { cerr << "illegal redir\n"; break; }
       }
      else ++s;
      if (redir[redir_num]!=0) cerr << "multiple redir\n";
      else redir[redir_num]=s;
    }
   else arg_vec.push_back(s);
 }
...
int pid=fork();
if (pid==0)
 {
   static int flags[2]={ O_RDONLY, O_CREAT|O_TRUNC|O_WRONLY };
   if (bg && redir[0]==0) redir[0]="/dev/null";
   for (int i=0;i<=1;++i)
      if (redir[i]!=0)
       {
         int fd=open(redir[i],flags[i],0644);
         if ( fd<0 || dup2(fd,i)<0 ) cerr << "redirection error\n";
         close(fd);
       }

   execvp(command,argv);
   ...
 }
```

Falls ein Prozess mit '`&`' gestartet und seine Eingabe nicht umgelenkt wurde, kann es passieren, dass sich Shell und dieser Prozess um Zeichen aus der Eingabe streiten. Daher ist es üblich, die Standard-Eingabe solcher Hintergrundprozesse mit `/dev/null` zu verbinden.

6.11 Pipes

In der *Shell* haben wir Pipes und die entsprechenden Symbole „`|`“ und „`|&`“ schon kennengelernt. So wird aber immer nur die Standard-Ausgabe (plus evtl. der Fehlerkanal) mit der Standard-Eingabe eines anderen verbunden.

Eine *allgemeine* Pipe ist nicht auf diesen speziellen Fall beschränkt. Sie ist eine dateiähnliche Verbindung, meist (aber nicht notwendigerweise) zwischen zwei Prozessen. Irgendein schreiben-

der Kanal des einen Prozesses wird mit irgendeinem lesenden Kanal des anderen verbunden. Der erste verhält sich dabei als „Producer“, erzeugt also Daten, die der zweite als „Consumer“ verbraucht.

Pipes sind sicherlich die einfachste Art, zwischen zwei Prozessen Daten auszutauschen. Diese Art von Prozesskommunikation ist allerdings ziemlich eingeschränkt. Mit ausgefeilten Mechanismen des Datenaustauschs und der dafür ggf. notwendigen Synchronisation beschäftigen wir uns später in einem eigenen Kapitel.

6.11.1 `pipe`

Für das Anlegen der Kommunikationskanäle auf Systemebene ist der Systemaufruf `pipe` (deklariert in `unistd.h`) verantwortlich:

UNIX
`int pipe(int fildes[2]);` legt eine Pipe an und füllt das Array `fildes` mit zwei passenden File-Deskriptoren

Über die File-Deskriptoren kann die Pipe danach angesprochen werden. `fildes[0]` dient zum Lesen aus der Pipe, `fildes[1]` zum Schreiben in die Pipe (analog zu `stdin`=0, `stdout`=1). Wenn man *bidirektionale* Kommunikation braucht, muss man *zwei* Pipes öffnen.

Ein Analogon bei der Kommunikation zwischen verschiedenen *Rechnern* ist übrigens die BSD-Funktion `socketpair`.

- Zeichen, die man nach `fildes[1]` schreibt, kann man nach FIFO-Art aus `fildes[0]` lesen. Man kann in einer Pipe nicht wie bei normalen Dateien mit der Schreib-/Leseposition herumwandern (etwa mit `seek`). Einmal gelesene Zeichen werden aus der FIFO entfernt und sind verloren.

- Sinnvoll wird das ganze natürlich erst, wenn ein Prozess das Schreiben und ein *anderer* das Lesen übernimmt.

  Wenn ein Prozess aus der Pipe zu lesen versucht, wenn sie gerade leer ist, wird er bis zum Eintreffen eines Zeichens (oder EOF) blockiert. Auch beim Schreiben kann ggf. blockiert werden, nämlich wenn die Pipe sehr voll ist, d.h. die interne Maximalgröße erreicht ist. Die Pipe muss dann erst wieder ein wenig leergelesen werden.

- Ein Prozess kann nicht die ganze Pipe schließen, wohl aber mit einem `close` die beiden Kanäle, die sich ja wie Dateien verhalten. Wenn kein Prozess mehr die Schreib-Hälfte der Pipe geöffnet hat, blockiert ein Lesen aus der Pipe nicht mehr, sondern liefert ein End-of-File (`read` liefert 0 zurück).

  Wenn es niemand mehr gibt, der aus der Pipe lesen könnte, werden danach geschriebene Daten weggeworfen, und der schreibende Prozess erhält das Signal `SIGPIPE`. Wenn dieses Signal ignoriert wird (was voreingestellt ist), schlägt der `write`-Befehl fehl (Rückgabewert -1, mit `errno=EPIPE`).

  Man sollte immer darauf achten, nicht mehr benutzte Kanäle einer Pipe zu schließen, da sie sonst eventuell unnötig im System verbleiben!

Beachte: Die größte Einschränkung von Pipes ist, dass die kommunizierenden Prozesse sich die *Filedeskriptoren teilen* müssen. Dazu benötigen sie einen *gemeinsamen Vorfahr*, der die Pipe anlegt, und von dem sie die Deskriptoren erben. Pipes sind also nur zwischen Elter- und Kind-Prozess, zwischen Bruder-Prozessen, etc. möglich. **Server**, die *beliebigen* anderen Prozessen Daten liefern, sind so nicht realisierbar!

Beispiel: Wir erzeugen mit `pipe` eine Pipe und verdoppeln danach mit `fork` unseren Prozess. Genau wie offene Dateien werden auch Pipes geerbt, d.h Elter und Kind haben beide Zugriff auf die Pipe. Der jeweils nicht gebrauchte Kanal wird sofort geschlossen (Eingabe beim Kind, Ausgabe beim Elter).

Danach schreibt das Kind die Zahlen 1 bis 20 in die Pipe (jeweils als mit Linefeed abgeschlossene *Zeile*), und der Elter kopiert einfach *zeichenweise* aus der Pipe auf die Standard-Ausgabe.

Da das Kind nur schreibt und der Elter nur liest, schließen wir zu Beginn den jeweils nicht benötigten Kanal. Wenn das Kind fertig ist (**return**) wird seine Seite der Pipe geschlossen. Der Elter erhält dann irgendwann beim Lesen ein End-of-File, d.h. `read` liefert 0 als Anzahl gelesener Zeichen zurück. Dann beendet sich der Elter.

```
int main()
 {
   int fildes[2];
   pipe(fildes);

   if (fork())
    {
      close(fildes[1]);

      char c;
      while (read(fildes[0],&c,1)>0) write(1,&c,1);

      int s;
      waitpid(-1,&s,0);
    }
   else
    {
      close(fildes[0]);
      for (int i=1;i<=20;++i)
       {
         char buf2[8];
         sprintf(buf2,"%d\n",i);
         write(fildes[1],buf2,strlen(buf2));
       }
    }
   return 0;
 }
```

Wenn man Pipes, wie man sie aus der Shell kennt, darstellt, ist es sinnvoll, dass der ursprüngliche Prozess der *letzte* in der Kette ist und das zuletzt erzeugte Kind das *erste*. Normalerweise beendet sich ein Prozess, wenn seine Standard-Eingabe zur Neige gegangen ist, d.h. wenn der

Prozess eins weiter links in der Kette sich beendet hat. Das Prozess-Sterben schreitet also von links nach rechts fort.

Beispiel: Wir simulieren eine Pipe-Konstruktion wie beim '|' in der Shell. Wir wollen folgende Filter-Kombination mit einem Programm zusammenbauen:

```
grep "^\ *#"  |   sed -e "s/^\ *//" -e "s/^#\ */#/"
```

Wir nennen unser Programm `praep`. Ein Aufruf wie „`cat *.c | praep`" würde alle Präprozessorzeilen der `C`-Quelltexte im aktuellen Verzeichnis ausgeben. Spaces vor und nach dem '`#`' werden dabei eliminiert.

Der *Elter*-Prozess ist der, der *rechts* vom '|' steht, der *Kind*-Prozess der *linke*. Beide Prozesse überschreiben sich mit `execlp` durch ein anderes Programm, der Elter durch `sed`, das Kind durch `grep`.

Nun schreibt aber `grep` auf seine *Standard*-Ausgabe, und `sed` liest von seiner *Standard*-Eingabe. Wie müssen mittels `dup2` diese Kanäle mit unseren Pipe-Kanälen verbinden!

```
int main()
 {
   int fildes[2];
   pipe(fildes);

   if (fork()==0)
    {
      close(fildes[0]);
      dup2(fildes[1],1);
      execlp( "grep" , "grep" , "^\\ *#" , 0 );
    }
   else
    {
      close(fildes[1]);
      dup2(fildes[0],0);
      execlp( "sed" , "sed" , "-e" , "s/^\\ *//" , "-e" , "s/^#\\ */#/" , 0 );
    }
 }
```

Jeder Prozess schließt vor dem `execlp` *zwei* Kanäle: den nicht benötigten der Pipe und *implizit* den durch `dup2` überschriebenen!

Bemerkung 1: Wir bauen hier Pipes nicht in unsere eigene Shell ein, da die Handhabung beliebig vieler Prozesse den Quelltext zu unübersichtlich macht. Das Prinzip ist aber genau das im vorangegangenen Beispiel beschriebene.

Die Aufrufe der Einzelprozesse könnten wieder mittels `vector` gespeichert werden. Beim Untersuchen der Aufrufzeile dürften wir aber beispielsweise nicht mehr `strtok` benutzen. Wir müssten ja auch bei '`;`' und '`&`' unterteilen, die Information über das jeweilige Trennzeichen geht bei `strtok` aber verloren.

Am besten definiert man die Syntax einer solchen Kommandozeile in Form einiger grammatischer Regeln und baut dann (per Hand oder mit einem Parser-Generator) zunächst einen

vernünftigen Parser. Die technische Seite der Prozess-Erzeugung sollte erst danach darauf aufgesetzt werden.

Bemerkung 2: In L*i*nux werden Pipes durch das normale Dateisystem VFS (dazu später) dargestellt. Die Daten werden werden aber nicht auf einem Sekundärspeicher, sondern mit Hilfe von `mmap` (siehe 3.2.1) im Hauptspeicher gelagert, in einem von beiden Prozessen zugreifbaren Segment.

6.11.2 `popen`

Es gibt in der C-Bibliothek (deklariert in `stdio.h`) eine weitere nützliche Funktion, die mit Pipes arbeitet:

UNIX
`FILE *popen(const char *command, const char *type);` die Shell `bin/sh` führt den String `command`, als Befehlszeile interpretiert, aus und koppelt seine Ein- oder Ausgabe an eine Pipe
`int pclose(FILE *stream);` schließt eine mit `popen` geöffnete Pipe

Der Aufruf ist also mit dem `system`-Aufruf verwandt. Es wird aber intern automatisch eine Pipe angelegt und der Prozess mit `fork` dupliziert, bevor die Shell gestartet wird. Entweder die Eingabe oder die Ausgabe der Shell wird mit der Pipe gekoppelt. Dafür setzt man `type="w"` (zum *Schreiben* in die *Eingabe* des Programms) bzw. `type="r"` (zum *Lesen* aus seiner *Ausgabe*).

Über den zurückgelieferten File-Pointer kann man dann z.B. die Ausgabe des Programms zeichenweise lesen und bearbeiten. Am Schluss schließt man diese Art von Pipe mit `pclose` auf diesen Pointer (*nicht* `fclose` verwenden)!

Beispiel: Wir führen in einem kurzen C-Programm `ls` aus, numerieren die angezeigten Dateien dabei aber durch.

```
#include <stdio.h>

int main()
 {
   int c,line=0,numout=1;
   FILE *pipe=popen("ls","r");

   while (c=fgetc(pipe), c!=EOF)
    {
      if (numout) { printf("%d: ",++line); numout=0; }
      putchar(c);
      if (c=='\n') numout=1;
    }
   pclose(pipe);
 }
```

Die Konstruktion mit dem `numout` ist notwendig, damit wir am Ende nicht eine Zeilennummer zuviel ausgeben.

7 Scheduling

Der Scheduler ist der Teil des Betriebssystems, der bei einem Prozesswechsel und mehreren lauffähigen Prozessen denjenigen unter ihnen auswählt, der die CPU als nächstes erhält. Er sollte dabei

- die CPU- und I/O-Ausnutzung maximieren
- die Zeit gering halten, die bereite Prozesse in der Ready-Queue verbringen (*waiting time*)
- die Betriebsmittel fair vergeben – nicht unbedingt alle Prozesse gleich behandeln, aber jedem die Chance geben, seine Aufgabe zu erfüllen
- akzeptable Antwortzeiten bei interaktivem Betrieb garantieren, im Ernstfall längere, aber gleichmäßige Antwortzeiten
- bei Überlast (sehr viele Prozesse) nicht zusammenbrechen, sondern auf sinnvolle Weise die Performance beschränken

Zunächst eine Warnung: Die früher eingeführten „Warteschlangen“ (Ready Queue, Waiting Queue, Seite 88, Device Queue, Seite 94) sind im Allgemeinen keine echten FIFO-Schlangen (first-in-first-out). Prozesse bekommen also nicht unbedingt in der Reihenfolge ihre Betriebsmittel zugeteilt, wie sie sie angefordert haben.

Es liegt in der Verantwortung des Schedulers, einen Prozess aus der Schlange auszuwählen, der nach bestimmten Bewertungskriterien (s.u.) als der *günstigste* im Hinblick auf CPU- oder Geräte-Ausnutzung angesehen wird.

7.1 Scheduling Level

Auf einigen Systemen wird unterschieden zwischen High, Medium und Low Level Scheduling:

High Level Scheduling: Nur in Batch-Systemen. Hier wird entschieden, welche neuen Jobs als echte Prozesse in das System integriert werden und in die Ready Queue gelangen (ansonsten warten sie zunächst auf Platte). Dieser Scheduler arbeitet in Minuten- oder Stunden-Bereichen.

Medium Level Scheduling: Nur bei Systemen, die Prozesse aus dem Hauptspeicher für längere Zeit wieder auf Platte auslagern. Das entspricht im Bild auf Seite 88 den Übergängen an den Extra-Zuständen *Suspended-Blocked* und *Suspended-Ready*. Dieser Scheduler arbeitet im Minuten-Bereich.

Low Level Scheduling: Das ist der wichtigste und komplexeste Scheduling-Mechanismus, der jedesmal in Aktion tritt, wenn ein Prozess I/O tätigt oder eine Zeitscheibe beendet ist, also ggf. Dutzende oder Hunderte Mal in der Sekunde.

Manchmal werden für die letzten beiden Ebenen auch die Begriffe „**Long Term Scheduling**“ und „**Short Term Scheduling**“ verwendet.

7.2 Bursts

Das typisch beobachtete Verhalten eines Prozesses ist folgendes: er durchläuft abwechselnd

- Phasen reiner CPU-Benutzung („**CPU Burst**“) und
- Phasen von I/O-Operationen (und Warten auf deren Beendigung, „**IO Burst**“)

Ein Prozess beginnt und beendet sein Leben natürlich mit einem CPU-Burst.

Bei den CPU-Bursts findet man meistens viele relativ kurze und wenig lange. Die Anteile der verschiedenen Bursts an der Gesamtzeit variieren von Prozess zu Prozess. Die absoluten Laufzeiten sind abhängig von den Eingabedaten, von den angesprochenen Geräten und sind nicht allgemein vorhersehbar.

Oft kann man Prozesse aber grob einteilen in CPU-intensive („**CPU bound**“), I/O-intensive („**I/O bound**“) und gemischte. Prozesse können ihr diesbezügliches Verhalten auch dynamisch ändern, worüber das System Buch führen kann.

Eine interessante Möglichkeit ist es, das *bisherige Verhalten* eines Prozesses statistisch zu erfassen und mitzuspeichern. Daraus kann der Scheduler versuchen, ein wahrscheinliches zukünftiges Verhalten abzulesen. Diese Information kann er dazu verwenden, das System zwischen CPU- und I/O-intensiven Prozessen auszubalancieren.

7.3 Präemptives Multitasking

Je nach Eingriffsmöglichkeit des Schedulers in den laufenden Prozess ist das Multitasking, das er realisiert, **präemptiv** (preemptive) oder **nicht präemptiv** (non-preemptive):

präemptiv: engl. „*preemptive*“, von *preemption*=Vorkaufsrecht, von lat. praeëmere=im voraus pachten.

Dem aktiven Prozess kann jederzeit (z.B. durch ein Timer-Signal oder eine beendete I/O-Operation) die CPU entzogen werden

nicht präemptiv: CPU-Bursts werden nie unterbrochen. Ein einmal aktivierter Prozess läuft also so lange weiter, bis er freiwillig die CPU-Kontrolle abgibt oder auf eine I/O-Operation warten muss.

Für sinnvolles Verhalten, insbesondere bei Vorhandensein interaktiver Prozesse, ist präemptives Multitasking absolut unabdingbar. Bei nicht-präemptivem Multitasking legt außerdem ein aktiver Prozess in einer Endlosschleife das gesamte System lahm.

Nicht-präemptives Multitasking nutzt oft die I/O-Geräte nicht optimal. I/O-intensive Prozesse leiden unter CPU-intensiven. Oft ist die Ready-Queue voller Prozesse, die auf eine I/O-Operation gewartet hatten, nun aber nicht an die Reihe kommen können, weil der aktuelle Prozess sich während eines langen CPU-Bursts nicht unterbrechen lässt. Ihre nächsten I/O-Operationen werden also hinausgezögert, und die Geräte bleiben in dieser Zeit ungenutzt.

7.4 Scheduling-Strategien

7.4.1 Scheduling-Kriterien

Zusammenfassend sind wichtige Größen, die in die Scheduling-Entscheidung eingehen können:

- die **Umgebung** des Prozesses (interaktiv, Batch-Prozess, Echtzeit-Notwendigkeit)
- das **Verhalten** des Prozesses (CPU-intensiver, I/O-intensiver)
- die **erwarteten Betriebsmittel** (geschätzte CPU-Zeit, Speicher, Dateien, Geräte; früher bei Batch-Systemen waren diese Angaben Teil des Jobs)
- die **bereits benötigten Betriebsmittel** (schon verbrauchte Zeit, etc.)
- die **Wartezeit** (Zeit, die ein Prozess bereits in der Ready Queue verbracht hat)
- die **Priorität** des Prozesses (z.B. Wichtigkeit, Alter, mehr s.u.)

Als Nächstes beschäftigen wir uns mit verschiedenen bekannten Strategien des CPU-Scheduling.

7.4.2 First-Come, First-Served (FCFS)

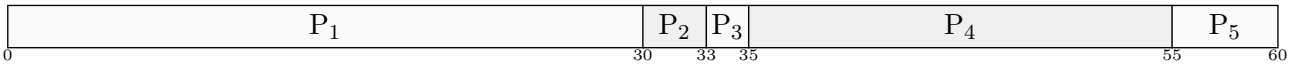
Das ist das einfachste Vergabeschema. Es wird selten alleine benutzt, meist nur in Verbindung mit anderen Schemata (s.u.).

- Die Prozess-Warteschlange ist tatsächlich eine **FIFO**-Schlange.
- Die Prozesse erhalten die CPU-Kontrolle in der Reihenfolge ihrer Anfragen, d.h. der Prozess wird aktiv, der am längsten gewartet hat.
- Das Multitasking ist nicht präemptiv, d.h. die Prozesse werden während eines CPU-Bursts nicht unterbrochen.

Beispiel: Wir betrachten fünf Prozesse P_1 bis P_5, die alle zum Zeitpunkt 0 bereit sind, die CPU zu übernehmen. Sie sollen die CPU-Burst-Zeiten 30, 3, 2, 20 und 5 besitzen (in irgendwelchen Einheiten, Millisekunden o.ä.) Wir lassen die I/O-Zeiten hier außer Betracht. Insbesondere soll die CPU nie idle sein.

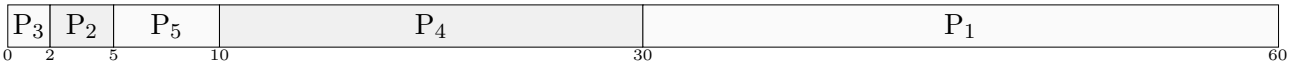
Die *Gesamtausführungszeit* für alle Prozesse ist natürlich unabhängig von jeder Scheduler-Entscheidung, nämlich die Summe der einzelnen CPU-Burst-Zeiten: 30+3+2+20+5=60.

Die *durchschnittliche Wartezeit* der Prozesse ist dagegen stark von der Reihenfolge der Ausführung abhängig (lange Prozesse vorn oder hinten):



Durchschnittliche Wartezeit: (0+30+33+35+55)/5=30.6

Durchschnittliche Wartezeit: (0+30+50+55+58)/5=38.6



Durchschnittliche Wartezeit: (0+2+5+10+30)/5=9.4

Über alle Verteilungen gemittelt ergibt sich die durchschnittliche Wartezeit 14.4.

Das FCFS-Scheduling bevorzugt *lange* Prozesse, was das Verhältnis CPU-Zeit zu Wartezeit angeht. In den obigen drei Fällen, bzw. über alle 120 möglichen Reihenfolgen gemittelt, ergeben sich folgende Werte:

	CPU-Zeit	$\mathcal{W}_1/\mathcal{C}$	$\mathcal{W}_2/\mathcal{C}$	$\mathcal{W}_3/\mathcal{C}$	$\mathcal{W}_{av}/\mathcal{C}$
P_1	30	0	0	1	15
P_2	3	10	18.3	0	28.5
P_3	2	16.5	29	1.5	29
P_4	20	1.75	1.5	0.5	20
P_5	5	11	10	1	27.5

7.4.3 Shortest-Job-First (SJF)

Hierbei wird die CPU jeweils an den Prozess vergeben, der den **kürzesten** nächsten CPU-Burst besitzt. Das kann üblicherweise natürlich nur abgeschätzt werden, sodass man nur Annäherungen an SJF implementieren kann. Bei mehreren gleich langen kürzesten Bursts wird unter ihnen per FCFS ausgewählt.

Das letzte Gantt-Diagramm bei FCFS stellte schon die SJF-Vergabe dar. Das Verhältnis von Wartezeit zu CPU-Zeit ist nun gleichmäßiger auf die Prozesse verteilt (siehe rechts).

	CPU-Zeit	Wartezeit	$\mathcal{W}/\mathcal{C}$
P_1	30	30	1
P_2	3	2	0.67
P_3	2	0	0
P_4	20	10	0.5
P_5	5	5	1

Man kann beweisen, dass SJF unter allen Auswahlstrategien die mit der kürzesten durchschnittlichen Wartezeit ist.

Beweisidee: Die durchschnittliche Wartezeit wird echt kürzer, wenn man einen kürzeren Prozess nach vorn schiebt. Genauer sei eine Verteilung folgende:

P_1	P_2	$\cdots$	P_{i-1}	P_i	P_{i+1}	$\cdots$	P_{j-1}	P_j	P_{j+1}	$\cdots$	P_n

Wenn man P_i und P_j austauscht, erhält man folgende Verteilung:

P_1	P_2	$\cdots$	P_{i-1}	P_j	P_{i+1}	$\cdots$	P_{j-1}	P_i	P_{j+1}	$\cdots$	P_n

Die Burstzeit von P_i soll mit t_i bezeichnet werden. Es ergeben sich dann folgende Zeitverschiebungen:

Prozesse P_{i+1} bis P_{j-1} ($j-i-1$ Stück)	:	je $t_j - t_i$
P_i startet später	:	$t_{i+1} + \ldots + t_j$
P_j startet früher	:	$-(t_i + \ldots + t_{j-1})$

Die Gesamtverschiebung ist also $(j-i-1)\cdot(t_j-t_i) + t_j - t_i = (j-i)\cdot(t_j-t_i)$. Für $t_j < t_i$ (kürzerer Prozess nach vorne) schrumpft die Wartezeit also echt. Die kürzeste Wartezeit ergibt sich daher bei SJF-Verteilung.

Zur Abschätzung des nächsten CPU-Bursts verwendet man die tatsächlichen Längen der vorangegangenen CPU-Bursts.

Häufig bildet man einen exponentiellen Durchschnitt. Die kurz zurückliegenden Bursts werden dabei stärker gewichtet. t_i seien die echten, τ_i die geschätzten Längen:

$$\begin{aligned}\tau_{n+1} &:= \alpha t_n + (1-\alpha)\tau_n \\ &= \alpha t_n + (1-\alpha)\alpha t_{n-1} + (1-\alpha)^2\tau_{n-1} \\ &= \ldots \\ &= \alpha t_n + (1-\alpha)\alpha t_{n-1} + \ldots + (1-\alpha)^j \alpha t_{n-j} + \ldots + (1-\alpha)^n \tau_0\end{aligned}$$

Die erste Schätzung τ_0 ist z.B. konstant für alle Prozesse. Für kleine α wirkt sich ein geändertes Burst-Verhalten nur langsam auf die weiteren Schätzungen aus, für Werte näher an 1 wird schnell auf Änderungen reagiert. Oft wählt man Werte zwischen 0.5 und 0.9.

Durch die erste Schreibweise erspart man es sich, *alle* Längen *mitzuspeichern* – es wird nur die letzte tatsächliche und die letzte Schätzung benötigt.

Beispiel: Als erste Schätzung nehmen wir 5, α sei $\frac{1}{2}$, wir runden immer auf.

Echte Burst-Zeiten 3,3,3,3,3,10,10,10,10,... liefern die Schätzungen $4, 4, 4, 4, 4, 7, 9, 10, 10, \ldots$

Echte Burst-Zeiten 3,3,3,3,3,10,3,3,3,3,... liefern die Schätzungen $4, 4, 4, 4, 4, 7, 5, 4, 4, 4, \ldots$

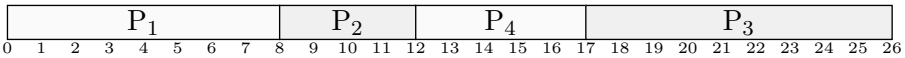
SJF kann **präemptiv** oder **nicht-präemptiv** implementiert werden. Die präemptive Version wird auch Shortest *Remaining* Time First (SRTF) genannt.

Ein Prozess, der neu in der Ready-Queue ankommt, erhält sofort die CPU, wenn er eine kürzere Burst-Zeit hat, als von der Burst-Zeit des aktuellen *noch übrig ist*.

Beispiel: Wir betrachten vier Prozesse, die *nacheinander* in der Ready Queue ankommen (siehe rechts).

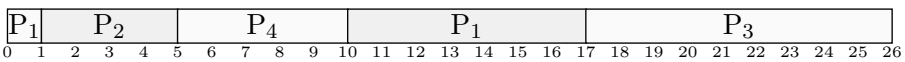
	Ankunftszeit	Burst-Zeit
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- Nicht-präemptiv erhalten wir folgendes Verhalten: P_1 startet, da es zum Zeitpunkt 0 der einzige Prozess ist, und wird nicht unterbrochen. Danach sind alle Prozesse „angekommen“ und werden nach ihrer Burst-Zeit sortiert ausgeführt.



  Die durchschnittliche Wartezeit ist $((0-0)+(8-1)+(17-2)+(12-3))/4 = 31/4 = 7.75$.

- Bei präemptivem SJF wird P_1 zum Zeitpunkt 1 *unterbrochen*. Es bleiben 7 Einheiten seines CPU-Bursts, aber Prozess 2 verspricht für sich 4 Einheiten. Später wird P_1 aber P_3 vorgezogen (7 gegenüber 9 Einheiten).



  Hier ist die durchschnittliche Wartezeit $(((0-0)+(10-1))+(1-1)+(17-2)+(5-3))/4 = 26/4 = 6.5$.

Ein Problem von SJF ist das sogenannte „**Verhungern**“ (**Starvation**). Prozesse mit relativ langem CPU-Burst kommen eventuell *nie* an die Reihe, weil immer wieder Prozesse mit kürzerem Burst bevorzugt werden. Methoden, das zu verhindern, werden weiter unten besprochen.

7.4.4 Highest Response Ratio Next (HRN)

Bei HRN handelt es sich um ein von SJF abgeleitetes Verfahren, das versucht, die unbeschränkte Benachteiligung langer Jobs zu umgehen.

Dazu wird mit dynamischen Verhaltenswerten gearbeitet, die sich jeweils aus der geschätzten Laufzeit eines Prozesses und der echten bisherigen Wartezeit errechnet:

$$R = \frac{\text{Wartezeit} + \text{Laufzeit}}{\text{Laufzeit}} = 1 + \frac{\text{Wartezeit}}{\text{Laufzeit}}$$

Der Job mit dem jeweils *höchsten* R wird vom Scheduler ausgewählt.

Alle Jobs haben zu Beginn eine Wartezeit von 0, starten also *alle* mit $R = 1$. Danach werden zunächst kürzere Jobs bevorzugt (kleinere Laufzeit im Nenner). Mit steigender Wartezeit (ohne Grenze) setzt sich diese aber irgendwann durch, sodass auch die längeren Prozesse zum Zug kommen und nicht verhungern können.

Beispiel: Zwei Prozesse P_1 und P_2 sollen die Laufzeiten 10 bzw. 100 Einheiten besitzen. Nach 5 Einheiten Wartezeit sind ihre Werte $R_1 = 1 + \frac{5}{10} = 1.5$ und $R_2 = 1 + \frac{5}{100} = 1.05$, der kürzere wird bevorzugt. Die Wartezeit des längeren Prozesses wird also stärker ansteigen, und irgendwann wird $R_2 \geq R_1$.

7.4.5 Priority Scheduling

Bei diesem Verfahren wird jedem Prozess ein numerischer Wert als „Priorität“ zugeordnet. Der Prozess mit der höchsten Priorität erhält jeweils die CPU. Es kann wiederum präemptiv oder nicht-präemptiv gearbeitet werden.

SJF erhält man als Spezialfall hiervon, wenn man als Priorität 1/Burstzeit wählt, sodass kurze Burstzeiten eine hohe Priorität zur Folge haben.

Vorsicht: Die Prioritäten werden meist durch natürliche Zahlen ab 0 dargestellt, oft aber so, dass **kleine** Zahlen für **hohe** Prioritäten stehen! Die SJF-Burstzeit könnte also direkt als eine solche Zahl verwendet werden.

Diverse Dinge können in die Vergabe der Priorität, also die Bewertung der (aktuellen) Wichtigkeit eines Prozesses eingehen:

- Zeit (Burstzeit, Verhältnis CPU-Zeit zu I/O-Zeit, etc.)
- Speicherverbrauch
- Dateizugriffe, Anzahl offener Dateien, etc.
- Externe Gründe: definierte Wichtigkeit von Prozess bzw. Benutzer, Preis der CPU-Zeit, etc.

Bei einigen Systemen kann ein Prozess seine eigene Priorität *absenken*, also andere Prozesse vorlassen. Unter UNIX gibt es dazu den „nice level“, den aber nie jemand benutzt. Der Aufruf „`nice` *`command`*“ startet beispielsweise das Programm *`command`* mit einem leicht erhöhten

Nice-Level (d.h. mit erniedrigter Priorität, systemabhängig, man kann auch explizit einen numerischen Wert angeben). Der Super-User kann den Nice-Level auch *erniedrigen*.

Ein großes Problem beim Priority Scheduling ist das Verhungern (Starvation), das wir im Spezialfall SJF erwähnt hatten. Prozesse mit niedriger Priorität erhalten eventuell *nie* die CPU, weil *immer* ein Prozess mit höhere Priorität vorliegt. (Gerüchteweise hungerte beim MIT ein Prozess einmal 6 Jahre, bis das System abgeschaltet wurde!)

Eine recht einfache und wirksame Lösung hierfür ist das Einführen des **Alterns** (**aging**). Die Priorität von Prozessen in der Ready-Queue wird allmählich erhöht, beispielsweise alle paar Minuten um 1. So ist garantiert, dass ein Prozess mit ursprünglich kleiner Priorität *irgendwann* alle anderen einholt.

7.4.6 Round-Robin-Scheduling (RR)

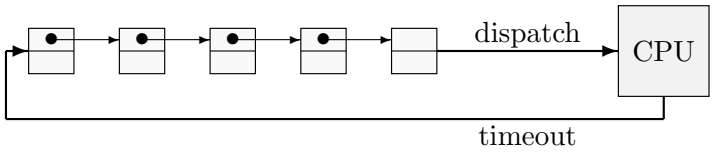
Dieses Verfahren arbeitet mit Zeitscheiben (time quanta), meist im Millisekundenbereich, die mit Hilfe eines Hardware-Timers (z.B. über Interrupts) gehandhabt werden.

Spätestens bei Ablauf einer Zeitscheibe wird der Scheduler aktiv und übergibt einem anderen bereiten Prozess die CPU-Kontrolle. Außer im Fall, dass es nur einen bereiten Prozess gibt, gehören also aufeinanderfolgende Zeitscheiben immer zu unterschiedlichen Prozessen.

Das Verfahren ist offensichtlich inhärent präemptiv. Wenn ein Prozess eher endet als seine Zeitscheibe, wird der Scheduler natürlich entsprechend früher aufgerufen.

Die Ready-Queue ist eine echte **zyklische FIFO**:

- der Prozess, der die Kontrolle abgibt, wird aus der Schlange genommen und hinten wieder angehängt,
- der neue vorderste Prozess wird der neue aktive.



Daher stammt auch die Bezeichnung:

> **round robin:** *a petition, remonstrance or the like, having the signatures arranged in circular form so as to disguise the order of signing.*

Die Qualität von RR hängt jeweils stark von der Größe der Zeitscheibe ab – und von der Hardware-Unterstützung für den Vorgang des Prozess-Umschaltens:

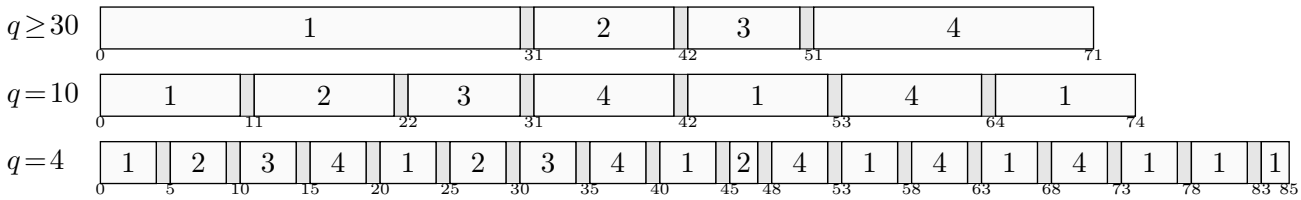
großes Quantum: wenig Zeitverlust durch Context Switches, die Prozesse müssen aber eventuell sehr lange warten (Degeneration in Richtung FCFS)

kleines Quantum: die Prozesse brauchen nicht so lange zu warten („Processor Sharing“), dafür Zeitverlust durch häufige Context Switches

Als Faustregel verwendet man, dass etwa 80 % der CPU-Bursts in einem Zeitquantum erledigt werden können.

Beispiel: Wir betrachten 4 Prozesse mit den CPU-Burstzeiten 30,10,8,20 Einheiten. Als Zeit für einen Context-Switch setzen wir 1 Einheit an, was allerdings unrealistisch groß ist.

Für Zeitquanta $q \geq 30$ erhalten wir natürlich dasselbe Verhalten wie bei FCFS, da alle Bursts in das Quantum passen. Unten sind die Gantt-Diagramme für zwei weitere Werte für q dargestellt:



Durch die zusätzlichen Context-Switches wächst die Gesamt-Laufzeit natürlich mit abnehmendem Quantum. Dagegen wird die durchschnittliche Aussetzzeit der Prozesse, also die Zeit zwischen CPU-Abgabe und CPU-Wiedererhalt, kleiner.

Am aussagekräftigsten für die Qualität eines Quantums ist aber die durchschnittliche Wartezeit. Sie ist zusammen mit den obigen Werten in der folgenden Tabelle dargestellt (gemittelt über die 4 Prozesse). Sie schwankt deutlich und erreicht ihr Minimum in unserem Fall bei $q = 10$.

Quantum	30	28	26	24	22	20	18	16	14	12	10	8	6	4	2	1
Laufzeit	71	72	72	72	72	72	73	73	74	74	74	77	80	85	101	135
Aussetzzeit	31	40	39	37	36	34	28	27	23	21	20	17	14	10	7	5
Wartezeit	31	40	39	37	36	34	36	34	34	32	**30**	38	42	43	54	79

Die 80%-Regel würde verlangen, dass 3 Bursts in ein Quantum passen, also $q = 20$.

7.4.7 Multilevel Queue Scheduling (MQS)

Wenn Prozesse in Klassen eingeteilt werden können, die sich untereinander stark in ihrem Zeitverhalten unterscheiden, lohnt es sich, mehrstufig zu arbeiten.

Solche Klassen könnten zum Beispiel folgende sein:

- Systemprozesse (Wichtigkeit)
- interaktive Prozesse (viel Tasten-I/O, kurze Antwortzeiten)
- I/O-intensive Prozesse (Datenbank-Anwendungen etc.)
- Batch-Prozesse (können bei Bedarf ein wenig liegengelassen werden)
- Klassen nach Benutzer-Hierarchie geordnet (Professoren, Studenten)

Für jede dieser Klassen gibt es eine eigene Ready-Queue.

„Außen“ wird eine Scheduling-Politik festgelegt, die jeweils eine **Klasse** auswählt, also beispielsweise festlegt, wie oft interaktive und wie oft System-Prozesse an die Reihe kommen.

„Innen“ (innerhalb jeder Klasse) kann eine eigene Politik verwendet werden, die der jeweiligen Klassen angepasst ist.

Typische Beispiele für die äußere Politik sind:

- Priority-Scheduling (Systemprozesse am dringendsten, Batch-Prozesse am wenigsten dringend)
- gewichtetes Time-Slicing zwischen den Klassen – beispielsweise wird die Batch-Klasse jedes zweite Mal übergangen, o.ä.

Ein Wechsel zwischen den Klassen kann erfolgen

- nur, wenn ein Prozesswechsel in der aktuellen Klasse stattfindet (nicht präemptiv)
- mittels Time-Slicing über die Klassen (präemptiv)
- zusätzlich, wenn ein neuer Prozess entsteht (präemptiv)

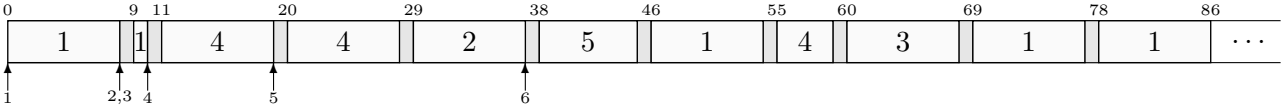
Beispiel: Wir sehen drei Klassen vor, die rechts aufgeführt sind.

Klasse 0: Systemprozesse (sofort aktiv)
Klasse 1: Wichtigkeitsstufe 1 (wichtiger)
Klasse 2: Wichtigkeitsstufe 2 (unwichtiger)

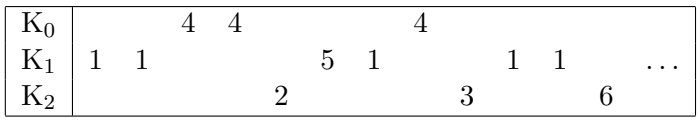
Das Scheduling soll Time-Slicing-getrieben sein, wobei die Klassen 0 und 1 doppelt soviele Slices bekommen sollen wie Klasse 2. Das Quantum soll 8 Einheiten betragen. Wir betrachten wieder nur CPU-Bursts.

Die rechts angegebenen sechs Prozesse sollen nacheinander entstehen.

	Klasse	Start	CPU-Zeit
P_1	K_1	0	30
P_2	K_2	8	20
P_3	K_2	8	30
P_4	K_0	10	20
P_5	K_1	19	7
P_6	K_2	37	25



Wenn man die Vergabe der Time-Slices an die Klassen darstellt, erhält man das Bild rechts.

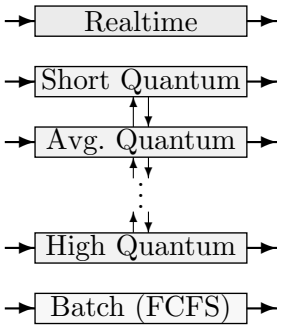


7.4.8 Multilevel Feedback Queue Scheduling (MFQS)

Dieses Verfahren ist eine Erweiterung des vorherigen, bei dem die Prozesse dynamisch zwischen den Klassen wechseln dürfen. Das System überwacht grob ihr Verhalten (daher „Feedback“) und ordnet sie ggf. neu ein.

Die Klassen werden generell nach CPU-Intensivität vergeben. Die höchsten Prioritäten erhalten dabei die interaktiven Prozesse (damit kurze Antwortzeiten garantiert sind) und die I/O-intensiven Prozesse (damit die Geräte möglichst gut ausgelastet werden). Prozesse mit langen CPU-Bursts werden tendenziell also länger liegengelassen. Das System führt dazu Buch über die Länge der zurückliegenden CPU- und I/O-Bursts.

Ein gängiges Verfahren (z.B. wird unter L*i*nux ähnlich gearbeitet) verwendet Prozess-Klassen, die sich in der Länge der ihren zugeteilten Zeitscheibe unterscheiden. Wenn ein Prozess seine Zeitscheibe aufbraucht (oder wenn er es wiederholt tut), wird er in eine Klasse mit längerer Zeitscheibe, aber niedrigerer Priorität verschoben. Wenn er dagegen aufgrund von I/O selten seine Zeitscheibe ausnutzt, kann er in die umgekehrte Richtung wandern.



Auch hier wird zusätzlich mit Altern gearbeitet, um Verhungern zu verhindern. Prozesse steigen also auch automatisch mit längerer Laufzeit allmählich in höhere Klassen auf.

Die Qualität von MFQS ist von vielen Parametern und Entscheidungen im Einzelfall abhängig:

- Anzahl der Klassen
- Politik innerhalb jeder Klasse
- Politik außen (wann rutschen Prozesse ab/steigen sie auf)
- Eintrittsklasse
- Geschwindigkeit des Alterns

7.4.9 Parametrisiertes Scheduling

Die üblichen Scheduling-Verfahren behandeln alle Prozesse als eigenständige Einheiten. Oft gibt es aber logische Zusammenhänge zwischen ihnen, die der Scheduler nicht gut überschauen kann – beispielsweise, wenn sie in einem Elter-Kind-Verhältnis zueinander stehen.

Der Elter-Prozess ist eventuell aber gut in der Lage, selbst abzuschätzen, welche seiner Kinder momentan besonders wichtig sind, und welche sich eher mehr Zeit lassen können. Damit er (begrenzten) Einfluss auf ihr Scheduling erhält, müssen die Scheduling-Verfahren **parametrisiert** werden. Bei der Erzeugung der Kinder (oder später) kann der Elter diese Parameter setzen.

- Bei einem *Priority Scheduling* ist der kanonische Parameter beispielsweise die Priorität. Es gibt Systeme, die es dem Elter erlauben, die Priorität seiner Kinder zu setzen bzw. zu verändern – meistens nicht über seine eigene Priorität hinaus, oder mit einem Faktor gekoppelt an seine eigene (eventuell wechselnde) Priorität.
- Bei einem *Multilevel Queue Scheduling* ist ein Parameter die Nummer der Queue. Ein Prozess könnte also als interaktiver Prozess ein Kind als Hintergrund-Prozess starten, einen I/O-intensiven, etc.

7.5 Echtzeit-Scheduling

Ein Echtzeit-Scheduling (*Real Time Scheduling*) ist in Systemen notwendig, die unter Einhaltung enger Zeitvorgaben auf bestimmte Ereignisse reagieren müssen, beispielsweise bei Steuerungsvorgängen. Die Maximierung von Betriebsmittel-Auslastung tritt dabei in dem Hintergrund.

Die Zeitvorgaben (z.B. als maximale Reaktionszeit) werden üblicherweise von den einzelnen Anwendungen festgelegt. Es gibt zwei Arten von entsprechenden Systemen:

Strikte Echtzeitsysteme: Ein Nichteinhalten der Zeitvorgabe hat katastrophale Folgen und ist unter allen Umständen zu vermeiden (Produktionsanlagen, Kernkraftwerke, etc.).

Schwache Echtzeitsysteme: Eine Verletzung hat negative, störende, aber keine katastrophalen Folgen (z.B. Abspielen von Video- und Musikdaten).

Es gibt zwei Arten der Steuerung von Echtzeit-Systemen:

Zeitgesteuert: Der Scheduler wird ausschließlich von Timer-Signalen gesteuert.

Ereignisgesteuert: Der Scheduler wird hauptsächlich durch externe (I/O-)Ereignisse gesteuert.

Erstere sind theoretisch und programmtechnisch einfacher zu handhaben. Das Verhalten letzterer besonders in Überlastfällen ist nicht gut vorhersehbar. Die jeweilige Anwendung kann natürlich dennoch den Einsatz ereignisgesteuerter Scheduler erzwingen.

Eine Echtzeit-Aktivität wird durch folgende Größen gekennzeichnet:

Bereitzeit (r, Ready Time), frühestmöglicher Beginn

Frist (d, Deadline), am spätesten erlaubter Abschluss

Ausführungszeit (Δe, Execution Time), Maximalabschätzung der Laufzeit

Offensichtlich gilt $\Delta e \leq d - r$.

Für periodische Aktivitäten gibt es folgende Bestimmungsgrößen:

Periode (Δp), Frequenz der Aktivität

Phase (Δh), Zeit von Periodenbeginn bis zur realen Ausführung

Ausführungszeit (Δe), Laufzeitschätzung für eine Periode

Der Scheduler plant für jede Echtzeit-Aktivität eine Startzeit s und eine Abschlusszeit $c \geq s + \Delta e$ ein. Bei nicht-präemptivem Scheduling kann die gestartete Aktivität nicht unterbrochen werden, und bei c gilt Gleichheit.

Zwei gängige Scheduler-Strategien sollen hier nur kurz angedeutet werden:

Earliest Deadline First (EDF): Die Aktivität mit der als nächstes erreichten Frist-Zeit erhält die CPU. Das Scheduling ähnelt Priority Scheduling (mit Fristen als Prioritätszahlen).

Rate Monotonic Scheduling (RMS): Dieses Verfahren ist in Reinform nur für periodische Aktivitäten geeignet. Prozesse mit hohen Frequenzen erhalten dabei hohe Prioritäten, solche mit langen Perioden niedrige.

7.6 Scheduling in realen Systemen

7.6.1 Scheduling in UNIX

Es gibt bei den verschiedenen UNIX-Geschmacksrichtungen kein einheitlich benutztes Scheduling-Verfahren. Fast alle Systeme basieren auf Multilevel Feedback Scheduling, also mit dynamischen Prioritäten, meist mit FCFS oder Round-Robin in den einzelnen Klassen.

- Die Priorität ist eine ganze Zahl im Bereich 0 bis etwa 60, wobei 0 die höchste Priorität darstellt. (BSD hat auch negative Prioritäts-Zahlen.)
- Mit steigendem CPU-Verbrauch wird die Priorität abgesenkt, um Prozesse mit kurzen Bursts nicht zu benachteiligen.
- Mit steigender Wartezeit wird die Priorität erhöht, eine Form der Alterung.

7.6.2 Scheduling bei BSD-UNIX

Die meisten BSD-Systeme arbeiten mit einer Form von Multilevel Feedback Scheduling. Dabei sollen interaktive Anwendungen bevorzugt werden. Außerdem wird versucht, möglichst wenig Prozesse lange im Kernel warten zu lassen. Prozesse, die auf schnelle I/O-Geräte warten (besonders solche, die von vielen benutzt werden, wie etwa Platten), werden besonders schnell bedient.

Die Anzahl der Teillisten ist systemabhängig, in 4.4BSD beispielsweise 32 (jeweils mit 8 Feinabstufungen durch Prioritätswerte). Es gibt dort u.a. die in der Tabelle rechts aufgeführten Klassen.

Pri.	Klasse
2	Benutzer, Priorität 2
1	Benutzer, Priorität 1
0	Benutzer, Priorität 0
-1	Kindprozess terminiert
-2	Terminalausgabe
-3	Terminaleingabe
-4	Seite auslagern
-5	auf Platte warten

Unter den Listen mit bereiten Prozessen wählt der Scheduler die mit der höchsten Priorität und teilt dem dort vordersten Prozess die CPU zu. Jede Schlange wird per Round-Robin verwaltet.

Z.B. alle 40 ms werden die Prioritätswerte korrigiert. Prozesse, die sich I/O-intensiv verhalten haben (nach SJF-ähnlicher Rechnung), steigen auf, und ein Alterungsprozess wird durchgeführt.

7.6.3 Scheduling bei UNIX SVR4

Bei SVR4 wird für spezielle Systeme auch ein Real-Time-Betrieb für ausgewählte Prozesse ermöglicht – etwa solche, die in einer Steuerung in Echtzeit auf externe Signale reagieren müssen. Dafür wird ein Multilevel-Queue-Scheduling mit zwei oder drei Klassen eingeführt:

- Timesharing-Prozesse
- Systemprozesse (nicht immer vorhanden, bevorzugt gegenüber Timesharing)
- Real-Time-Prozesse

Die ersten beiden Klassen werden mit einer Scheduling-Politik bedient wie oben angedeutet. Ein Real-Time-Prozess erhält dagegen *sofort* die CPU, wenn er *ready* wird. Sollten mehrere Real-Time-Prozesse *ready* sein, wird auf sie ein Priority-Scheduling angewandt. Das Erzeugen von Real-Time-Prozessen wird nicht jedem Benutzer automatisch gestattet.

7.6.4 Scheduling in L*i*nux

L*i*nux lehnt sich hier an SVR4 an und besitzt eine einfache Form von Real-Time-Prozessen (in sehr neuen Versionen verbesserte). Real-Time-Prozesse erhalten eine sehr viel höhere Priorität als alle „normalen“ Prozesse. Für das Priority-Scheduling zwischen Real-Time-Prozessen gibt es ein spezielles Attribut, die „relative Priorität“.

Es gibt keine Prozess-Klassen im eigentlichen Sinn und auch nur eine Ready Queue. Durch Attribute („Scheduling Policy“) werden aber doch drei Klassen realisiert, zwei für Real-Time-Prozesse (`SCHED_ERR` und `SCHED_FIFO`) und eine für normale Prozesse (`SCHED_OTHER`). Die Policy wird (im Kernel) in der Funktion `setscheduler(pid,policy)` gesetzt.

Jeder Prozess erhält einen Zähler, der beim Start auf seine vorgegebene Priorität gesetzt wird (die vom Elter geerbt wird). Bei jedem Timer-Impuls wird der Zähler des aktiven Prozesses um 1 erniedrigt. Wenn er bei 0 ankommt, hat der Prozess seine Zeitscheibe aufgebraucht, und der Scheduler wird aufgerufen.

Der L*i*nux-Scheduler wird durch verschiedene Vorgänge aktiv:

- wenn eine Prozess in einen Wartezustand gerät,
- wenn ein Prozess seine Zeitscheibe aufgebraucht hat,
- am Ende eines Systemaufrufs.

Er ordnet jedem Prozess aus der Ready-Queue ein Gewicht zu. Bei normalen Prozessen entspricht dies ihrem momentanen Zählerstand, bei Real-Time-Prozessen dem Zählerstand plus 1000. Die Gewichte von Real-Time-Prozessen sind daher immer größer als die von normalen.

Der Scheduler aktiviert den Prozess mit dem höchsten Gewicht. Haben mehrere Prozesse dasselbe höchste Gewicht, wird der vorderste (in der Queue) ausgewählt. Wenn der zuvor aktive Prozess nicht weiterläuft, wird er an das Ende der Ready-Queue angehängt.

Es folgt eine der Übersicht halber leicht gekürzte Version der Scheduler-Funktion von L*i*nux (in `kernel/sched.c`):

```
void schedule(void)
 {
   int c;
   struct task_struct *p;
   struct task_struct *prev, *next;
   unsigned long timeout = 0;

   allow_interrupts();

   if (intr_count)
    { printk("Aiee: scheduling in interrupt %p\n",__builtin_return_address(0));
      return;
    }

   if (bh_active & bh_mask)                          // Bottom-Half-Behandlung
    { ... }                                       // (vorher aufgeschobene Arbeit)

   run_task_queue(&tq_scheduler);
```

```
  need_resched=0;
  prev=current;
  cli();

  if (prev->counter==0 && prev->policy == SCHED_RR)    // Zeitscheibe zuende
   { prev->counter = prev->priority;                   // counter auffrischen
     move_last_runqueue(prev);                         // nach hinten schieben
   }

  switch (prev->state)
   {
     case TASK_INTERRUPTIBLE:
        if (prev->signal & ~prev->blocked) goto makerunnable;
        timeout = prev->timeout;
        if (timeout && (timeout <= jiffies))
         {
           prev->timeout = 0;
           timeout = 0;
     makerunnable:
           prev->state = TASK_RUNNING;
           break;
         }
     default:
        del_from_runqueue(prev);
        // fall-thru
     case TASK_RUNNING:
   }
  p = init_task.next_run;
  sti();

  prev->processor = NO_PROC_ID;

  c=-1000;
  next = idle_task;
  while (p!=&init_task)
   {
     int weight=goodness(p,prev,this_cpu);             // goodness folgt unten
     if (weight>c) { c=weight; next=p; }
     p=p->next_run;
   }

  if (c==0)                                        // alle Zähler 0, dann neu setzen
   {
     for_each_task(p)
        p->counter = (p->counter >> 1) + p->priority;
   }

  if (prev!=next)
   { ... }                                                // Prozeß verliert CPU
}
```

```
static inline int goodness
 (struct task_struct * p, struct task_struct * prev, int this_cpu)
 {
   int weight;

   if (p->policy != SCHED_OTHER)                          // Real-Time-Prozess
      return 1000 + p->rt_priority;

   weight = p->counter;
   if (weight && p==prev) ++weight;
   return weight;
 }
```

7.6.5 Scheduling in Windows

Das Scheduling unter Windows arbeitet ebenfalls mit einem Multilevel Feedback Scheduling (mit den vier Klassen *Idle*, *Normal*, *High* und *Real-Time*), durch Prioritäten weiter unterteilt.

32-Bit-Windows bietet auch Multithreading, und das hauptsächliche Scheduling findet auf Ebene der Threads statt. Es wird üblicherweise mit einer Zeitscheibe von 20 ms gearbeitet.

Durch die 16-Bit-Prozesse spielen außerdem noch einige historische Überbleibsel eine Rolle. Die genauere Behandlung wird auf das Kapitel zu Prozessen unter Windows verschoben.

8 Prozesskommunikation

Prozesse laufen fast nie isoliert ab. In Multitasking-Systemen müssen sie sich beispielsweise die Ressourcen (Betriebsmittel) teilen, und oft ist eine Zusammenarbeit mehrerer Prozesse erwünscht und sinnvoll (siehe Pipelines, Server-Prozesse, kommunizierende Rechner, etc.). Der „Interprozess-Kommunikation“ (IPC, interprocess communication) kommt daher eine große Bedeutung zu.

Zusammenarbeit zwischen Prozessen ermöglicht beispielsweise folgendes:

Modularisierung größerer Aufgaben in Teilprozesse (dadurch größere Fehlerfreiheit, leichtere Wartbarkeit, Erweiterbarkeit, etc.)

Serverprozesse, d.h. Nutzung von Daten und Diensten durch mehrere Prozesse

Verteilbarkeit auf mehrere Prozessoren

Man unterscheidet folgende Beziehungen zwischen Prozessen:

vollständige Unabhängigkeit – keine Teilung *spezifischer* Ressourcen (nur *allgemeine physische* Ressourcen wie z.B. Dateisystem, nicht *logische* Ressourcen wie eine *bestimmte* Datei) – keine besonderen Maßnahmen erforderlich

relative Unabhängigkeit – von Zeit zu Zeit Zugriff auf eine gemeinsame Ressource (wie eine Datenbank) – Sperren/Freigeben der Ressource erforderlich

gleichläufige Prozesse – konzeptionelle Zusammenarbeit – Synchronisation der Aktionen erforderlich

Die Aufteilung der physischen Ressourcen ist immer direkte Aufgabe des Betriebssystems und wurde teilweise ja schon besprochen. In diesem Kapitel wird die gleichzeitige Benutzung logischer Ressourcen wie Dateien, gemeinsam benutzte Speicherbereiche, Mailboxen etc. behandelt.

Solche Ressourcen werden in zwei Klassen aufgeteilt:

verbrauchbar – ein nur zwischenzeitlich existierendes Objekt wie ein Signal oder eine Nachricht

wiederverwendbar – ein längerfristig oder permanent existierendes Objekt wie gemeinsam benutzte Dateien/Speicherbereiche/Geräte

Daten, die mehrere Prozesse benötigen, heißen **shared data** (*shared variables*, *shared resources*). Solange ein Prozess (ggf. exklusiven) Zugriff hat, sagt man, er „*hält*“ die Ressource. Die *technischen* Möglichkeiten für zwei Prozesse, sich z.B. Speicherbereiche zu teilen, sind systemabhängig und werden für UNIX später besprochen. In diesem Kapitel geht es zunächst um die Mechanismen zur Koordination der Zugriffe.

Es gibt vier grundlegende zu lösende Probleme:

Synchronisation – eine bestimmte Operation in einem Prozess darf erst ausgeführt werden, wenn eine bestimmte andere Operation in einem anderen Prozess ausgeführt wurde

wechselseitiger Ausschluss (mutual exclusion) – nur ein Prozess darf zur selben Zeit Zugriff auf ein Objekt haben (oder auf kontrollierte Weise lesend auch mehrere)

Vermeiden von Deadlocks – Prozesse, die auf dieselben Ressource zugreifen, dürfen sich nicht gegenseitig endlos blockieren können

Vermeiden von Verhungern – ein Prozess darf nicht endlos auf seine benötigte Ressource warten müssen

8.1 Race Conditions

Es gibt *immer* Zugriffsprobleme, wenn mehrere Prozesse auf dieselben Ressourcen zugreifen können. Da sie sich sozusagen um diese Ressourcen streiten, nennt man diesen Zustand auch *race condition*.

Abhängig von der genauen zeitlichen Abfolge der Zugriffe der Prozesse auf ein Ressourcen-Objekt entstehen verschiedene ungewollte Zustände. Meist liegt die besondere Schwierigkeit darin, dass Prozesse in der Mitte eines Blocks *zusammengehöriger* Operationen auf dem Objekt *unterbrochen* werden könnten und das Objekt dadurch intern **inkonsistent** wird.

Entsprechende Fehler in Programmen sind eventuell extrem schwer zu finden, weil sich die Fehlersituation nicht genau reproduzieren lässt. Leicht unterschiedliche Startzeiten, andere Prozesse oder interne Vorgänge im System führen sofort zu Verschiebungen.

Beispiel: Als einfachste Kommunikation haben wir bereits die Pipelines kennengelernt. Zwei Prozesse stehen dort in einem einfachen Producer–Consumer-Verhältnis. Das Problem wird auch „*Bounded-Buffer Problem*“ genannt. Selbst hier könnte es aber schon Zugriffsprobleme geben – der aber durch die Systemverwaltung der Pipes verhindert wird.

Wie auch immer die interne Warteschlange implementiert ist – das Ablegen eines neuen Datums besteht aus mindestens zwei Schritten: das eigentliche Datum muss abgelegt werden, und in der Steuerstruktur der Schlange muss eingetragen werden, dass es ein Datum mehr gibt.

Es gibt Probleme, wenn der schreibende Prozess *zwischen* den beiden Operationen unterbrochen wird, die Warteschlange vorher leer war und der lesende Prozess an die Reihe kommt.

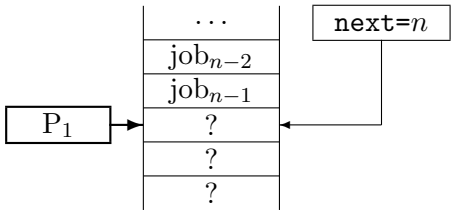
Wenn zuerst das Datum eingetragen wird, und die Steuerstruktur ist unverändert, sieht die Schlange für den Consumer leer aus, und er blockiert, obwohl Daten vorhanden sind.

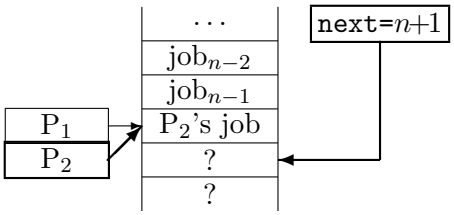
Wenn zuerst die Steuerstruktur verändert wird, scheint die Schlange für den Consumer einen Eintrag zu haben. Je nach Implementation wird auf ein altes (ungültiges) Datum zugegriffen, oder sogar ein Absturz verursacht (ungültiger Speicherzugriff). Der Consumer hinterlässt die Schlange dann leer. Beim späteren eigentlichen Eintragen des Datums durch den Producer kann es dadurch erneut Probleme geben.

Beispiel: Ein besonders beliebtes Beispiel sind Warteschlangen bei *Drucker-Spoolern*. Dort ist die Situation deswegen anders als oben, weil mehrere Prozesse in die Schlange *hineinschreiben* können.

Die Schlange sei beispielsweise als String-Tabelle implementiert, in die die Pfade der auszudruckenden Dateien eingetragen werden. Der Index des nächsten freien Eintrags sei eine für alle Prozesse zugängliche globale Variable `next`.

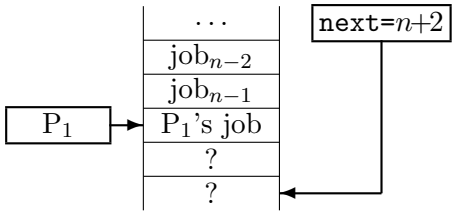
Wenn ein Prozess P_1 eine Datei ausdrucken möchte, liest er also zunächst die Nummer des nächsten freien Eintrags und ermittelt daraus die Adresse des Tabellen-Eintrags. Nun möge er gerade unterbrochen werden!





Ein Prozess P_2 kommt an die Reihe, der ebenfalls eine Datei drucken will. Er tut dasselbe wie P_1, schafft es aber im Gegensatz zu ihm, den gewünschten Dateinamen in die Tabelle einzutragen. Dann erhöht er `next`.

Irgendwann macht P_1 an der Stelle weiter, wo er unterbrochen wurde. Er schreibt also seinen Dateinamen an die vorher berechnete Adresse – und überschreibt damit den Namen, den P_2 eingetragen hatte. Dann erhöht er `next` *nochmals* und hinterlässt einen undefinierten Eintrag in der Tabelle.



Probleme gibt es also immer an Stellen, an denen eine Folge von Operationen durch einen Task-Wechsel unterbrochen werden könnte. Eine Folge, die garantiert geschlossen verarbeitet wird, heißt dagegen **atomare Operation**.

Auf den meisten Systemen sind Maschinenbefehle atomare Operationen. Auf Hochsprachen-Ebene sind nicht-atomare Operationen manchmal leicht zu übersehen, beispielsweise wenn auf einem 32-Bit-Rechner mit 64-Bit langen Integer-Zahlen gearbeitet wird (wie **`long`** in `JAVA`) und eine einfache Addition intern in zwei oder mehr Maschinenbefehle zerlegt wird!

Die System-Mechanismen im Zusammenhang mit atomaren Operationen, die wir noch genauer kennenlernen werden, sind folgende:

- **Low-Level** (hardwarenah):
  Interrupt-Abschaltung, TSL-Befehle, RMW-Befehle
- **High-Level** (Betriebssystem-Routinen):
  kritische Bereiche, Semaphore, Locks, Monitore, Message-Boxen

Die „High-Level"-Mechanismen beruhen aber fast immer auf irgendeiner Art von „Low-Level"-Unterstützung.

8.2 Kritische Bereiche

Ein Betriebssystem, das den Prozessen Möglichkeiten zur Verfügung stellt, untereinander Daten auszutauschen, *muss* dafür Sorge tragen, dass niemals Inkonsistenzen in den entsprechenden Datenstrukturen auftreten.

Wir suchen für solche Daten nach einem Mechanismus für „gegenseitigen Ausschluss“ (**mutual exclusion**), d.h. es soll nur jeweils maximal ein Prozess zur selben Zeit Zugriff auf dieselben Daten haben können.

Die allermeiste Zeit werden Prozesse mit lokalen Dingen beschäftigt sein, die ohne Shared Data auskommen. Von Zeit zu Zeit betreten sie dann einen Bereich, in dem sie mit anderen Prozessen um Daten konkurrieren könnten. Solche Bereiche nennt man **kritische Bereiche** (*critical sections*).

Vom Prinzip her sieht die Struktur eines Prozesses also wie folgt aus:

```
while (true)
  {
    entry section
    critical section      ⟵ Bereich, der auf Shared Data zugreift
    exit section
    remainder section     ⟵ Bereich, der nicht auf Shared Data zugreift
  }
```

Die grundlegende Idee ist es nun, zu fordern, dass sich immer nur maximal ein Prozess zur selben Zeit in seinem kritischen Bereich befindet. Andere Prozesse, die einen eigenen kritischen Bereich betreten wollen, müssten daran gehindert werden, bis der erste seinen Bereich wieder verlässt.

Diese Forderung verhindert das Auftreten irgendwelcher Race Conditions. Allerdings könnte das gesamte Systemverhalten sich so drastisch verschlechtern, wenn kritische Bereiche schlecht (zu allgemein, zu groß) definiert sind und unnötig Prozesse blockiert werden. Üblicherweise fordert man folgendes:

- Es dürfen nie zwei Prozesse gleichzeitig in einem kritischen Bereich arbeiten (*mutual exclusion*).
- Ein Prozess außerhalb seiner kritischen Bereiche darf keine anderen Prozesse blockieren, d.h. bei freiem kritischen Bereich darf die Entscheidung, welcher Prozess eintreten darf, nicht endlos verschoben werden (*progress*).
- Kein Prozess darf endlos lang vom Betreten eines kritischen Bereichs abgehalten werden (*bounded waiting*).

Es sind immer kritische Bereiche zweier oder mehrerer Prozesse aneinander gekoppelt, die Bezüge auf *gemeinsame Ressourcen* enthalten. Natürlich macht es keinen Sinn, nach Betreten eines kritischen Bereichs durch einen Prozess *alle* Bereiche im ganzen System für andere Prozesse zu sperren.

Real arbeitet man also vielmehr immer mit einer *Gruppe* zusammengehöriger kritischer Bereiche, die wie eine Ressource (in der Art einer Datei oder eines gemeinsamen Speicherbereichs) anzusehen ist. Sie wird beim System angemeldet und danach über eine ID-Nummer oder ähnliches angesprochen.

Win32 (also Windows95$^+$ und NT) stellt im Zusammenhang mit Threads Operationen für kritische Bereiche zur Verfügung (`EnterCriticalSection` usw.) – siehe dazu das entsprechende spätere Kapitel. Unter UNIX gibt es solche Funktionen nicht, sie lassen sich aber direkt mit Semaphoren ersetzen (auch dazu später).

8.2.1 Interrupts sperren

Wenn ein Prozess (in einem Ein-Prozessor-System) beim Betreten eines kritischen Bereichs einfach alle Interrupts sperrt und sie beim Verlassen wieder erlaubt, ist gesichert, dass er der einzige in einem kritischen Bereich ist.

Diese Lösung ist aber extrem ungeschickt, da normale Benutzerprozesse starken Einfluss auf das System haben. Bei Programmierfehlern (oder böswilliger Absicht) kann sich das System komplett aufhängen, Timer-Signale und I/O-Anfragen werden missachtet, etc.

Innerhalb des Kernels findet dieser Vorgang allerdings manchmal sehr kurzzeitig statt. Wenn wichtige interne Systemstrukturen (Prozesstabelle) auf den neuesten Stand gebracht werden, ist das oft die einzige Möglichkeit, sie konsistent zu halten. Auch zur Realisierung anderer Synchronisations-Mechanismen (wie Semaphoren) werden Interrupts kurz gesperrt.

```
while (true)
  {
    disable interrupts
    critical section
    enable interrupts
    remainder section
  }
```

8.2.2 Busy Waiting

Der Begriff *busy waiting* bedeutet lediglich, dass ein Prozess vor dem Betreten eines kritischen Bereichs ggf. auf das Eintreten einer Bedingung *warten* muss, in dem er (in einer Schleife) immer wieder diese Bedingung abtestet. Das verbraucht natürlich auch CPU-Zeit und ist nicht die allergünstigste Lösung.

Die einfachsten Methoden für gegenseitigen Ausschluss sind aber Busy-Waiting-Methoden, die wir als Einführung also dennoch zunächst betrachten werden.

8.2.2.1 Verschluss-Variablen

Reine Software-Lösungen für gegenseitigen Ausschluss sind ziemlich schwierig zu entwerfen.

Ein „**Lock**“ (Schloss) ist eine Vorrichtung irgendeiner Art, die einen oder mehrere Prozesse daran hindert, auf eine Ressource zuzugreifen oder ein Stück Code zu betreten. Wenn eine Variable als Lock verwendet wird, nennt man sie Verschluss-Variable.

Entsprechend könnten wir als einfachstes versuchen, eine gemeinsame Variable `lock` zu verwenden, die anzeigt, ob irgendein Prozess in einem kritischen Bereich ist. Wenn ein Prozess in einen Bereich eintreten will, wartet er so lange, bis `lock=false` ist. Dann setzt er `lock` auf `true` und tritt ein. Am Schluss des Bereichs setzt er `lock` zurück auf `false`.

```
global volatile bool lock=false;

while (true)                          while (true)
  {                                     {
    while (lock);                         while (lock);
    lock=true;                            lock=true;
    critical_section_0();                 critical_section_1();
    lock=false;                           lock=false;
    remainder_section_0();                remainder_section_1();
  }                                     }
```

Wir benutzen hier ein Pseudo-Schlüsselwort **global** für Prozess-globale Daten, d.h. solche, die sich alle jeweils *relevanten* Prozesse teilen. Das ANSI-C-Schlüsselwort **volatile** deutet an, dass sich der Inhalt der Variable ohne Zutun des Programms ändern kann – eben durch das System oder (hier) einen anderen Prozess.

Leider bringt diese Lösung überhaupt nichts, da die Eintritts-Sektion aus zwei zusammengehörigen Operationen besteht: dem Abfragen und dem Setzen von `lock`. Zwischen ihnen könnte aber der Prozess unterbrochen werden.

Der Prozess kann in dem Moment unterbrochen werden, wo er feststellt, dass `lock=false` ist. Ein anderer Prozess kommt an die Reihe, tritt ein und wird unterbrochen. Der erste Prozess läuft weiter und ist immer noch der Meinung, freie Bahn zu haben. Beide Prozesse sind in kritischen Bereichen. Mehrfaches Abtesten hilft genauso wenig.

8.2.2.2 Striktes Abwechseln

Wir betrachten der Einfachheit halber nur zwei Prozesse P_0 und P_1. Eine gemeinsame Variable `turn`$\in \{0, 1\}$ (zu Beginn 0) bestimmt, welcher Prozess einen kritischen Bereich betreten darf.

Vor dem Betreten seines kritischen Bereichs muss ein Prozess so lange warten, bis `turn` als Wert seine Nummer hat. Als C-ähnlicher Pseudocode sieht das für die beiden Prozesse wie folgt aus:

```
                    global volatile int turn=0;
while (true)                                    while (true)
  {                                               {
    while (turn!=0);                                while (turn!=1);
    critical_section_0();                           critical_section_1();
    turn=1;                                         turn=0;
    remainder_section_0();                          remainder_section_1();
  }                                               }
```

Diese Lösung ist absolut sicher und erzeugt keine Race Conditions. Sie ist aber völlig impraktikabel, denn sie erzwingt, dass beide Prozesse *strikt abwechselnd* ihre kritischen Bereiche betreten. Wenn P_0 das etwa doppelt so oft tun möchte wie P_1, ist P_0 die meiste Zeit nur mit Warten (also Busy Waiting) beschäftigt und verschwendet massiv CPU-Zeit!

8.2.2.3 Petersons Lösung

Die folgende sichere Lösung, die kein striktes Abwechseln erzwingt, stammt von G.L.Peterson (1981) und ist hier wiederum für nur zwei Prozesse dargestellt. Sie bewirkt dennoch sehr viel Busy Waiting!

Die Ein- und Austrittsbereiche (also die Kästen der bisherigen Bilder) seien als zwei Funktionen `enter_region` bzw. `leave_region` dargestellt (mit der Prozessnummer als Parameter):

```
global volatile int turn;
global volatile bool interested[2]={ false, false };

void enter_region(int process)                  // process = 0 oder 1
```

```
 {
   int other=1-process;                          // der jeweils andere Prozeß
   interested[process]=true;                     // Interesse anmelden
   turn=process;
   while ( turn==other && interested[other] );   // busy waiting
 }
void leave_region(int process)
 {
   interested[process]=false;                    // Interesse abmelden
 }
```

Als erstes sei P_0 interessiert. Es setzt `interested[0]=true` und `turn=0`. Da `interested[1]==false`, ist die Schleife sofort beendet, und P_0 kann seinen Bereich betreten.

Wenn in der Zwischenzeit auch P_1 interessiert ist, setzt es zwar `interested[1]=true` und `turn=1`, muss aber wegen `interested[0]==true` warten, bis P_0 fertig ist. Dann darf P_1 eintreten. P_0 kann dabei nicht dazwischenfunken, da `interested[1]==true`. Es würde also in der Schleife abgefangen.

Wenn beide Prozesse fast gleichzeitig einen kritischen Bereich betreten wollen, beschreiben sie ggf. kurz hintereinander `turn`, und der erste Wert ist verloren. Beide `interested`-Werte sind `true`. Der Prozess, der zuletzt `turn` beschrieben hat, bleibt deshalb an der Schleife hängen (Busy Waiting). Sobald der andere Prozess wieder die CPU erhält, darf er seinen kritischen Bereich betreten.

8.2.2.4 Bäckerei-Algorithmus

Für die Kommunikation von mehr als zwei Prozessen hat man den „*bakery algorithm*“ entwickelt, der an die Handhabung in (amerikanischen) Großhandels-Lagern angelehnt ist.

Jeder Kunde erhält beim Eintreten eine *Nummer*, und es wird jeweils der Kunde mit der niedrigsten Nummer bedient. Unsere Kunden sind natürlich die beteiligten Prozesse, die in den kritischen Bereich eintreten möchten. Sie erhalten jedesmal eine neue Nummer.

Die Anzahl der beteiligten Prozesse sei `N`. Dann benötigt der Algorithmus zwei Prozess-globale Arrays:

```
global volatile int number[N];              // mit 0 vorbelegt!
global volatile bool choosing[N];           // mit false vorbelegt!
```

`choosing` zeigt an, dass der jeweilige Prozess gerade erst seine Nummer erhält, `number` ist die erhaltene Nummer.

```
void enter_region(int p)
 {
   choosing[p]=true;
   number[p]=1+max(number);          // Maximum über das ganze Array
   choosing[p]=false;

   for ( int p2=0 ; p2<N ; ++p2 )
    {
```

```
        while (choosing[p2]);          // zweimal busy waiting
        while ( number[p2]!=0
           && ( number[p2]<number[p] || (number[p2]==number[p] && p2<p) ) ) ;
      }
   }

void leave_region(int p)
 {
   number[p]=0;
 }
```

Unsere Version verhindert nicht, dass zwei Prozesse (sehr selten) dieselbe Nummer erhalten, nämlich wenn die Maximums-Berechnung annähernd zeitgleich erfolgt. Daher soll in diesem Fall der Prozess von ihnen mit der kleinsten *PID* Vorrang haben. Der zweite Teil der zweiten Schleife sieht daher etwas umständlich aus.

Auf diese Weise ist der gegenseitige Ausschluss garantiert. Die erste Bedingung in der zweiten Schleife ist erfüllt bei Prozessen im kritischen Bereich oder bei solchen, die schon eine Nummer für den Eintritt erhalten haben.

Angenommen, P_i sei gerade in seinem kritischen Bereich, und P_j möchte eintreten. Wenn P_j eine echt höhere Nummer erhalten hat, blockiert er in der zweiten Schleife schon vorn wegen `number[`i`]<number[`j`]`. Wenn die Nummern übereinstimmen, blockiert die Schleife hinten wegen $i < j$, weil sie für i nicht blockiert hat.

8.2.2.5 TSL-Befehle

Diese Lösung benötigt einen speziellen Maschinenbefehl, der oft `TSL` genannt wird – Test and Set Lock. Er lädt den Wert einer Speicherstelle in ein Prozessor-Register und schreibt dann einen von Null verschiedenen Wert an diese Speicherstelle.

Auch hier haben wir es also eigentlich mit zwei Operationen zu tun. Da das ganze aber als ein Maschinenbefehl (ggf. per Mikrocode) implementiert und damit „unteilbar“ ist, ist garantiert, dass der aktuelle Prozess nicht zwischendurch unterbrochen wird.

Der Befehl lässt sich natürlich gut für wechselseitigen Ausschluss einsetzen. Ein- und Austrittsfunktion sind allerdings entsprechend diesmal in einer Pseudo-*Maschinensprache* (Intel-ähnlich) notiert. Unsere Sperr-Variable heißt wieder `lock`:

```
enter_region:                    leave_region:
   tsl reg,lock                     move lock,#0
   cmp reg,#0                       ret
   jnz enter_region
   ret
```

Die Eintrittsfunktion läuft so lange „busily waiting“, bis sie per `TSL` in `lock` eine Null vorfindet. Andere Prozesse finden dagegen immer einen anderen Wert vor, auch wenn sie direkt nach dem `TSL` an die Reihe kommen.

`TSL`-Konstruktionen werden besonders gern in Multi-Prozessor-Systemen eingesetzt.

Allgemeiner werden `RMW`-Befehle (Read-Modify-Write) betrachtet, die nicht notwendigerweise auf ein Maschinenwort beschränkt sind und bestimmte Werte zurückschreiben können. Befehle dieser Klasse in gängigen Prozessor-Architekturen sind folgende:

Test And Set: Wert auslesen, 1 zurückschreiben

Exchange: Austauschen eines Datenworts Register/Speicher

Compare And Swap: Wert einlesen, bei Übereinstimmung mit einem Register mit diesem austauschen

8.2.2.6 Priority Inversion

Ein besonders unangenehmer Effekt bei allen Methoden mit Busy Waiting ist das „Priority Inversion Problem“. Es kann in Systemen auftreten, die mit unterschiedlichen Prozess-Prioritäten arbeiten:

Eventuell erhält ein Prozess höherer Priorität die CPU, darf aber gerade nicht seinen kritischen Bereich betreten und ist nur mit Busy Waiting beschäftigt. Der Prozess, der gerade in seinem kritischen Bereich ist, kommt wegen niedrigerer Priorität aber eventuell *nie* an die Reihe! Das System hängt dann in der Warteschleife des anderen Prozesses fest.

8.2.3 Blockieren

Günstiger, als einen Prozess, der nicht in seinen kritischen Bereich eintreten darf, in einer Warteschleife hängen zu lassen, ist es, ihn in dieser Zeit aus dem Task-Switching auszunehmen, also zu blockieren.

In der einfachsten Form führt man einen Systemaufruf `sleep` ein (nicht mit der UNIX-Bibliotheksfunktion verwechseln), der einen Prozess P so lange blockiert, bis er explizit durch einen anderen Aufruf `wakeup(`P`)` „aufgeweckt“ wird. Damit kann man nicht nur kritische Bereiche schützen, sondern auch in Ausnahmefällen (wie Pufferüberlauf) warten, bis die Situation bereinigt ist und es für den Prozess Sinn macht, weiterzulaufen.

Beispiel: Die typischste Art von Zusammenarbeit zweier Prozesse haben wir schon kennengelernt – das Producer–Consumer-Verhältnis, etwa bei der Drucker-Warteschlange aus 8.1.

Die Idee ist hier, dass der Producer sich mit `sleep` schlafenlegt, wenn er eine volle Warteschlange vorfindet, und der Consumer, wenn er kein Element abholen kann. Die Prozesse wecken sich gegenseitig auf, wenn sich die Situation ändert.

Die Größe der Schlange soll durch die globale Konstante `N` festgelegt sein. Wir benötigen auf jeden Fall noch eine globale Steuervariable, beispielsweise soll `count` die Anzahl der momentan in der Schlange befindlichen Elemente aufnehmen.

- Wenn der Producer `count`=`N` liest, legt er sich schlafen. Im Fall `count`≠`N` erhöht er dagegen `count` um 1, legt sein Element ab und weckt ggf. den Consumer auf.
- Wenn der Consumer `count`=0 vorfindet, legt er sich schlafen. Ansonsten erniedrigt er `count` um 1, entnimmt ein Element und weckt ggf. den Producer auf.

Die Vorgänge stellen sich wie folgt als Pseudo-Code dar:

```
void producer()
 {
   for (;;)
    {
      item i=produce_item();
      if (count==N) sleep();
      enter_item(i);
      if (++count==1) wakeup(consumer);
    }
 }
```

```
void consumer()
 {
   for (;;)
    {
      if (count==0) sleep();
      item i=remove_item();
      if (--count==N-1) wakeup(producer);
      consume_item(i);
    }
 }
```

Die `sleep`- und `wakeup`-Aufrufe eliminieren zwar das Busy Waiting, schützen uns aber leider nicht mehr vor Race Conditions. Probleme gibt es bei Unterbrechungen zwischen dem Lesen von `count`, der Schlangen-Manipulation und dem Verändern von `count`:

- Angenommen, der Consumer liest den Wert von `count` in ein Register (er sei 0), wird dann aber unterbrochen. Der Producer fügt dann ein Element in die vorher leere Schlange ein und schickt dem Consumer sein Wecksignal.
- Der Consumer war aber noch nicht dazu gekommen, sich schlafenzulegen, und das Signal geht verloren. Irgendwann kommt der Consumer an die Reihe, vergleicht seinen Register-Wert mit 0 und kommt zu dem (falschen) Schluss, die Schlange sei leer, und legt sich schlafen – und er wird beim nächsten Mal nicht vom Producer aufgeweckt.
- Der Producer produziert nun weitere Elemente, bis die Schlange voll ist, und legt sich schlafen – in der Erwartung, irgendwann vom Consumer geweckt zu werden. *Beide* Prozesse schlafen den Dornröschenschlaf.

In unserem Fall würde es helfen, Wecksignale zwischenzuspeichern, die einen folgenden `sleep`-Aufruf einfach canceln würden. Es gibt aber Situationen, bei denen selbst beliebig viele zwischengespeicherte Signale keine Abhilfe sind.

8.3 Semaphore

Die Beobachtungen in Fällen wie dem vorangegangenen führten zur Einführung einer Art Zählvariablen für Wecksignale, sogenannten **Semaphoren** (E.W. Dijkstra 1965, englisch „*semaphore*“, Zeichenträger, von griechisch $\sigma\varepsilon\mu\alpha$=Zeichen).

8.3.1 Semaphor-Operationen

Es gibt genau zwei Operationen auf ihnen: `down` zum Herunter- und `up` zum Hinaufzählen. Grob betrachtet, übernimmt `down` die Rolle von `sleep`, berücksichtigt aber zwischenzeitlich eingegangene `up`-Aufrufe, und `up` übernimmt die Rolle von `wakeup`, zählt aber „zu viel“ eingegangene Signale mit. Die internen Werte sind nicht-negative ganze Zahlen.

`down`: Wenn der Wert des Semaphors größer als Null ist, wird er um 1 erniedrigt. Ansonsten (d.h. es wurde noch kein Signal gezählt), legt sich der aufrufende Prozess schlafen und wartet.

`up`: Wenn es einen Prozess gibt, der auf diesen Semaphor wartet, wird dieser aufgeweckt. Ansonsten wird der Wert des Semaphors um 1 erhöht (ein „überzähliges“ Signal wird gezählt).

Wie diese Operationen in bestimmten Synchronisations-Situationen genau eingesetzt werden, betrachten wir weiter unten ausführlich.

Dijkstra hatte die Operationen ursprünglich mit niederländischen Worten benannt, zunächst P für *proberen te verlagen* (versuchen zu erniedrigen) und V für *verhogen* (erhöhen). Später versuchte er es noch mit P für *paseer* (betreten) und V für *verlaat* (verlassen), wurde aber schließlich international durch `down` und `up` überstimmt.

Um Race Conditions durch Unterbrechungen zwischen den einzelnen Abläufen zu verhindern, müssen `up` und `down` **atomar** (nicht unterbrechbar) sein. Sie werden üblicherweise vom System zur Verfügung gestellt:

- In einem Einprozessor-System kann (sehr) kurzzeitig das Task-Switching unterbunden werden.
- In einem Mehrprozessor-System ohne TSL-Hardware können die beiden Operationen in einen kritischen Bereich eingeschlossen werden, für den der Zutritt über eine Software-Lösung geregelt wird (wie den Bäckerei-Algorithmus).

Offensichtlich ist es nötig, sich zusätzlich zum ganzzahligen Wert des Semaphors auch die Prozesse zu merken, die auf seine Änderung warten. Schematisch kann man sich also den Datentyp Semaphor wie unten angegeben vorstellen. Diese Implementation vermeidet negative Werte von Semaphoren.

Vorsicht – die Operationen müssen natürlich *atomar* sein, was wir aber im Pseudocode nicht gut angeben können! Man kann sich z.B. einen Aufruf „`disable_interrupts`“ am Anfang und einen Aufruf „`enable_interrupts`“ beim Verlassen der Funktionen vorstellen.

```
class semaphore
 {
   private:
      int v;
      queue<process> l;

   public:
      semaphore(int s=0) : v(s) { }

      void down()
       {
         if (v>0) --v;
         else
          {
            l.enqueue(thisprocess);
            thisprocess.block();
          }
       }
```

```
    void up()
     {
       if (l.isempty()) ++v;
       else l.dequeue().wakeup();
     }
};
```

Beispielsweise sollen zwei Prozesse abwechselnd `down` und `up` für einen Semaphor aufrufen. In der Tabelle sind für eine zufällig ausgewählte Befehls-Reihenfolge die sich ergebenden Semaphor- und Prozess-Zustände angegeben:

```
semaphore s=1;
void P_i(int i)   // i=0,1
 {
   for (;;)
    {
      s.down();
      s.up();
    }
 }
```

	s.v	s.l	P_0	P_1
	1	–	running	ready
P0: s.down();	0	–	ready	running
P1: s.down();	0	P_1	running	blocked
P0: s.up();	0	–	running	ready
P0: s.down();	0	P_0	blocked	running
P1: s.up();	0	–	running	ready
P0: s.up();	1	–	...	...

Alternativ kann man auch interne **negative** Werte zulassen. Sie geben dann automatisch jeweils die Anzahl der Prozesse in der Warteschlange des Semaphors an. Beide Versionen verhalten sich nach außen aber identisch.

```
void down(process p)
 {
   if (--v<0)
    {
      l.enqueue(thisprocess);
      thisprocess.block();
    }
 }
```

```
void up()
 {
   if (++v<=0) l.dequeue().wakeup();
 }
```

Die meisten Implementationen arbeiten mit nicht-negativen Werten. Es ist kein Verlust, den Wertebereich bei 0 „abzuschneiden“, da die Länge der Liste diese Aufgabe übernimmt.

Rechts ist angegeben, wie sich `s` mit möglichen negativen Werten verhält, wenn die Befehls-Reihenfolge von oben auftritt.

	s.v	s.l	P_0	P_1
	1	–	running	ready
P0: s.down();	0	–	ready	running
P1: s.down();	-1	P_1	running	blocked
P0: s.up();	0	–	running	ready
P0: s.down();	-1	P_0	blocked	running
P1: s.up();	0	–	running	ready
P0: s.up();	1	–	...	...

Manchmal gibt es spezielle Implementationen für Paare von Prozessen. Es kann dann (wie im Beispiel oben) höchstens der jeweils andere Prozess warten, und statt der Queue kann man einen einzigen PID-Eintrag verwenden („**private Semaphore**“).

Statt einer Queue kann z.B. auch eine Liste (ggf. mit Prioritäten) verwendet werden, sodass die Prozesse nicht nach der FIFO-Reihenfolge aufgeweckt werden.

Anstelle der objektorientierten Schreibweise `s.up()` usw. werden wir (in Anlehnung an die verbreitete Literatur) manchmal `up(s)` usw. verwenden.

UNIX stellt mit dem Systemaufruf `semop` eine Reihe von einzelnen Semaphor-Operationen zur Verfügung, die wir im nächsten Kapitel besprechen werden.

8.3.2 Einsatz von Semaphoren

Semaphore können auf zwei unterschiedliche Arten verwendet werden:

Wechselseitiger Ausschluss (*mutual exclusion*):

Sogenannte **binäre Semaphore** (Werte 0 und 1, manchmal „**Mutex**“ genannt) können zur Regelung des Eintritts beliebig vieler Prozesse in kritische Bereiche verwendet werden. Sie werden dann *mit 1 initialisiert*, beim Eintritt eines Prozesses erniedrigt, beim Verlassen wieder auf 1 erhöht. So ist gewährleistet, dass sich maximal ein Prozess in seinem kritischen Bereich befindet:

```
global semaphore mutex=1;

void enter_region(int p) { down(mutex); }
void leave_region(int p) { up(mutex); }
```

Das Pseudo-Schlüsselwort **`global`** werden wir bei Semaphoren in den folgenden Beispielen weglassen, da die Globalität in der Natur der Semaphore liegt.

Wenn ein binärer Semaphor direkt an eine Ressource gekoppelt ist und den Zugriff auf sie steuert, nennt man ihn auch ein **Lock** für diese Ressource. Die Operationen `down` und `up` werden dann meist als `acquire` (erwerben) und `release` (freigeben) bezeichnet. Semaphore sind aber allgemeiner als Locks, da man mehrere Ressourcen mit einem Semaphor verbinden kann.

Synchronisation:

Zur Steuerung der Abfolge von Operationen – beispielsweise für die Blockade der beiden Prozesse im Producer–Consumer-Problem. Hier wird nicht die *Inkonsistenz* einer Datenstruktur vermieden, sondern die sinnvolle gemeinsame Benutzung der Datenstruktur reguliert. Diese Semaphore nennt man manchmal **zählende Semaphore** (*counting* oder *counted semaphores*). Sie werden typischerweise mit 0 oder mit der Anzahl zugreifender Prozesse initialisiert.

Rechts ist angegeben, wie erzwungen werden kann, dass `process2` den Befehl B erst ausführen kann, **nachdem** `process1` den Befehl A ausgeführt hat.

```
semaphore flag=0;

void process1()        void process2()
 {                      {
   A;                     down(flag);
   up(flag);              B;
 }                      }
```

Wenn `process2` zuerst beim `down` ankommt, blockiert es, und er wird beim `up` aufgeweckt. Kommt andererseits zuerst `process1` beim `up` an, wird `flag=1`, und das spätere `down` blockiert `process2` nicht mehr.

Natürlich ist das Funktionieren eines Semaphor-Mechanismus davon abhängig, dass alle beteiligten Prozesse die Semaphore auch korrekt einsetzen, beispielsweise bei Semaphoren als Locks:

- nicht unnötig viel Zeit zwischen `down` und `up` verbringen
- vor dem Zugriff das passende `down` aufrufen
  (sonst gibt es unkontrollierte Zugriffe)
- nach dem Zugriff das passende `up` aufrufen
  (sonst bleibt die Ressource blockiert)
- das `down` nicht zweimal hintereinander aufrufen
  (sonst blockiert sich der Prozess selbst – und alle anderen!)
- das `up` nicht zweimal hintereinander aufrufen
  (sonst blockiert ein späteres `down` nicht, das es aber sollte)

Je nach Hardware sind binäre Semaphore einfacher zu implementieren als zählende. Man kann aber relativ einfach zählende mit Hilfe binärer aufbauen, nach folgendem Schema:

```
class counting_semaphore
 {
   private:
      binary_semaphore s1,s2,s3;
      int v;

   public:
      counting_semaphore(int i) : s1(1), s2(0), s3(1), v(i) { }

      void down()
       {
          s3.down();
          s1.down();
          if (--v<0) { s1.up(); s2.down(); }
          else s1.up();
          s3.up();
       }

      void up()
       {
         s1.down();
         if (++v<=0) s2.up();
         s1.up();
       }
 };
```

Der binäre Semaphor `s2` dient hier zur eigentlichen Signal-Auslösung. Um die ungeschützten Operationen mit dem Zähler `v` zu schützen, baut `s1` um diesen Mechanismus einen kritischen Bereich auf. Wenn in `down` ein `s2.down` ausgelöst wird, blockiert ggf. der Prozess – deshalb muss *zuerst* mit `s1.up` der kritische Bereich verlassen werden. Um aber parallel ablaufende `down`-Aufrufe daran zu hindern, an dieser Stelle zu unterbrechen, muss mit `s3` ein *weiterer* kritischer Bereich um die ganze Konstruktion herumgelegt werden.

Beispiel: Wir bauen beide Arten von Semaphoren in unser Producer–Consumer-Beispiel von oben ein:

- Den Zugriff auf die Schlange sichern wir mit einem binären Semaphor `mutex`.
- Die Blockierung bei voller/leerer Schlange regeln wir mit zählenden Semaphoren `count` bzw. `free`.

(Es gibt eine Lösung, die ohne wechselseitigen Ausschluss auskommt, aber mit einem weiteren speziellen Typ arbeitet, den wir hier nicht besprechen wollen.)

```
const int N=100;
semaphore mutex=1;
semaphore count=0;
semaphore free=N;

extern void down(semaphore&);
extern void up(semaphore&);
```

```
void producer()
 {
   for (;;)
    {
      item i=produce_item();
      down(free);
      down(mutex);
      enter_item(i);
      up(mutex);
      up(count);
    }
 }
```

```
void consumer()
 {
   for (;;)
    {
      down(count);
      down(mutex);
      item i=remove_item();
      up(mutex);
      up(free);
      consume_item(i);
    }
 }
```

8.3.3 Deadlocks

Man muss sehr vorsichtig bei der Reihenfolge der Semaphor-Operationen sein! Angenommen, wir *vertauschen* im Producer des letzten Beispiels die beiden `down`-Operationen.

- Wenn der Producer in seinen kritischen Bereich eintritt, die Schlange aber voll ist, blockiert er beim `down(free)`.
- Durch das vorangegangene `down(mutex)` bleibt aber dem Consumer der Weg in seinen Bereich versperrt, wo die einzige Möglichkeit läge, die Schlange zu leeren.

Wir haben einen **Deadlock** (eine „Verklemmung“) erzeugt – die beiden Prozesse blockieren sich gegenseitig.

Eine allgemeinere Deadlock-Situation sieht aus wie rechts angegeben. Es gibt zwar zu jedem `down`-Aufruf einen passenden `up`-Aufruf im jeweils anderen Prozess. Beide Prozesse warten aber an ihrem ersten `down` auf ein Signal vom jeweils anderen, das nie kommen kann.

Wie ein System versuchen kann, Deadlocks zu verhindern, besprechen wir in einem späteren Kapitel.

```
semaphore s=1,q=1;

void process1()
 {
   down(s);
   down(q);
   ...
   up(s);
   up(q);
 }
```

```
void process2()
 {
   down(q);
   down(s);
   ...
   up(q);
   up(s);
 }
```

8.3.4 Readers/Writers Problem

Neben den Producern/Consumern gibt es bei Betriebssystemen eine weitere klassische Situation: Eine Ressource wird von mehreren Prozessen parallel beschrieben bzw. gelesen. Lesende Prozesse sollen Reader, schreibende Prozesse Writer heißen. Das Problem nennt man „*Readers/Writers Problem*“.

Es können offensichtlich problemlos beliebig viele Prozesse gleichzeitig aus der Ressource lesen. Schwierigkeiten gibt es, sobald mindestens ein Writer beteiligt ist. Daher muss man fordern, dass ein Writer immer exklusiven Zugriff hat.

Da die eigentlichen Lese- und Schreibaktionen von der jeweiligen Ressource abhängen, können wir sie hier nicht aufführen. Stattdessen geben wir vier Funktionen an, die vor bzw. nach jedem Zugriff aufgerufen werden müssen. Zur geordneten Kooperation müssen sich die Prozesse natürlich an dieses Schema halten:

```
class ReadersWriters
 {
   private:
      binary_semaphore mutex,writing;
      int readers;
   public:
      shared_resource() : mutex(1), writing(1), readers(0) { }
      void start_writing() { down(writing); }
      void stop_writing() { up(writing); }
      void start_reading()
       {
         down(mutex);
         if (++readers==1) down(writing);
         up(mutex);
       }
      void stop_reading()
       {
         down(mutex);
         if (--readers==0) up(writing);
         up(mutex);
       }
 };
```

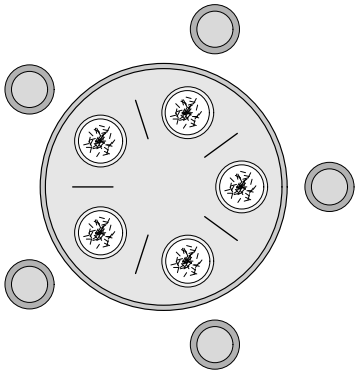
`mutex` baut um die Operationen mit dem Zähler `readers` einen kritischen Bereich auf. `writing` stellt die Exklusivität beim Schreibzugriff sicher. Der jeweils *erste* Reader zählt ihn herunter, der jeweils *letzte* Reader zählt ihn hinauf, ebenso natürlich *jeder* Writer.

Unsere Implementation legt nicht fest, was bei der Aufgabe des exklusiven Zugriff eines Writers passiert: Es kann sowohl ein wartender Writer direkt wieder Exklusivzugriff erhalten, wie auch die Reader gemeinsamen Zugriff erhalten können. Die Entscheidung hängt von der Implementation der Warteschlange der Semaphore ab.

Es besteht für alle Prozesse die Gefahr des Verhungerns, bei mehreren Readern aber besonders für den Writer. Solange noch irgendein Reader Zugriff hat, hat ein Writer keine Chance auf exklusiven Zugriff.

8.3.5 Das Philosophen-Problem

Beim „*Dining Philosophers Problem*“ handelt es sich nicht unbedingt um eine Situation, die im Zusammenhang mit Betriebssystemen häufig eintritt. Sie steht aber stellvertretend für eine ganze Klasse von Synchronisationsproblemen.



Fünf (manchmal chinesische) Philosophen sitzen um einen runden Tisch herum. Sie verbringen ihr ganzes Leben damit, abwechselnd zu denken und zu essen. Für zweiteres hat jeder einen Teller Reis vor sich, und jeweils zwischen zwei Tellern liegt ein Ess-Stäbchen. Zum Essen sind zwei dieser Stäbchen notwendig. (Die ursprüngliche Formulierung mit Spaghetti und zwei Gabeln war nicht ganz so realitätsnah.)

Wenn ein Philosoph hungrig wird, nimmt er nacheinander das Stäbchen zu seiner linken und zu seiner rechten auf, isst eine Weile und legt die Stäbchen nacheinander wieder ab.

Wenn eines der Stäbchen gerade nicht verfügbar sein sollte, wartet er, bis es abgelegt wird. Falls er bereits ein erstes Stäbchen aufgenommen haben sollte, legt er dieses aber nicht wieder ab!

Mit binären Semaphoren kann man natürlich leicht garantieren, dass niemals zwei Philosophen gleichzeitig dasselbe Stäbchen benutzen:

```
binary_semaphore chopstick[5]={ 1,1,1,1,1 };

void philosopher(int i)    // i=0...4
 {
   for (;;)
    {
      think();
      down(chopstick[i]);
      down(chopstick[(i+1)%5]);
      eat();
      up(chopstick[i]);
      up(chopstick[(i+1)%5]);
    }
 }
```

Beispiel: Leider erzeugt diese Lösung schnell einen *Deadlock*. Wenn alle fünf gleichzeitig beginnen wollen zu essen, heben alle ihr linkes Stäbchen auf und finden dann das rechte nicht mehr vor. Alle Philosophen hängen für immer beim zweiten `down` fest und verhungern.

Es gibt einige Möglichkeiten, das in diesem speziellen Fall zu verhindern:

- Der Ess-Vorgang wird mit einem zählenden Semaphor gesichert, sodass maximal vier Philosophen gleichzeitig mit dem Aufnehmen der Stäbchen anfangen können.
- Das Aufheben der Stäbchen wird in einen kritischen Bereich eingeschlossen. Darin gibt der Philosoph (zunächst) auf, wenn er nicht *beide* Stäbchen vorfindet.

- Geradzahlige Philosophen nehmen zuerst das linke Stäbchen auf, ungeradzahlige dagegen zuerst das rechte (funktioniert allgemein nur bei ungeradzahlig vielen Philosophen).

Wenn man lediglich darauf achtet, einen Deadlock zu verhindern, kann es dennoch sein, dass Philosophen verhungern (beispielsweise im letzten Fall).

8.3.6 UND-Synchronisation

Der bei der Synchronisation am häufigsten benutzte Operator ist das logische UND: Ein Prozess benötigt eine Ressource UND eine weitere, um arbeiten zu können. Um die Entscheidung treffen zu können, zu warten oder weiterzulaufen, müssen mehrere Bedingungen gleichzeitig erfüllt sein, etc.

Gerade bei solchen UND-Entscheidungen werden oft Deadlocks erzeugt, wie wir in 8.3.3 und 8.3.5 ja gesehen hatten.

In solch speziellen Situationen, wenn mehrere Ressourcen hintereinander angefordert werden, gibt es eine einfache Möglichkeit, Deadlocks zu vermeiden. Dazu kann man die Semaphor-Operationen `down` und `up` auf **Mengen** von Semaphoren erweitern. Es muss natürlich weiterhin sichergestellt sein, dass diese Operationen *atomar* ausgeführt werden!

Unsere Erweiterungen sollen `s_down` und `s_up` heißen ('`s`' für simultan).

```
binary_semaphore semaphore::s_mutex;

static void semaphore::s_down(semaphore s[], int count)
 {
   int i;
   for (;;)
    {
      s_mutex.down();
      for ( i=0 ; i<count ; ++i ) if (s[i].v==0) break;
      if (i==count) break;
      s[i].l.enqueue(thisprocess);
      s_mutex.up();
      thisprocess.block();
    }
   for ( int i=0 ; i<count ; ++i ) --s[i].v;
   s_mutex.up();
 }

static void semaphore::s_up(semaphore s[], int count)
 {
   s_mutex.down();
   for ( int i=0 ; i<count ; ++i )
    {
      while (!s[i].l.isempty()) s[i].l.dequeue().wakeup();
      ++s[i].v;
    }
   s_mutex.up();
 }
```

Um die Atomarizität der Operationen zu gewährleisten, schließen wir sie jeweils durch einen binären Semaphor `s_mutex` in einen kritischen Bereich ein.

Im Pseudo-Code wollen wir die simultanen Operationen statt mit der Array-Schreibweise auch wie folgt notieren können: `s_down(`$\texttt{sem}_1$`,`$\texttt{sem}_2$`,...,`$\texttt{sem}_n$`)`, etc.

Beim UNIX-Systemaufruf `semop` werden keine einzelne Semaphoren betroffen; es wird immer gleich mit Semaphor-Mengen gearbeitet.

Beispiel: Unser Philosophen-Problem, mit den erweiterten Operationen gelöst, sieht wie folgt aus:

```
binary_semaphore chopstick[5]={ 1,1,1,1,1 };
void philosopher(int i)     // i=0...4
 {
   for (;;)
    {
      think();
      s_down(chopstick[i], chopstick[(i+1)%5]);
      eat();
      s_up(chopstick[i], chopstick[(i+1)%5]);
    }
 }
```

Diese Lösung kann keinen Deadlock erzeugen. Verhungern ist allerdings immer noch möglich.

8.3.7 Zigarettendreher-Problem

Dieses Problem wurde 1971 von Patil formuliert. Drei Prozesse C_1, C_2 und C_3 sind Raucher, die ihre Zigaretten zunächst selbst drehen müssen. Sie benötigen dazu drei Ressourcen: Tabak, Papier und ein Streichholz. C_1 besitzt genügend Tabak, C_2 genügend Papier und C_3 genügend Streichhölzer. Drei andere Prozesse P_1, P_2 und P_3 bieten diese Ressourcen an:

- P_1 liefert Tabak und Streichhölzer,
- P_2 liefert Streichhölzer und Papier,
- P_3 liefert Papier und Tabak.

Es kann immer nur einer der drei Producer zur selben Zeit liefern. Die Producer sind erst wieder bereit, wenn das zuletzt gelieferte Produkt vollständig verbraucht worden ist.

Eine Lösung ist mit erweiterten Semaphor-Operationen vergleichsweise einfach:

```
binary_semaphore tobacco=0, wrapper=0, match=0, mutex=1;

void P_1()
 {
   for (;;)
    {
      down(mutex);
      s_up(tobacco,match);
    }
 }
```

```
void P_2()
 {
   for (;;)
    {
      down(mutex);
      s_up(match,wrapper);
    }
 }
```

```
void P_3()
 {
   for (;;)
    {
      down(mutex);
      s_up(wrapper,tobacco);
    }
 }
```

```
void C_1()
 {
   for (;;)
    {
      s_down(match,wrapper);
      smoke();
      up(mutex);
    }
 }
```

```
void C_2()
 {
   for (;;)
    {
      s_down(wrapper,tobacco);
      smoke();
      up(mutex);
    }
 }
```

```
void C_3()
 {
   for (;;)
    {
      s_down(tobacco,match);
      smoke();
      up(mutex);
    }
 }
```

8.4 Monitore

Ein „**Monitor**“ ist ein eleganter Mechanismus zur Prozess-Synchronisation (der Begriff wurde von C.A.R. Hoare geprägt).

Semaphore sind noch vergleichsweise Low-Level, reichen aber sowohl für wechselseitigen Ausschluss wie auch zur Synchronisation völlig aus. Besonders durch diese doppelte Verwendung ist Code, der mehrere Semaphore verwendet, aber manchmal etwas schwer zu lesen und zu warten.

- Ein Monitor versucht hier Abhilfe zu schaffen, indem er in seiner Struktur dem eigentlichen Programm angenähert ist. Ein Monitor koppelt kritische Bereiche verschiedener Prozesse. Der Programmierer ist dadurch gezwungen, die kritischen Bereiche aus dem Code zu isolieren und in den Monitor einzubauen, während er Semaphore über sein ganzes Programm verstreuen dürfte.

- Genauer besteht ein Monitor aus einem Satz von Variablen und Methoden (Funktionen, Prozeduren), von denen nur genau spezifizierte von außen zugänglich sind.

  Wir haben also in etwa ein *Objekt* einer *Klasse* im Sinn objektorientierter Programmierung vor uns. Entsprechend werden wir hier unseren Pseudo-Code hier an die Klassenschreibweise von `C++` anlehnen.

- Ein Monitor ist ein prozessübergreifendes Objekt. Was ihn aber auszeichnet, ist, dass immer nur maximal ein Prozess in ihm aktiv sein, also eine in ihm enthaltende Methode ausführen darf!

  So kann auf simple Weise der wechselseitige Ausschluss realisiert werden, ohne dass sich der Programmierer selbst um die Details zu kümmern braucht (wobei er so schwerwiegende Fehler machen könnte wie das Vertauschen von Semaphoren).

- Die Variablen im Monitor sind gemeinsame Ressourcen. Da nur in den Monitor-Methoden auf sie zugegriffen werden kann, sind sie automatisch vor gemischtem Zugriff geschützt.

- Es wird aber weiterhin eine Möglichkeit benötigt, Prozesse aufgrund von Bedingungen (Schlange voll/leer) zu blockieren. Dazu stellt der Monitor sogenannte *Bedingungsvariable* („*condition variables*“) zur Verfügung, auf die die Monitor-Operationen `wait` und `notify` anwendbar sind:
  - Ein Prozess ruft `wait` auf, um sich selbst zu blockieren, bis eine günstigere Situation eintritt.

– Ein Prozess ruft `notify` auf, um wartenden Prozessen mitzuteilen, dass sich die Situation eingestellt hat.

Da nach einem `notify` ein wartender Prozess wieder aktiv werden soll, aber nur ein Prozess innerhalb des Monitors erlaubt ist, darf `notify` nur der letzte Befehl in einer Monitor-Methode sein (evtl. mit der Ausnahme eines **`return`** zur Wertrückgabe).

- In einigen Monitor-Versionen gibt es eine weitere Funktion namens `notifyAll` oder `broadcast`, die *alle* Prozesse aufweckt, die auf die angegebene Bedingung warten.

Es ist schon leicht zu erkennen, dass ein Monitor in etwa einem binären Semaphor (zur Eintrittskontrolle) zusammen mit einer Menge von zählenden Semaphoren (einer pro Bedingungsvariable) entspricht.

Beispiel: Als erstes wollen wir das Readers/Writers-Problem mit einem Monitor lösen:

```
monitor ReadersWriters
 {
   int readers;                              // zählt die Reader
   bool writing;                             // Writer hat exklusiven Zugriff
   condition readOK, writeOK;                // getrennte Bedingungs-Variablen

   void start_reading()
    {
      if (writing) readOK.wait();            // Writer schreibt, Reader blockiert
      ++readers;                             // ok, darf schreiben
      if (readOK.queued()) readOK.notify(); // einen anderen Leser wecken
    }

   void stop_reading()
    {
      if (--readers==0) writeOK.notify();    // kein Leser mehr ⇒ schreiben okay
    }

   void start_writing()
    {
      if ( readers!=0 || writing )           // kein exklusiver Zugriff möglich
         writeOK.wait();
      writing=true;                          // aber jetzt
    }

   void stop_writing()
    {
      writing=false;
      if (readOK.queued()!=0) readOK.notify();  // Reader bevorzugen
      else writeOK.notify();
    }
 };
```

Die zusätzliche Lesefunktion **`bool queued()`** der Bedingungsvariablen schaut nach, ob es Prozesse in der Warteschlange der Variable gibt. Wir erreichen hier, dass Reader gegenüber Writern

leicht bevorzugt werden. Gibt es eine solche Funktion nicht, muss man die Variablen `readOK` und `writeOK` verschmelzen. Es bleibt am Ende von `stop_writing` dann dem Scheduler-Zufall überlassen, ob als nächstes ein Reader oder Writer an die Reihe kommt.

In `start_reading` wird `queued` dazu benutzt, dass sich die Reader nacheinander aufwecken, sobald einer aufgeweckt wurde. Ohne `queued` kann man in `stop_writing` die Funktion `notifyAll` mit demselben Endeffekt benutzen.

Beispiel: Auch unser Producer–Consumer-Problem lässt sich einfach mit einem Monitor lösen, der die eigentlichen Warteschlangen-Operationen als Methoden enthält:

```
monitor PC
 {
   condition full, empty;
   int count;

   void prod_func(item i)              // wird nur vom Producer aufgerufen
    {
      if (count==N) full.wait();       // Schlange voll, Producer blockiert
      enter_item(i);
      if (++count==1) empty.notify();  // Schlange nicht mehr leer ⇒ Consumer
    }

   item cons_func()                    // wird nur vom Consumer aufgerufen
    {
      if (count==0) empty.wait();      // Schlange leer, Consumer blockiert
      item i=remove_item();
      if (--count==N-1) full.notify(); // Schlange nicht mehr voll ⇒ Producer
      return i;
    }
 };

void producer()                        void consumer()
 {                                      {
   for (;;)                               for (;;)
    {                                      {
      PC.prod_func(produce_item());          consume_item(PC.cons_func());
    }                                      }
 }                                      }
```

Semaphore sind leicht in jeder Programmiersprache (durch Aufrufe von Bibliotheksfunktionen) zu verwenden. Auch für Monitore wäre das zwar denkbar, aber umständlich. Man müsste jeweils einen Monitor neu anlegen, Funktionen darin registrieren, das Testen der `notify`-Bedingung wird schwierig, etc.

Es ist sinnvoller, wenn das Monitor-Konzept direkt in das Design der Sprache eingeht, wie es z.B. bei den eher exotischen Sprachen *Concurrent Euclid* und `Mesa` der Fall ist.

Eine sinnvolle Teil-Realisierung gibt es aber auch in `JAVA` bei Threads mit dem Schlüsselwort **`synchronized`**. Methoden in einem Objekt, die hiermit deklariert werden, können nur von

einem Thread zur selben Zeit durchlaufen werden. JAVA verwaltet den entsprechenden Lock-Mechanismus selbst.

Ein JAVA-Objekt wird also automatisch zum Monitor, wenn alle öffentlichen Methoden als **synchronized** deklariert werden (und alle Daten **private**).

Beispiel: Wir implementieren das **Producer–Consumer-Problem** vollständig in JAVA. Wir erzeugen und verbrauchen Strings. Als Warteschlange verwenden wir der Einfachheit halber einen String-Vektor, den wir selbst aber nur mit einer bestimmten Maximalzahl an Elementen füllen. Der Producer erzeugt jede Sekunde einen String (mit der aktuellen Systemzeit zum Inhalt).

An ihn hängen wir *zwei* Consumer, die sich um die Strings streiten. In der unten angegebenen Version holen sie sich jeder alle 2 Sekunden einen String ab, sind also gleichberechtigt und mit dem Producer gleichauf. Durch Modifikation der Zeiten kann man andere Effekte erzielen.

Damit das Schlüsselwort **synchronized** funktionieren kann, müssen die „Monitor-Funktionen" im selben Objekt liegen – bei uns sind sie in den Producer gelegt.

Die Consumer numerieren sich intern selbst (zur Identifikation in ihren Test-Ausgaben). Die Zählvariable counter wird dazu ebenfalls geschützt durch einen **synchronized**-Mechanismus.

Die JAVA-Syntax und -Semantik kann hier natürlich nicht ausführlich erklärt werden. Mit Hilfe der vorangegangenen Überlegungen sollte sich der folgende Quelltext aber einigermaßen verstehen lassen (Übersetzen mit `javac prodcons.java`, Starten mit `java prodcons`):

```
import java.util.*;                              // für Vector und Date

public class prodcons                            // "Controlling Class", Startklasse
 {
   public static void main(String[] args)       // "Hauptprogramm"
    {
      PC_Monitor mon=new PC_Monitor();           // einen Monitor anlegen
      new Producer(mon);                         // Producer anlegen
      new Consumer(mon);                         // Consumer anlegen
      new Consumer(mon);                         // noch einen
    }
 }

class PC_Monitor
 {
   static final int MAXQUEUE=5;                  // Maximalfüllung der Warteschlange
   private Vector queue=new Vector();            // die Warteschlange (java.util.Vector)

   public synchronized void prod_func(String s) throws InterruptedException
    {
      while (queue.size()==MAXQUEUE) wait();     // Blockieren, falls Schlange voll
      queue.addElement(s);                       // Element in die Schlange einfügen
      notify();                                  // Signalisieren: Schlange nicht mehr leer
    }
```

```
   public synchronized String cons_func() throws InterruptedException
    {
      notify();                                  // Signalisieren: Schlange nicht mehr voll
      while (queue.size()==0) wait();            // Blockieren, falls Schlange leer
      String s=(String)queue.firstElement(); // Element aus der Schlange herausholen
      queue.removeElement(s);
      return s;
    }
 }

class Producer extends Thread                    // Producer-Klasse, arbeitet mit Threads
 {
   private PC_Monitor PC;                        // zuständiger Monitor

   Producer(PC_Monitor m)                        // Konstruktor von Producer
    {
      PC=m;                                      // übergebenen Monitor merken
      start();                                   // Thread starten (ruft intern run auf)
    }

   String produce_item()                         // Item produzieren:
    {
      return new Date().toString();              // Datum holen und in String umwandeln
    }

   public void run()                             // wird von start automatisch aufgerufen
    {
      try
       { for (;;)                                // für immer:
          { PC.prod_func(produce_item());        // Item produzieren, Ablageversuch
            sleep(1000);                         // 1 Sekunde schlafen
          }
       }
      catch (InterruptedException e) { }
    }
 }

class Consumer extends Thread                    // Consumer-Klasse, arbeitet mit Threads
 {
   private PC_Monitor PC;                        // der zuständige Monitor
   private static int counter;                   // Zähler für die verschiedenen Consumer
   private int number;                           // Nummer dieses Consumers

   public synchronized void getNumber()          // eigene Nummer ermitteln (sync)
    { number=++counter; }

   Consumer(PC_Monitor m)                        // Konstruktor von Consumer
    {
      getNumber();                               // eigene Nummer ermitteln
```

```
    PC=m;                                   // übergebenen Monitor merken
    start();                                // Thread starten (ruft intern run auf)
  }

  void consume_item(String s)               // Item konsumieren
  {
    System.out.println(number+": "+s);      // durch Ausgeben auf den Bildschirm
  }

  public void run()                         // wird von start automatisch aufgerufen
  {
    try
     { for (;;)                             // für immer
        { consume_item(PC.cons_func());     // Item abholen, konsumieren
          sleep(2000);                      // 2 Sekunden schlafen
        }
     }
    catch (InterruptedException e) { }
  }
}
```

JAVA implementiert keinen vollständigen Monitor-Mechanismus. Es gibt hier keine Bedingungsvariablen (oder nur genau eine pro Monitor). Das Warten (`wait`) und Aufwecken (`notify`) ist hier sozusagen „Bedingungs-global“. Es liegt in der Verantwortung des aufgeweckten Threads, zu prüfen, ob die erwartete Situation eingetreten ist, oder ob er weiterschlafen muss.

Beispiel: Außerdem wollen wir das **Philosophen-Problem** mit einem JAVA-Monitor lösen.

Was die technische Seite von JAVA angeht, so ist zu beachten, dass wir für jeden Philosophen einen Thread starten, der alle paar Sekunden hungrig wird und dann (ggf. nach Wartezeit) einige Sekunden lang isst, gesteuert durch einen Zufallsgenerator.

Die Stäbchen und die Zugriffe auf sie liegen in der Monitor-Klasse. Aber auch die statische Methode `show` für die Bildschirm-Anzeige muss synchronisiert werden (mit sich selbst!), da sie sonst von verschiedenen Philosophen fast gleichzeitig aufgerufen werden und die Ausgabe durcheinandergeraten könnte. Statische Methoden und die an ein Objekt gebundenen liegen in JAVA in getrennten Monitoren.

Beachte, dass ein satter Philosoph so oft `notify` aufruft, wie er hungrige Nachbarn hat.

```
import java.util.Random;

public class dining_philosophers                    // Startklasse
 {
   public static void main(String[] args)           // "Hauptprogramm"
    {
      Philosopher[] p=new Philosopher[5];           // Array aus Referenzen
      Ph_Monitor mon=new Ph_Monitor(p);
      for (int i=0;i<5;++i) p[i]=new Philosopher(i,mon,p);
      Philosopher.show();                           // erste Anzeige
    }
 }
```

```
class Ph_Monitor
 {
   final boolean[] chopstick;
   final Philosopher[] p;

   Ph_Monitor(Philosopher[] parr)
    {
      p=parr;
      chopstick=new boolean[p.length];
    }

   synchronized void hungry(int i) throws InterruptedException
    {
      p[i].state=Philosopher.hungry;
      Philosopher.show();
      while (chopstick[i]||chopstick[(i+1)%p.length]) wait();
      chopstick[i]=chopstick[(i+1)%p.length]=true;
      p[i].state=Philosopher.eating;
      Philosopher.show();
    }

   synchronized void full(int i)
    {
      p[i].state=Philosopher.thinking;
      Philosopher.show();
      chopstick[i]=chopstick[(i+1)%p.length]=false;
      int count=0;
      if (p[(i+p.length-1)%p.length].state==Philosopher.hungry) ++count;
      if (p[(i+1)%p.length].state==Philosopher.hungry) ++count;
      while (count-->0) notify();
    }
 }

class Philosopher extends Thread
 {
   static final int thinking=0, hungry=1, eating=2;
   static final String[] s_msg={ "thinking", " hungry ", " eating " };
   static Random rangen=new Random();
   static Philosopher p[];                          // Array aller Philosophen
   int state=thinking;                              // Zustand
   Ph_Monitor mon;                                  // Monitor
   int num;                                         // Numerierung

   Philosopher(int i, Ph_Monitor m, Philosopher[] parr)
    { num=i; mon=m; p=parr; start(); }

   public void run()                                // Einsprung für den Thread
    {
      try                                           // Interrupt abfangen
       {
```

```
        for (;;)
         {
           sleep(100+rangen.nextInt()&1023);        // denken
           mon.hungry(num);                         // hungrig werden
           sleep(100+rangen.nextInt()&1023);        // essen
           mon.full(num);                           // satt sein
         }
       }
      catch (InterruptedException e) { }
    }

    static synchronized void show()                 // Anzeige aller fünf P.
     {
       int n=p.length;
       for (int i=0;i<n;++i)
        {
          System.out.print(
             (p[i].state==eating || p[(i+n-1)%n].state==eating)?"X ":"| ");
          System.out.print(s_msg[p[i].state]+" ");
        }
       System.out.println(
          (p[0].state==eating || p[n-1].state==eating)?"X ":"| ");
     }
  }
```

Eine mögliche Ausgabe ist folgende:

```
| thinking | thinking | thinking | thinking | thinking |
| thinking | thinking |  hungry  | thinking | thinking |
| thinking | thinking X  eating  X thinking | thinking |
| thinking |  hungry  X  eating  X thinking | thinking |
|  hungry  |  hungry  X  eating  X thinking | thinking |
X  eating  X  hungry  X  eating  X thinking | thinking X
X  eating  X  hungry  X  eating  X  hungry  | thinking X
X  eating  X  hungry  | thinking |  hungry  | thinking X
X  eating  X  hungry  | thinking X  eating  X thinking X
...
```

8.5 Nachrichten

Die bisher kennengelernten Mechanismen dienten zur Synchronisation von Prozessen, insbesondere der

- Vermeidung von Inkonsistenzen durch gemischten Zugriff aus gemeinsame Daten,
- Synchronisation, der Regelung von Zugriffszeitpunkt und -Reihenfolge.

Sie boten noch keine Möglichkeit für den eigentlichen Datenaustausch. Eine allgemeine Methode dazu ist das Austauschen von Nachrichten, das „**Message Passing**“.

Es ist dadurch von besonderem Vorteil, dass es auch auf Systemen mit *mehreren Prozessoren* oder in Netzen mit *mehreren Rechnern* durchführbar ist. (Dort ist es außerdem meist die einzige

Möglichkeit, Prozesse zu synchronisieren.) Es sind keine weiteren gemeinsamen Ressourcen wie ein gemeinsam zugreifbares Dateisystem oder gemeinsamer Speicher notwendig.

Eine Interprozess-Kommunikation zwischen unabhängigen Prozessen arbeitet mit Message Passing, wenn sie folgende beiden Operationen zur Verfügung stellt (zuerst formuliert von Brinch-Hansen 1973):

`send(`*destination*`,`*message*`)` – Verschicken einer „Nachricht“
`receive(`*source*`,`*message*`)` – Empfangen einer „Nachricht“

source und *destination* stehen dabei für Systemstrukturen, über die die Prozesse einander finden. Hier wäre direkt die Identifikation des jeweils anderen Prozesses (etwa über seine PID) denkbar, oder es könnte eine Art „Mailbox“ angegeben werden, die von mehreren Prozessen verwendet wird.

Das Senden kopiert den Inhalt der Nachricht zunächst aus dem Speicherbereich des sendenden Prozesses in einen Systempuffer, sodass die entsprechende Variable nach dem `send`-Aufruf verändert werden kann. Der Datenaustausch erfolgt irgendwann später auf Systemebene. Entsprechend kopiert der `receive`-Aufruf die Nachricht aus einem Puffer des Empfänger-Rechners in den Speicherbereich des empfangenden Prozesses.

Es sind beliebig viele Implementationen denkbar. Die IPC könnte unidirektional oder bidirektional sein. Außerdem gibt es folgende Unterscheidung:

indirekt – die IPC findet über Systemstrukturen wie Warteschlangen o.ä. statt. Bei verteilten Systemen ist fast nur MP sinnvoll denkbar.

direkt – zwischen zwei Prozessen, das System ermöglicht den Austausch z.B. über bei gemeinsamem Speicher, Busse oder Ports (schnell, aber natürlich nur in entsprechenden Systemen realisierbar, nicht in Netzen).

Üblicherweise puffert das System die empfangenen Nachrichten, bis sie abgeholt werden. Im Netz kann es aber auch Puffer auf der Absenderseite geben (Leitungseigenschaften, Sammelnachrichten, etc.).

Wenn nicht gepuffert wird, müssen `send`- und `receive`-Aufrufe *direkt* miteinander kommunizieren. Dann nennt man das Zustandekommen der Kommunikation ein **Rendezvous**. Wenn zuerst das `send` ausgeführt wird, blockiert der Prozess, bis der Empfänger-Prozess bei einem `receive` ankommt – analog umgekehrt. So sollten allerdings nur zwei Prozesse (und nicht mehr) miteinander reden, da sich sonst eventuell die Gesprächspartner nicht in der richtigen Reihenfolge finden und sich alle gegenseitig blockieren.

Probleme, die bei der Implementation von Message Passing gelöst werden müssen, sind folgende:

- Semaphor- oder Monitor-Operationen sind sehr schnell, der eigentliche Datenaustausch kann dagegen recht langsam werden. Wenn die IPC auf einem Rechner (und insbesondere bei nur einem Prozessor) stattfindet, kann (und sollte) sie besonders effektiv implementiert werden, z.B. durch gemeinsamen Speicher.
- Die Nachrichten müssen eindeutig identifizierbar sein. Sie werden daher mit einem Absender- und Empfänger-Hinweis versehen, die eine Prozess- und (im Netz) Rechner-Identifikation in irgendeiner Form enthalten.

- Im Netz können Nachrichten bei der Übertragung schon einmal verlorengehen. Oft arbeitet man deshalb mit einer Empfangsbestätigung („*Acknowledgement*“), deren Ausbleiben nach einer gewissen Zeit zu einem Neuversenden der Nachricht führt.

  Wenn das Acknowledgement verlorengeht, wird die ursprüngliche Nachricht überflüssigerweise erneut gesandt. Daher werden die Nachrichten durchnummeriert, und eine zweimal empfangene Nachricht wird beim Empfänger ignoriert.

- Um zu verhindern, dass sich gefälschte Nachrichten in den Strom einschleichen, kann man eine Form von Verschlüsselung verwenden, auf die sie die kommunizierenden Prozesse vorher einigen müssen.

Beispiel: Wir lösen nun unser Producer–Consumer-Problem mit Hilfe von Message Passing. Die beiden Prozesse sollen sich dabei direkt mit `send` und `receive` ansprechen können. Das Beispiel deutet außerdem an, wie man z.B. in einem Netz Prozesse mit Hilfe von Nachrichten synchronisieren kann.

Der Producer könnte nun theoretisch mit hoher Geschwindigkeit beliebig viele Nachrichten produzieren und losschicken. Das System wäre gezwungen, sie zwischenzupuffern, könnte aber irgendwann überlastet sein:

```
void producer()
 {
   for (;;)
    {
      message m;
      build_message(m,produce_item());
      send(consumer,m);
    }
 }
```

```
void consumer()
 {
   for (;;)
    {
      message m;
      receive(producer,m);
      consume_item(extract_item(m));
    }
 }
```

Die beiden Funktionen `build_message` und `extract_item` dienen zum Ein-/Ausbau der eigentlichen Daten in die/aus der Message.

Wir können die Geschwindigkeiten von Producer und Consumer hier dadurch angleichen, dass wir den Consumer Dummy-Nachrichten an den Producer verschicken lassen, um anzudeuten, dass er empfangsbereit ist. Der Producer quittiert den Empfang jeder Dummy-Nachricht mit dem Versenden einer echten Nachricht.

```
void producer()
 {
   for (;;)
    {
      message m;
      item i=produce_item();
      receive(consumer,m);
      build_message(m,i);
      send(consumer,m);
    }
 }
```

```
void consumer()
 {
   message m;
   for (int i=0;i<N;++i)
      send(producer,m);
   for (;;)
    {
      receive(producer,m);
      item i=extract_item(m);
      send(producer,m);
      consume_item(i);
    }
 }
```

Zu Beginn schickt der Consumer `N` Dummy-Nachrichten. Dadurch ist die Gesamtzahl der Nachrichten im System auf `N` beschränkt. Die beiden Prozesse blockieren automatisch in ihren `receive`-Aufrufen.

8.6 Äquivalenz der IPC-Mechanismen

Es sind noch wesentlich mehr Synchronisations-Mechanismen als die besprochenen bekannt. Es ist aber für ein Betriebssystem nicht notwendig, sie alle zu implementieren. Man kommt mit einem aus: *entweder* Semaphore *oder* Monitore *oder* Nachrichten.

Die Semaphore sind die einfachsten der Mechanismen und werden von fast jedem System mehr oder weniger direkt zur Verfügung gestellt. Wir schauen uns deshalb an, wie man die anderen Arten mit Semaphoren realisieren könnte.

8.6.1 Monitore, gebaut mit Semaphoren

Folgende Pseudo-`C++`-Klasse realisiert einen Monitor mit Hilfe von Semaphoren. Zu Beginn jeder zu synchronisierenden Funktion muss explizit `enter()`, an ihrem Ende `leave()` aufgerufen werden. Die Bedingungsvariablen sind ebenfalls Semaphore, die aber außerhalb des Monitors definiert und den Funktionen `wait` und `notify` übergeben werden müssen:

```
class Monitor
 {
   private:
      semaphore mutex;
   public:
      Monitor() : mutex(1) { }
      void enter() { down(mutex); }
      void leave() { up(mutex); }
      void notify(semaphore &s) { up(s); }
      void wait(semaphore &s) { up(mutex); down(s); }
 };
```

Der Semaphor `mutex` kontrolliert den Eintritt in den kritischen Bereich. Für jede Bedingungsvariable muss aber ein zusätzlicher Semaphor verwendet werden!

- Die `wait`-Operation gibt zunächst den kritischen Bereich frei und wartet dann ggf. mit `down` auf ein Signal.
- Die Monitor-Operation `notify` darf ja immer nur am Ende eines kritischen Bereichs aufgerufen werden. Deshalb ist die hier angegebene Implementation möglich: Es wird `up` für die entsprechende Variable aufgerufen, ohne `mutex` heraufzuzählen. Der durch das `up` aufgeweckte Prozess *übernimmt* auf diese Weise die Blockierung des kritischen Bereichs.

8.6.2 Nachrichten, gebaut mit Semaphoren

Der Einfachheit der Betrachtung halber soll das Austauschen der Nachrichten in einem Bereich gemeinsamen Speichers stattfinden. Die entstehenden Datenstrukturen werden leider etwas länglich und können hier nicht vollständig aufgeführt werden.

- In dem gemeinsamen Bereich können beliebig viele Message-Boxes angelegt werden, z.B. eine für jeden Prozess für die für ihn bestimmten Nachrichten.
- Jede Box führt mit Hilfe von Listen Buch darüber, welche Prozesse *nicht* an sie senden konnten (wegen Überfüllung, `send`-Liste), und welche Prozesse *nichts* aus ihr abholen konnten (weil sie leer war, `receive`-Liste).
- Der gesamte Bereich wird mit Hilfe eines binären Semaphors `mutex` geschützt, d.h. zu Beginn der Operationen `send` und `receive` wird ein `down(mutex)`, am Ende ein `up(mutex)` ausgeführt.

Das Blockieren bei einem `send` oder `receive` geschieht mit einem persönlichen Semaphor für jeden der kommunizierenden Prozesse.

- Wenn ein Prozess ein `receive` für eine Box aufruft, die nicht ganz leer ist, kann er direkt ein Element abholen. Ansonsten trägt er sich in die `receive`-Liste der Box ein, gibt den kritischen Bereich frei, ruft `down` für seinen persönlichen Semaphor auf und blockiert sich damit. Wenn er aufgeweckt wird, reserviert er sich sofort wieder den kritischen Bereich, entfernt sich aus der `receive`-Liste und holt die neue Nachricht ab.
- Analog trägt sich ein Prozess in die `send`-Liste einer Box ein, wenn er ein `send` nicht direkt ausführen kann, weil die Box voll ist. Er gibt dann ebenso den kritischen Bereich frei und ruft `down` für seinen Semaphor auf.
- Nach einem erfolgreichen `send` muss der sendende Prozess überprüfen, ob die `receive`-Liste der Box nicht leer ist, und ggf. einen dort wartenden Prozess aufwecken. Er verlässt dann den kritischen Bereich, der vom aufgeweckten Prozess direkt wieder reserviert wird.

9 Prozesskommunikation in UNIX

9.1 FIFOs

Wir hatten bereits in Abschnitt 6.11 die einfachste Art von Kommunikation kennengelernt: Datenwarteschlangen im Form der Pipelines. Ihr wesentlicher Nachteil ist, dass die kommunizierenden Prozesse einen gemeinsamen Vorfahren besitzen müssen, der die Pipeline anlegt.

Es gibt eine andere Art von Warteschlangen, deren Benutzer nicht direkt miteinander verwandt sein müssen. Sie werden über normale Dateisystem-Namen identifiziert und heißen daher auch **Named Pipes**, meistens aber einfach **FIFO**s. Sie können wie Dateien gelesen, beschrieben, gelinkt werden, etc.

9.1.1 Die Bibliotheksfunktion `mkfifo`

Mit folgender Bibliotheksfunktion (deklariert in `sys/stat.h`) legt man eine FIFO an:

UNIX
`int mkfifo(const char *path, mode_t mode);` legt eine FIFO mit dem Pfad `path` im Dateisystem mit den Zugriffsrechten `mode&~umask` an

Die FIFO existiert danach als spezielle Datei. Der erste Buchstabe in einer Directory-Auflistung ist ein 'p' (für Pipe):

```
prw-------   1 root     root            0 Nov 29 01:47 initctl
```

`/dev/initctl` ist übrigens die FIFO, über die das Kommando `init` dem `init`-Prozess den Befehl schickt, in einen anderen Run Level zu schalten.

Eine FIFO kann nach dem Anlegen wie eine normale Datei behandelt werden. Jeder Prozess, der auf diese spezielle Datei zugreifen darf, kann die FIFO benutzen. Er öffnet sie dazu mit `open` – zum Lesen mit dem Flag `O_RDONLY`, zum Schreiben mit `O_WRONLY`. Der Datenzugriff erfolgt dann mit `read` bzw. `write`.

Prozesse müssen sich natürlich auf den Namen der FIFO einigen. Server-Prozesse benutzen immer Standard-Pfade, damit ihre Clients sie auch finden.

Es ist folgendes zu beachten:

- Es können *mehrere* Prozesse in die FIFO *schreiben* und *mehrere* Prozesse aus der FIFO *lesen*. Einmal gelesene Daten sind verloren.
- Die Daten mehrerer schreibender Prozesse werden *gemischt*. Es ist aber sichergestellt, dass die Daten aus einem einzigen `write`-Aufruf ohne Unterbrechung geschrieben werden – bis zum einem Maximum von `POSIX_PIPE_BUF` Bytes (z.B. 512).
- Wenn man eine FIFO zum Schreiben öffnet, ohne dass ein lesender Prozess existiert, blockiert `open`.
- Wenn man eine FIFO zum Lesen öffnet, ohne dass ein schreibender Prozess existiert, blockiert `open`.

- Wenn man im `open`-Aufruf das Flag `O_NONBLOCK` angibt, wird in den beiden obigen Fällen nicht blockiert. Beim Öffnen zum Schreiben erhält man einen Fehler.
- Wenn man in eine leserlose FIFO schreibt, wird das Signal `SIGPIPE` generiert.
- Wenn der letzte schreibende Prozess die FIFO schließt, erhalten die lesenden Prozesse ein EOF.

9.1.2 Das Kommando `mkfifo`

Am einfachsten ausprobieren kann man FIFOs mit dem Kommando `mkfifo`, das eine Schnittstelle zur gleichnamigen Bibliotheksfunktion ist:

`mkfifo [-m` *mode*`]` *pathname(s)*

Auf einigen Systemen gibt es das Kommando nicht – dort kann man sich mit `mknod` behelfen, das FIFOs oder Special Files anlegt (dazu später):

`mknod [-m` *mode*`]` *pathname* `p`

Von einer Shell aus gestartete Prozesse sind natürlich alle Nachfahren der Shell, sodass man mit Pipes auskäme. Der Vorteil von FIFOs ist, dass man

- in *weiter auseinanderliegenden* Kommandos in Shell-Skripten auf die FIFO zugreifen kann,
- mehrere FIFOs von einem Prozess ausgehend anlegen kann. d.h. die Ausgabe kann mehrfach verwendet werden Ausgabe und Fehlerkanal können in getrennte FIFOs wandern.

Beispiele:

- Wir legen in einem Fenster eine FIFO an und versuchen, sie zu füllen:

  ```
  makefifo /tmp/testfifo
  cp test.txt /tmp/testfifo
  ```

  Der `cp`-Aufruf blockiert, da die FIFO noch keinen Leser hat. In einem Fenster tippen wir nun

  ```
  cat /tmp/testfifo
  ```

  Wir erhalten dort den Inhalt von `test.txt` und kehren sofort zur Shell zurück. Gleichzeitig ist auch der `cp`-Aufruf im anderen Fenster beendet.

- So kann man mit zwei FIFOs arbeiten:

  ```
  mkfifo /tmp/outfifo /tmp/errfifo
  testcmd > /tmp/outfifo 2> /tmp/errfifo
  ```

  Das Kommando blockiert so lange, bis beide FIFOs einen Leser gefunden haben.

- Hier liefert ein schreibender Prozess `writer` Daten an drei lesende Prozesse `reader1`, `reader2`, `reader3`:

```
mkfifo /tmp/testfifo /tmp/testfifo2
reader2 < /tmp/testfifo &
reader3 < /tmp/testfifo2 &
writer | tee /tmp/testfifo | tee /tmp/testfifo2 | reader1
rm /tmp/testfifo /tmp/testfifo2
```

9.1.3 Server und Clients

Wie erwähnt, lösen FIFOs das Kommunikationsproblem zwischen Server- und Client-Prozessen, die nicht direkt miteinander verwandt sind. Wir implementieren zwei einfache solche Beispiele.

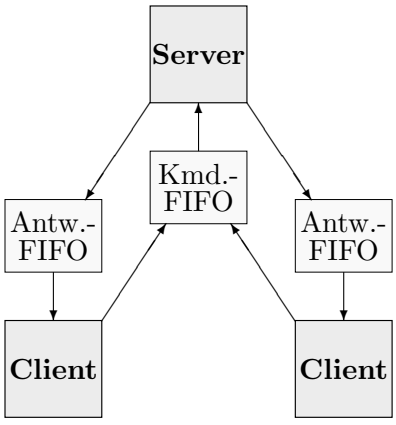
Beispiel: Zunächst schreiben wir nur einen C-Server, der darauf wartet, dass er Kommandos von einem Client erhält. Seine Reaktion besteht hier nur aus der Ausgabe der Kommandos auf seine Standard-Ausgabe:

```
int main()
 {
   int fd,err;
   char c;
   static char fifoname[]="/tmp/echoserv";

   err=mkfifo(fifoname,0x666);
   fd=open(fifoname,O_RDONLY);
   if ( err || fd<0 ) { perror(fifoname); exit(0); }

   while (read(fd,&c,1)==1) putchar(c);

   close(fd);
   unlink(fifoname);
 }
```

Mehrere Clients können gleichzeitig an den Server senden. Dazu setzen wir einfach in mehreren Fenstern das Kommando

```
cat > /tmp/echoserv
```

ab. Die danach eingegebenen Zeichen werden (durch die Terminal-Einstellung zeilenweise) an den Server geschickt, der sie (ggf. gemischt) ausgibt. Das `cat` können wir mit einem `EOF` (`CTRL-D`) abbrechen, wodurch die FIFO von diesem Prozess aus geschlossen wird. Wenn der letzte Client sich verabschiedet, liefert das `read` im Server eine Null, und der Server bricht ab und löscht die FIFO.



Wenn der Server den Clients Daten liefert, kann dieser Verkehr natürlich nicht über dieselbe FIFO abgewickelt werden wie die Kommandos. In diesem Fall legt jeder Client eine weitere FIFO an, in die der Server die Antwort schreibt.

Üblicherweise enthält der Name dieser FIFO die Prozessnummer des Clients, damit keine Überschneidungen entstehen können. Diese PID muss natürlich auch im Kommando mitgeschickt

werden. Der Client muss nach Erhalt der Antwort die FIFO selbst löschen.

Beispiel: Unser Server ermittelt den kleinsten Teiler (>1) einer natürlichen Zahl, die ihm von einem Client geschickt wird. Zur Identifikation von Primzahlen liefert er 0 (eine 1 bei 0 und 1).

Die Kommandos, die ihm geschickt werden, haben das Format „*pid*:*num*“, wobei *pid* die Prozess-Nummer des Clients und *num* die zu untersuchende Zahl ist.

Den Namen der Kommando-FIFO und den Namensanfang der Antwort-FIFOs definieren wir in einer Header-Datei `fifoname.h`, die Client und Server einbinden:

```
#define REQFIFO "/tmp/divireq"
#define ANSFIFO "/tmp/divians"
```

Wir zeigen zunächst das Client-Programm. Ihm können in der Aufrufzeile beliebig viele Zahlen übergeben werden, die es an den Server schickt. Es gibt eine Interpretation der erhaltenen Antwort aus. Wir wollen hier keine Binärdaten, sondern ASCII-Daten schicken. Daher werden zur einfacheren Formatierung mit `fdopen` C-Ströme um die File-Deskriptoren gelegt.

```
#include "fifoname.h"

int main(int argc, char *argv[])
 {
   int fd_cmd,i;
   FILE *cmdfifo;

   if ((fd_cmd=open(REQFIFO,O_WRONLY))<0)
    { fputs("server not running\n",stderr); exit(0); }
   cmdfifo=fdopen(fd_cmd,"w");

   for ( i=1 ; i<argc ; ++i )
    {
      char ansname[80],buffer[80];
      int err,div,fd_ans;
      long num;
      pid_t self=getpid();
      FILE *ansfifo;

      sprintf(ansname,ANSFIFO".%d",self);
      if (mkfifo(ansname,0666) || (fd_ans=open(ansname,O_RDWR))<0 )
       { perror("can't create answer fifo"); exit(0); }
      ansfifo=fdopen(fd_ans,"rw");

      num=abs(atol(argv[i]));
      fprintf(cmdfifo,"%d:%ld\n",self,num);
      fflush(cmdfifo);

      if (fscanf(ansfifo,"%d",&div)!=1)
         perror("no answer");
      else
       {
         if (div) printf("%d is divisible by %d\n",num,div);
```

```
        else printf("%d is prime\n",num);
       }
      fclose(ansfifo);
      unlink(ansname);
    }
   fclose(cmdfifo);
 }
```

Der Server legt zunächst die Kommando-FIFO an und öffnet sie zum Lesen und zum Schreiben. Dadurch gibt es immer einen schreibenden Prozess, sodass die FIFO beim Lesen nie ein EOF meldet.

Der Server läuft in einer Endlosschleife und kann nur durch ein Signal abgebrochen werden. Damit auch bei einem CTRL-C die Kommando-FIFO weggeräumt wird, müssen wir einen Signal-Handler schreiben.

```
#include "fifoname.h"

static FILE *cmdfifo;

static void sigint_handler(int sig)
 {
   fclose(cmdfifo);
   unlink(REQFIFO);
   exit(0);
 }

static unsigned long divisor(unsigned long l)
 {
   unsigned long d,s;
   if (l<4) return (l<=1);
   if ((l&1)==0) return 2;
   s=(unsigned long)sqrt((double)l);
   for (d=3;d<=s;d+=2) if (l%d==0) return d;
   return 0;
 }

int main()
 {
   int fd;
   signal(SIGINT,sigint_handler);

   if ( mkfifo(REQFIFO,0666) || (fd=open(REQFIFO,O_RDWR))<0 )
    { perror("can't create " REQFIFO); sigint_handler(SIGINT); }
   cmdfifo=fdopen(fd,"rw");

   for (;;)
    {
      char buffer[256], *P=buffer;
      fgets(buffer,256,cmdfifo);
```

```
      P=strchr(buffer,':');
      if (P==0)
         fprintf(stderr,"illegal request: %s\n",buffer);
      else
       {
         int fd2;
         unsigned long num;

         *P=0;
         num=atoi(P+1);
         sprintf(buffer,ANSFIFO".%d",atoi(buffer));
         if ((fd2=open(buffer,O_WRONLY))<0)
            fprintf(stderr,"can't open %s\n",buffer);
         else
          {
            FILE *ansfifo=fdopen(fd2,"w");
            fprintf(ansfifo,"%d\n",divisor(num));
            fclose(ansfifo);
          }
       }
    }
}
```

9.2 IPC-Mechanismen

Es gibt wenig sinnvolle Synchronisations-Mechanismen, die in POSIX definiert sind (außer dem Record Locking, s.u.). System V definiert allerdings gleich drei davon, die sich im Lauf der Zeit über fast alle UNIX-Versionen (auch L*i*nux) verbreitet haben:

- `msg_`: Nachrichtenaustausch (*Message Passing*)
- `sem_`: Semaphore
- `shm_`: gemeinsamer Speicher (*Shared Memory*)

Es wird jeweils ein Satz von Operationen zur Verfügung gestellt, der aber immer eine Funktion `_get` (Erfragen/Anlegen eines IPC-Objekts) und eine Funktion `_ctl` (IPC-Kontrolle) enthält, beispielsweise

```
int msgget(key_t key, int flg);
```

Ein IPC-Objekt (aller drei Typen) wird durch einen Schlüssel vom Typ `key_t` (aus `sys/ipc.h`) identifiziert. Die `_get`-Funktionen verhalten sich dabei alle wie folgt:

Falls `key`=`IPC_PRIVATE`, wird ein neues Objekt angelegt, und die Funktion gibt seine ID zurück.

Falls `key`≠`IPC_PRIVATE`:
Falls es ein IPC-Objekt des jeweiligen Typs mit dem Schlüssel `key` gibt, wird dessen ID zurückgegeben.
Ansonsten, falls das Bit `IPC_CREAT` in `flg` gesetzt ist, wird ein neues Objekt kreiert und dessen ID zurückgegeben.
Sonst wird mit Fehlerstatus abgebrochen.

- Eigentlich waren die _get-Aufrufe so gedacht, dass sich die kommunizierenden Prozesse von vornherein auf einen bestimmten Schlüssel festlegen. Der erste Prozess (z.B. ein Server) legt dann das IPC-Objekt an (mit `IPC_CREAT`), die anderen erfragen (ohne `IPC_CREAT`) dann nur noch die ID des schon existierenden Objekts.

  Das funktioniert allerdings nur, wenn systemweit die Verwendung der Schlüssel festgelegt ist, sodass es nicht zu Überschneidungen kommen kann. Deshalb geht man meist einen anderen Weg:

  Der Server-Prozess legt (mit `IPC_PRIVATE`) das IPC-Objekt an und legt dessen ID in einer vorbestimmten Datei ab („Lockfile“), aus der sie sich die Server-Prozesse wiederum abholen. So werden wir meist arbeiten.

- Es gibt außerdem die Bibliotheksfunktion `ftok` (File-To-Key). Sie errechnet aus dem Pfadnamen einer existierenden Datei einen Schlüssel vom Typ `key_t`:

  UNIX

  `key_t ftok(char *pathname, char proj);`
  errechnet aus dem Pfadnamen `pathname` und dem Projektnamen `proj` einen Schlüssel vom Typ `key_t`. Es gehen dabei Gerätenummer des beteiligten Dateisystems und I-Number der Datei ein. Die Einzigartigkeit des Schlüssels im ganzen System ist *nicht* garantiert, aber relativ wahrscheinlich.

  Alle beteiligten Prozesse müssen sich dennoch auf eine existierende Datei einigen, beispielsweise:

  ```
  key_t key=ftok("/tmp/mylock",'X');
  ```

- Jedes Objekt hat Besitzer und Zugriffsrechte analog zu Dateien. Insbesondere gibt es also einen Besitzer, Gruppenbesitzer, Erzeuger und Gruppen-Erzeuger. Die unteren 9 Bits von `flg` geben die Zugriffsrechte zu Beginn an. Besitzer und Rechte können mit _ctl-Aufrufen geändert werden.

  Gelöscht werden IPC-Objekte mit einem speziellen _ctl-Aufruf (nur vom Besitzer).

- Das Kommando `ipcs` (IPC-Status) zeigt Informationen zu allen im Moment im System befindlichen IPC-Objekte an:

  ```
  ------ Message Queues --------
  msqid      owner      perms      used-bytes  messages
  128        root       700        0           0

  ------ Semaphore Arrays --------
  semid      owner      perms      nsems      status
  3          root       0          1

  ------ Shared Memory Segments --------
  shmid      owner      perms      bytes      nattch     status
  0          nobody     600        46084      6          dest
  ```

  Mit den Optionen „`-s`“, „`-q`“ bzw. „`-m`“ kann man die Ausgabe auf eine Sektion einschränken.

Das Kommando `ipcrm` löscht ein IPC-Objekt, die Berechtigung dazu vorausgesetzt. Es wird die Art des Objekts (`sem`, `msg` oder `shm`) und die ID angegeben:

```
ipcrm sem 3
```

9.3 Message Queues

Es gibt eine Möglichkeit zur indirekten IPC über den Austausch von Nachrichten per Mailbox, die sogenannte **Message Queue**, die von beliebig vielen Prozessen aus erreichbar ist. Alles, was mit den Message-Operationen möglich ist, lässt sich auch über FIFOs realisieren. Programme werden aber oft übersichtlicher als eine gleichwertige FIFO-Version.

Strukturen, Konstanten und Aufrufe sind in `sys/ipc.h` und `sys/msg.h` definiert. Die Nachrichten haben auf Benutzer-Ebene folgenden Aufbau:

```
struct msgbuf
 {
    long mtype;          // Nachrichten-Typ, >0
    char mtext[1];       // eigentlich variabel langes Array
 };
```

Wie lang die Nachricht tatsächlich ist, muss beim Senden explizit angegeben werden. Am besten definiert man eine eigene Struktur, die den zu verschickenden Daten angepasst ist, und ermittelt mit **`sizeof`** ihre Länge.

Es gibt folgenden Satz von Systemaufrufen zu Messages:

UNIX
`int msgget(key_t key, int flg);` **get** message queue, erfragt die ID einer bestehenden Message-Queue oder legt eine neue an (siehe 9.2).
`int msgctl(int qid, int cmd, struct msqid_ds *buf);` message queue **control**, dient zum direkten Auslesen oder Einschreiben von Teilen der Message-Queue-Struktur. Am wichtigsten ist das Kommando `cmd=IPC_RMID` zum Löschen der Queue.
`int msgsnd(int qid, struct msgbuf *p, int sz, int flg);` message **send**, sendet eine Nachricht an die durch `qid` bezeichnete Queue. `mp` zeigt auf die Daten, `sz` gibt die tatsächliche Länge an, `flg` steuert das Verhalten übergroßer Nachrichten. Wenn die Queue voll ist, blockiert der Aufruf, bis sie verkleinert, gelöscht oder der Prozess unterbrochen wird.
`int msgrcv(int qid, struct msgbuf *mp, int sz, long typ, int flg);` message **receive**, holt eine Nachricht aus der Queue `qid` ab (sie wird aus der Queue entfernt). Es wird folgende Message abgeholt: `typ`=0: die vorderste (also älteste) Message `typ`>0: die vorderste Message dieses Typs – es sei denn, `MSG_EXCEPT` ist in `flg` gesetzt, dann die vorderste *nicht* dieses Typs! `typ`<0: die vorderste mit dem kleinsten Typ $\leq$ |`typ`| Wenn `IPC_NOWAIT` in `flg` gesetzt ist, wird nicht auf eine passende Nachricht gewartet, sondern mit -1 abgebrochen.

Beispiel: Wir formulieren unser Client-Server-Beispiel aus dem FIFO-Abschnitt 9.1 für Message Queues um. Wenn wir schon eine Message-Struktur verschicken, arbeiten wir diesmal direkt mit binären Daten.

Wir brauchen nur eine Queue überhaupt. Die Messages enthalten ein Feld mit der PID der Clients, und die Clients sind so nett und holen sich nur Nachrichten ab, die für sie bestimmt sind. Zusätzlich benötigen wir aber ein Kommunikations-File, in dem der Server den Schlüssel der Queue ablegt, die er angelegt hat.

Unsere genaue Message-Struktur und den Namen des Files definieren wir in einer gemeinsamen Header-Datei `divq.h`:

```
#define QIDFILE "/tmp/divqid"

struct mymsgbuf
 {
   long mtype;
   pid_t pid;
   unsigned long data;
 };
```

Der Client holt sich als erstes die Nummer der Queue aus dem Kommunikationsfile. Danach braucht er nur noch je einen Send- und Receive-Aufruf:

```
#include "divq.h"

int main(int argc, char *argv[])
 {
   int fd,i,qid;
   struct mymsgbuf msg;
   pid_t self=getpid();

   if ((fd=open(QIDFILE,O_RDONLY))<0)
    { fputs("server not running\n",stderr); exit(0); }
   read(fd,&qid,sizeof(qid));
   close(fd);

   for ( i=1 ; i<argc ; ++i )
    {
      long num=abs(atol(argv[i]));
      msg.mtype=1;
      msg.pid=self;
      msg.data=abs(num);
      if ( msgsnd(qid,(struct msgbuf *)&msg,sizeof(msg),0)
        || msgrcv(qid,(struct msgbuf *)&msg,sizeof(msg),self,0)==0)
       { perror("queue error"); exit(0); }

      if (msg.data) printf("%d is divisible by %d\n",num,msg.data);
      else printf("%d is prime\n",num);
    }
 }
```

Der Server vereinfacht sich ebenfalls ein wenig (die Funktion `divisor` ist ausgelassen):

```
#include "divq.h"

static int qid;

static void sigint_handler(int sig)
 {
   if (qid>0) msgctl(qid,IPC_RMID,0);
   unlink(QIDFILE);
   exit(0);
 }

int main()
 {
   int fd;
   struct mymsgbuf msg;
   signal(SIGINT,sigint_handler);

   if ((qid=msgget(IPC_PRIVATE,0666))<=0
      || (fd=open(QIDFILE,O_WRONLY|O_CREAT,0644))<0)
    { perror("can't open queue\n"); sigint_handler(SIGINT); }
   write(fd,&qid,sizeof(qid));
   close(fd);

   for (;;)
    {
      msgrcv(qid,(struct msgbuf *)&msg,sizeof(msg),1,0);
      msg.mtype=msg.pid;
      msg.data=divisor(msg.data);
      msgsnd(qid,(struct msgbuf *)&msg,sizeof(msg),0);
    }
 }
```

Während unser Server läuft, erscheint er auch in der Liste von „`ipcs -q`“, z.B.:

```
------ Message Queues --------
msqid     owner     perms     used-bytes  messages
128       root      700       0           0
2178      axel      666       12          0
```

Um die Anzeige „12 Bytes“ zu erhalten, wurde der Server kurzzeitig mit `CTRL-Z` gestoppt.

9.4 Semaphore

Die Werte von System-V-Semaphoren sind vom Typ **`unsigned short`**, der maximale Wert ist `SEMVMX` (aus `sys/sem.h`, z.B. 32767). Man kann allerdings gleich mit einer ganzen *Menge* von Semaphoren in einem Aufruf arbeiten (UND-Synchronisation, siehe Seite 197). Das macht z.B. immer Sinn, wenn man mit mehreren verknüpften Ressourcen arbeitet.

Die verwendeten Funktionen und Strukturen sind in `sys/sem.h` und `sys/ipc.h` definiert.

UNIX
`int semget(key_t key, int nsems, int flg);` **get** semaphore, erfragt die ID einer bestehenden Semaphor-Menge oder legt eine neue an (siehe 9.2). `nsems` gibt die Anzahl der Elemente an. Die Semaphor-Werte sind nach dem Aufruf uninitialisiert!
`int semctl(int semid, int semnum, int cmd, union semun arg);` semaphore **control**, führt eine Kontroll-Operation auf der Semaphor-Menge mit der ID `semid` aus. `semnum` selektiert ggf. einen einzelnen Semaphor. `cmd` wählt das genaue Kommando aus. `arg` enthält Kommando-spezifische Argumente.
`int semop(int semid, struct sembuf *sops, unsigned int nsops);` semaphore **operation**, führt mehrere Semaphor-Operationen atomar aus. `sops` ist ein Zeiger auf ein Array mit den Operations-Beschreibungen, `nsops` die Länge des Arrays. Nur wenn alle Operationen erfolgreich *wären*, werden sie auch überhaupt durchgeführt.

9.4.1 `semget`

Ein Server-Prozess könnte wie folgt eine Menge von acht Semaphoren anlegen und die erhaltene ID durch ein Lockfile bekanntgeben:

```
int semid=semget(IPC_PRIVATE,8,0666|IPC_CREAT);
int lockfile=open("/tmp/semtest_lock",O_WRONLY|O_CREAT);
if (lockfile<=0) exit(0);
write(lockfile,&semid,sizeof(int));
close(lockfile);
```

Ein Client-Prozess erhält den Zugriff auf diese Menge wie folgt über das Lockfile:

```
int semid;
int lockfile=open("/tmp/semtest_lock",O_RDONLY);
if (lockfile<=0) exit(0);
read(lockfile,&semid,sizeof(int));
close(lockfile);
```

`ipcs` gibt in etwa Folgendes aus:

```
------ Semaphore Arrays --------
semid     owner     perms     nsems     status
1         axel      0         8
```

Leider sind die neuen Semaphoren nicht direkt im `semget`-Aufruf mit Werten belegbar. Sie werden alle mit 0 initialisiert, was üblicherweise bedeutet, dass die Ressourcen, die sie schützen, als „belegt“ markiert sind.

Ist das (wie meistens) nicht gewünscht, muss man danach einen `semctl`-Aufruf (s.u.) tätigen. Leider kann der Prozess dann aber zwischen den beiden Aufrufen bereits unterbrochen werden! Das könnte zu großen Problemen führen.

In unserem Server-Beispiel lässt sich das glücklicherweise dadurch lösen, dass die ID einfach erst dann in die Datei geschrieben wird, wenn die Semaphore ihre Werte erhalten haben.

9.4.2 semctl

Diese Funktion ist für diverse Kontroll-Operationen auf einer Semaphor-Menge gedacht. Es gibt u.a. folgende Kommandos (Parameter `cmd`):

`IPC_SET`	Setzen von Besitzern und Zugriffsrechten
`IPC_RMID`	Löschen der ganzen Semaphor-Menge
`GETVAL`	Lesen eines Semaphor-Werts
`GETALL`	Lesen aller Semaphor-Werte
`SETVAL`	Setzen eines Semaphor-Werts
`SETALL`	Setzen aller Semaphor-Werte
`GETPID`	Lesen der PID des Prozesses, der zuletzt auf den Semaphor Nummer `semnum` zugegriffen hat

Die verwendete Struktur `semun` ist wie folgt aufgebaut:

```
union semun
 {
   int val;                    // Wert beim Kommando SETVAL
   struct semid_ds *buf;       // Werte bei IPC_STAT und IPC_SET
   ushort *array;              // Puffer bei GETALL und SETALL
 };
```

Wir schauen uns hier nur die einfachste Operation an, nämlich das **Löschen** einer Semaphor-Menge. Der Parameter `semnum` wird ignoriert, die ganze Menge wird gelöscht. `arg` wird auch nicht benötigt, ein entsprechender Parameter muss leider dennoch übergeben werden:

```
union semun dummy;
semctl(semid,0,IPC_RMID,dummy);
```

9.4.3 semop

Hiermit führt man `up`- und `down`-ähnliche Operationen auf einer Semaphor-Menge aus. Es wird ein ganzes Array von Operationen angegeben, und es ist garantiert, dass sie insgesamt atomar ausgeführt werden.

Die Funktion `semop` erhält als Parameter ein Array `sops` (der Länge `nsops`) aus Objekten der folgenden Art:

```
struct sembuf
 {
   short sem_num;      // Nummer des Semaphors im Array, ab 0
   short sem_op;       // Art der Operation
   short sem_flg;      // Flags: IPC_NOWAIT oder SEM_UNDO
 };
```

Die Werte von `sem_op` haben folgende Auswirkungen (Lese- bzw. Schreiberlaubnis des jeweiligen Prozesses vorausgesetzt):

- `sem_op`>0: Dieser Wert wird auf den Wert des Semaphors addiert. Ein Wert von 1 entspricht also einem `up` im üblichen Sinn. Der Aufruf blockiert entsprechend *nie*.
- `sem_op`=0: Der Prozess blockiert so lange, bis der Wert des Semaphors 0 erreicht.

- `sem_op`<0: Der Absolutwert |`sem_op`| wird subtrahiert. Ein Wert von -1 entspricht also einem `down`.

  Würde dabei der Wert kleiner als 0, blockiert der Prozess, bis der Semaphor mindestens den Wert |`sem_op`| angenommen hat, und erst dann findet die Subtraktion statt!

Blockaden werden auch aufgehoben, wenn der Prozess ein Signal erhält, oder wenn der Semaphor zwischendurch zerstört wird (`semop` liefert dann -1 zurück)!

Die Flags haben folgende Bedeutung:

- Wenn `IPC_NOWAIT` gesetzt ist, schlagen `semop`-Aufrufe fehl, bei denen der Prozess normalerweise blockieren würde.
- Bei gesetztem `SEM_UNDO` werden Änderungen an dem Semaphor „geloggt“ und bei Beendigung des Prozesses wieder rückgängig gemacht. Das ist eine Hilfe, falls ein Prozess ein `SIGKILL`-Signal erhält und nicht mehr selbst aufräumen kann. Es könnten sonst Ressourcen fälschlicherweise als belegt markiert bleiben.

`up` und `down` sind also nicht direkt implementiert, sondern lassen sich mit `semop` folgendermaßen nachbilden:

```
static void sem_up_down(int id, int value)
 {
   static struct sembuf semaphor;
   semaphor.sem_op=value;
   semaphor.sem_flg=SEM_UNDO;
   if (semop(id,&semaphor,1)) { perror("down"); exit(10); }
 }

inline void sem_down(int id) { sem_up_down(id,-1); }
inline void sem_up(int id)   { sem_up_down(id,1);  }
```

9.4.4 Eine Klasse für Semaphore

Sehr oft braucht man nur ganz grundlegende Semaphor-Operationen und ist gezwungen, die sehr allgemein gehaltenen Systemaufrufe mit diversen Parametern zu versehen. Als Beispiel stellen wir daher hier eine einfache `C++`-Klasse vor, die die wichtigsten Mechanismen zur Verfügung stellt.

Wir implementieren dabei die einfachen `up` und `down`, wie auch die simultanen Operationen `sup` und `sdown`. Letztere nehmen beliebig viele Semaphor-Nummern als Argumente, die Anzahl ist das erste Argument, also beispielsweise `sdown(3, 0,3,7);`

Zunächst folgt die **Header-Datei** `semarray.h`:

```
class semarray
 {
   private:
      bool creator;                          // neu angelegt oder nicht?
      int semid,numsem;                      // System-ID, Anzahl
```

```
    const char *lockfile;                          // Name des Lockfiles
    struct sembuf *sembuffer;                      // Puffer fur semop
    void init_creator(const char *, int);
    void sem_up_down(int semnum, int value);
    void sdown_sup(va_list &va, int size, int value);

  public:                                          // Konstruktoren/Destruktor
    semarray(const char *lockfile, int size, int value);
    semarray(const char *lockfile, int size, const int *valarray);
    semarray(const char *lockfile);
    ~semarray();

    int id() const { return semid; }               // ID-Abfrage
    void down(int num=0) { sem_up_down(num,-1); }  // down für einen Sem.
    void up(int num=0) { sem_up_down(num,1); }     // up für einen Sem.
    void sdown(int size, ...);                     // simultanes down
    void sup(int size, ...);                       // simultanes up
};
```

Verschiedene Prozesse können sich natürlich *nicht* solche Objekte teilen, auch wenn zwei Objekte intern auf dieselben Semaphore verweisen können. Es gibt daher zwei verschiedene Möglichkeiten, ein `semarray`-Objekt anzulegen:

- als Erzeuger (die ersten beiden Konstruktoren). Hier kann man direkt alle Semaphore der Menge mit Werten versehen – entweder alle mit demselben oder mit Werten aus einem `int`-Array.
- als Teilhaber (der dritte Konstruktor). Als Parameter wird der Name eines Lockfiles erwartet, in dem die ID einer bestehenden Menge abgelegt ist. Dieser Konstruktor testet mit `semctl`, ob es auch eine lesbare Semaphor-Menge mit der angegebenen ID gibt.

Ob die Menge neu erzeugt oder mitverwendet wurde, wird in `creator` gespeichert. Nur der Erzeuger löscht sie im Destruktor auch automatisch wieder.

Die Funktionen `down` und `up` sind wieder wie oben implementiert, wobei jetzt zusätzlich die Nummer des Semaphors in der Menge angegeben werden muss. Für `sdown` und `sup` wird ein Array von `struct sembuf` benötigt, das einmal im Konstruktor angelegt wird.

Ein **Server**-ähnlicher Prozess, der nur einen Semaphor z.B. für einen kritischen Bereich benötigt, arbeitet dann in etwa wie folgt mit der Klasse:

```
int main()
 {
   semarray sema("/tmp/semarraylock",1,1);
   for (;;)
    {
      sema.down();
      puts("critical section");
      sema.up();
      puts("non-critical section");
    }
 }
```

Ein passender **Client** hängt sich dann mit folgender Definition an:

```
semarray sema2("/tmp/semarraylock");
```

Der Rest seines Code könnte dem Server-Code entsprechen.

Es folgt die **Implementationsdatei** `semarray.cpp` für die restlichen Funktionen (die Include-Angaben sind aus Platzgründen in einen Kommentar gewandert). Bei irgendwelchen Fehlern wird hier der ganze Prozess abgebrochen (das könnte man mit Exceptions natürlich eleganter gestalten).

```
// include sys/ipc.h sys/sem.h stdio.h stdlib.h unistd.h fcntl.h string.h semarray.h

void semarray::init_creator(const char *lockname, int size)
 {
   int fd=open(lockname,O_RDONLY);
   close(fd);
   if (fd>=0) { fprintf(stderr,"lock %s exists\n",lockname); exit(1); }

   semid=semget(IPC_PRIVATE,size,0666|IPC_CREAT);
   if (semid<0) { perror("semget"); exit(1); }

   fd=open(lockname,O_CREAT|O_TRUNC|O_WRONLY,0644);
   if (fd<0) { perror(lockname); exit(1); }
   write(fd,&semid,sizeof(semid));
   close(fd);
   lockfile=strdup(lockname);
 }

semarray::semarray(const char *lockname, int size, int value)
 : creator(true), lockfile(0), numsem(size), sembuffer(new(struct sembuf)[size])
 {
   init_creator(lockname,size);
   union semun sem_union;
   sem_union.val=value;
   for (int i=0;i<size;++i)
      semctl(semid,i,SETVAL,sem_union);
 }

semarray::semarray(const char *lockname, int size, const int *valarray)
 : creator(true), lockfile(0), numsem(size), sembuffer(new(struct sembuf)[size])
 {
   init_creator(lockname,size);
   union semun sem_union;
   for (int i=0;i<size;++i)
    {
      sem_union.val=*valarray++;
      semctl(semid,i,SETVAL,sem_union);
    }
 }
```

```
semarray::semarray(const char *lockname) : creator(false), lockfile(0)
 {
   int fd=open(lockname,O_RDONLY);
   if (fd<0) { perror(lockname); exit(1); }
   int id;
   read(fd,&id,sizeof(id));
   close(fd);

   union semun sem_union;
   if (semctl(id,0,GETPID,sem_union)<0) { perror("semctl"); exit(1); }
   semid=id;
   struct semid_ds buffer;
   sem_union.buf=&buffer;
   semctl(id,0,IPC_STAT,sem_union);
   numsem=sem_union.buf->sem_nsems;
   sembuffer=new(struct sembuf)[numsem];
 }

semarray::~semarray()
 {
   if (creator)
    {
      union semun dummy;
      semctl(semid,0,IPC_RMID,dummy);
      if (lockfile) { unlink(lockfile); free((char *)lockfile); }
    }
   delete[] sembuffer;
 }

void semarray::sdown_sup(va_list &ap, int size, int value)
 {
   struct sembuf *P=sembuffer;
   for (int i=0;i<size;++i,++P)
    {
      int semnum=va_arg(ap,int);
      if (semnum>=numsem) { fprintf(stderr,"illegal semaphore\n"); exit(1); }
      P->sem_num=semnum;
      P->sem_op=value;
      P->sem_flg=0;
    }
   va_end(ap);
   if (semop(semid,sembuffer,size)) { perror("semop2"); exit(1); }
 }

void semarray::sdown(int size, ...)
 {
   va_list ap;
   va_start(ap,size);
   sdown_sup(ap,size,-1);
 }
```

```
void semarray::sup(int size, ...)
 {
   va_list ap;
   va_start(ap,size);
   sdown_sup(ap,size,1);
 }

void semarray::sem_up_down(int semnum, int value)
 {
   struct sembuf sbuf={ semnum, value, 0 };
   if (semnum>=numsem) { fprintf(stderr,"illegal semaphore\n"); exit(1); }
   if (semop(semid,&sbuf,1)) { perror("semop1"); exit(1); }
 }
```

Beispiel: Wir implementieren die Semaphor-Lösung für das Philosophen-Problem (Seite 198) mit Hilfe der Klasse `semarray`.

Der erste Prozess legt das Semaphor-Array (fünf Semaphore, einer pro Stäbchen) neu an. Dann erzeugt er vier Kinder, die sich mit dem dritten Konstruktor an das erzeugte Array anhängen.

Die fünf Prozesse bleiben hier nach dem `fork` im selben Code. Sie könnten also über eine globale Variable an die ID der Semaphore kommen. Da üblicherweise aber fremde Prozesse kommunizieren, arbeiten wir hier zur Demonstration dennoch mit der Lösung über das Lockfile.

```
semarray *chopsticks;
void sigint_handler(int) { delete chopsticks; }

int main()
 {
   chopsticks=new semarray("/tmp/philolock",5,1);
   signal(SIGINT,sigint_handler);
   srandom(time(0)+i);

   int i;
   for (i=0;i<4;++i) if (fork()==0) break;
   if (i!=4) chopsticks=new semarray("/tmp/philolock");

   for (;;)
    {
      cout << i << " thinking" << endl;
      sleep(random()&7);
      cout << i << " hungry" << endl;
      chopsticks->sdown(2,i,(i+1)%5);
      cout << i << " eating" << endl;
      sleep(random()&7);
      chopsticks->sup(2,i,(i+1)%5);
    }
 }
```

Wenn das Programm mit `CTRL-C` abgebrochen wird, sorgt der Handler dafür, dass Lockfile und Semaphor-Array freigegeben werden. (In dieser einfachen Version beschweren sich dann evtl. die Kinder, wenn ihnen die Semaphore weggelöscht werden.)

9.5 Shared Memory

Um größere Datenmengen innerhalb eines Ein-Prozessor-Systems auszutauschen, sind FIFOs und Message Queues unverhältnismäßig langsam. Für diesen Fall gibt es die Möglichkeit, einzelne Speicher-Segmente ausnahmsweise für *mehrere* Prozesse zugänglich zu machen („*shared memory segments*“).

Das System verwaltet die Segmente wiederum mit IDs. Für den eigentlichen Zugriff verbindet man den eigenen Prozess mit dem Segment („*attach*“) und erhält dann einen echten Pointer, über den man ganz normal auf den Speicher zugreifen kann. Benötigt man das Segment nicht mehr, koppelt man es wieder vom Prozess ab („*detach*“).

Zugriffe mehrere Prozesse müssen natürlich (z.B. mit Semaphoren) synchronisiert werden. Es handelt sich hier um das typische Readers/Writers-Problem aus 8.3.4.1. Writer brauchen exklusiven Zugriff, während gleichzeitig mehrere Reader erlaubt sind.

Die notwendigen Strukturen und Funktionen sind in `sys/shm.h` und `sys/ipc.h` definiert. Es gibt folgende Systemaufrufe:

UNIX
`int shmget(key_t key, int size, int flg);` **get** shared memory segment, erfragt die ID eines bestehenden Segments oder legt ein neues an (siehe 9.2). `size` gibt die Größe des Segments in Bytes an.
`int shmctl(int id, int cmd, struct shmid_ds *buf);` shared memory **control**, zur Manipulation eines Segments (`cmd=IPC_RMID` zum Löschen).
`char *shmat(int id, char *addr, int flg);` shared memory **attach**, Ankoppeln des Segments `id` an den aktuellen Prozess. Man erhält einen Pointer zurück, über den man in das Segment schauen und schreiben kann. `addr` ist ein Adressvorschlag, der vom System meist ignoriert wird (am besten `addr=0`). Durch Setzen von `SHM_RDONLY` in `flg` kann das Segment schreibgeschützt werden.
`int shmdt(char *addr);` shared memory **detach**, Abkoppeln des Segments ab der Adresse `addr` vom aktuellen Prozess.

Die Segmente können größer sein als angefordert – `size` wird immer auf ein ganzzahliges Vielfaches von `PAGE_SIZE` (z.B. 4 KByte) aufgerundet. Die maximale Größe eines Segments ist `SHMMAX` (z.B. 32 MByte).

Jedes Segment zählt mit, an wieviel Prozesse es aktuell angekoppelt ist. Es wird nicht automatisch gelöscht, wenn es keinen Besitzer mehr hat, erst wenn der Hauptspeicher wiederverwendet werden muss.

Die Segmente werden bei einem `fork` an das Kind vererbt. Bei `exec` und `_exit` dagegen werden sie abgekoppelt.

Das Anlegen eines Segments für 256 `double`-Zahlen und Beschreiben mit Zufallszahlen könnte also beispielsweise wie folgt geschehen:

```
double *data,*P;
int i,shmid;

shmid=shmget(IPC_PRIVATE,sizeof(double[256]),0644|IPC_CREAT);
data=(double *)shmat(shmid,0,0);
if (data==(double*)-1) { perror("sorry"); exit(0); }
for (i=256,P=data;i>0;--i) *P++=drand48();
```

„`ipcs -m`“ liefert als Ausgabe etwas wie:

```
------ Shared Memory Segments --------
shmid      owner      perms      bytes      nattch     status
1          axel       644        2048       1
```

Bei Beendigung des Prozesses wird das Segment automatisch abgekoppelt. Wenn es bereits vorher nicht mehr benötigt wird, sollte man es aber natürlich explizit abkoppeln und freigeben:

```
if (data!=0) shmdt((void *)data);
if (shmid>=0) shmctl(shmid,IPC_RMID,0);
```

Mit `shmctl` können außerdem Status und Besitzer abgefragt werden, und der Superuser kann das Segment sperren und wieder freigeben.

9.5.1 Eine Klasse für exklusive Segmente

Ganz analog zur Klasse für Semaphoren stellen wir nun noch eine einfache Klasse für gemeinsame Speichersegmente vor. Sie nehmen einem das Ankoppeln und Entkoppeln, sowie zusätzliche Verwaltungsarbeit mit Semaphoren ab. Sie sind der Einfachheit halber für den exklusiven Zugriff (beim Lesen und beim Schreiben) gedacht.

Die Objekte lassen sich dazu mit den Funktionen `lock` und `unlock` sperren bzw. freigeben. Dazu verwendet die Klasse intern Locks in Form binärer Semaphore aus unserer Klasse `semarray`.

Es gibt wiederum einen Konstruktor, der das Segment neu anlegt, und einen, der über eine bekannte ID auf ein bestehendes Segment zugreift. In das Segment werden vorne die ID des verwendeten Semaphors und bei Bedarf die ID des Prozesses eingeschrieben, der zuletzt Zugriff hatte (auf diese Weise können Client und Server zusammenfinden).

Es folgt zunächst die Header-Datei `comseg.h`:

```
class comseg
 {
   private:
      bool creator;                          // neu angelegt?
      semarray *sem;                         // Semaphor-Menge
      struct da_tag
       {
         pid_t pid;                          // letzte Prozeß-ID
         char data[0];                       // eigentliche Daten
       } *data_area;                         // Segment-Inhalt
```

```
  public:                                              // neu anlegen, mit Größe
     comseg(const char *lockfile, unsigned long datasize, int mode);
     comseg(const char *lockfile, int mode);          // mitverwenden
     ~comseg();                                       // Destruktor
     int id() const { return shm_id; }                // ID auslesen
     void setpid(pid_t pid) { data_area->pid=pid; }   // PID einschreiben
     pid_t pid() const { return data_area->pid; }     // PID auslesen
     char *data() const { return data_area->data; }   // Zeiger auf Daten lesen
     void lock() { sem->down(); }                     // Zugriff sperren
     void unlock() { sem->up(); }                     // Zugriff freigeben
};
```

Die Implementationsdatei comseg.cpp birgt keine Überraschungen:

```
// include sys/types sys/ipc sys/shm sys/stat stdio unistd fcntl
// include semarray comseg

comseg::comseg(const char *lockname, unsigned long datasize, int mode)
 : creator(true)
{
  struct stat statbuf;
  if (stat(lockname,&statbuf)>=0)
   { fprintf(stderr,"%s exists\n",lockname); exit(1); }

  shm_id=shmget(IPC_PRIVATE,sizeof(*data_area)+datasize,0666|IPC_CREAT);
  if (shm_id<0) { perror("shmget"); exit(0); }
  data_area=(da_tag *)shmat(shm_id,0,mode);            // Ankoppeln
  if (data_area==(da_tag *)-1)
   { shmctl(shm_id,IPC_RMID,0); perror("shmat"); exit(0); }

  sem=new semarray(lockname,1,1);                      // Semaphor neu anlegen

  int fd=open(lockname,O_RDWR|O_APPEND);
  if (fd<0) { perror("open"); exit(1); }
  write(fd,&shm_id,sizeof(shm_id));
  close(fd);
}

comseg::comseg(const char *lockname, int mode) : creator(false)
{
  sem=new semarray(lockname);                          // Semaphor mitverwenden

  int fd=open(lockname,O_RDONLY); if (fd<0) { perror("open"); exit(1); }
  lseek(fd,sizeof(int),SEEK_SET); read(fd,&shm_id,sizeof(shm_id));
  close(fd);

  data_area=(da_tag *)shmat(shm_id,0,mode);     // Ankoppeln
  if (data_area==(da_tag *)-1) { perror("shmat"); exit(0); }
}
```

```
comseg::~comseg()
 {
   shmdt((char *)data_area);                    // Abkoppeln
   if (creator) shmctl(shm_id,IPC_RMID,0);      // nur der Erzeuger gibt frei
   delete sem;                                  // Semaphor freigeben
 }
```

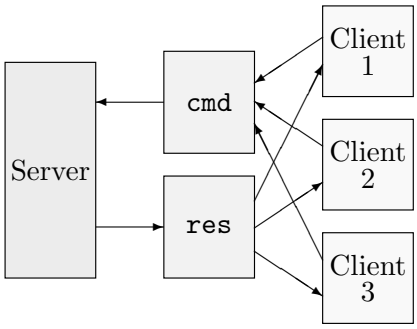
Im nächsten Abschnitt folgt direkt ein ausführliches Beispiel für die Verwendung der Klasse.

9.5.2 Client/Server-Beispiel mit Shared Memory

Wir verwenden unsere Klasse `comseg` in einem Client–Server-Einsatz. Der Server dient diesmal dazu, „lange“ natürliche Zahlen miteinander zu multiplizieren (unsere Puffer sind hier auf ca. 2048 Stellen beschränkt). Schon hier – und noch weniger bei noch größeren Datenmengen – macht es keinen Sinn, die Daten durch FIFOs oder Message Queues zu schleusen.

In unserem FIFO-Beispiel gab es *eine* Ergebnis-FIFO *pro Client*, im Beispiel mit den Message Queues nur *eine* Queue *insgesamt*. Hier richten wir zwei Shared-Memory-Segmente ein:

- Das Segment `cmd` mit der Anfrage (zwei Zahlen mit max. ca. 2000 Stellen) wird vom Client gelockt, bevor er die beiden Faktoren hineinschreibt, und vom Server entlockt, nachdem er diese Daten entnommen hat.
- Das Segment `res` mit dem Ergebnis (das Produkt mit max. ca. 4000 Stellen) wird vom Server gelockt, bevor er das Produkt hineinschreibt, und vom Client entlockt, wenn er es entnommen hat.



Clients und Server verwenden außerdem Signale:

- Der Client schickt dem Server `SIGUSR1`, wenn er seine zwei Zahlen nach `cmd` geschrieben hat. Dann wartet er mit `pause` auf ein Antwort-Signal vom Server und holt das Ergebnis aus `res` ab.
- Der Server hängt fast immer an einem `pause`-Aufruf, bis er ein Signal von einem Client erhält. Er holt das Kommando, rechnet und schickt diesem Client `SIGUSR1`, wenn er das Ergebnis nach `res` geschrieben hat.

Wenn man zusätzliche Eingaben in das Programm einbaut (wie „`mit dem Schreiben beginnen <RET>?`“), die vor den Lese- und Schreiboperationen anhalten, kann man verfolgen, wie ein Client mit einem neuen Kommando an einem Semaphor angehalten wird, weil der Server die Seite noch nicht freigegeben hat, etc.

Client und Server verwenden folgende Header-Datei `lmcs.h` mit den Namen der Lockfiles und der Datengröße des Segments:

```
static const char LM_NAME1[]="/tmp/lm_lock1";
static const char LM_NAME2[]="/tmp/lm_lock2";
#define DATA_SIZE 4088
```

Die eigentliche Rechnung lagern wir in ein Modul `longmul.o` aus, das über folgende Header-Datei vom Server verwendet wird:

```
// longmul.h
extern char *long_mul(const char *, const char *);
```

`long_mul` verändert also seine Argumente nicht und legt selbständig den Speicher für das Resultat an, und zwar mit `calloc`, da **new** den Speicher nicht mit Nullen vorbelegt. Der Server gibt den Speicher nach Verwendung mit **free** frei. Die Implementation wird der Vollständigkeit halber am Ende dieses Abschnitts nachgeliefert.

Es folgt zunächst der Quelltext des Servers:

```
// include sys/ipc sys/shm iostream stdlib string signal fcntl unistd ctype
// include lmcs longmul semarray comseg

comseg *cmd, *res;
bool answer=false;

void cleanup() { delete cmd; delete res; }

void error(char *str) { cerr << "server error: " << str << endl; exit(1); }

void sigint_handler(int s) { cerr << "server interrupted" << endl; exit(1); }

void sigusr1_handler(int s) { answer=true; }

int isnumber(char *P)
 {
   char c;
   while ((c=*P++)!='\0') if (!isdigit(c)) return 0;
   return 1;
 }

int main()
 {
   atexit(cleanup);
   signal(SIGINT,sigint_handler);

   static struct sigaction sa;
   sa.sa_handler=sigusr1_handler;
   sa.sa_flags=SA_RESTART;
   sigaction(SIGUSR1,&sa,&sa);

   cmd=new comseg(LM_NAME1,DATA_SIZE,0);
   res=new comseg(LM_NAME2,DATA_SIZE,0);
   res->setpid(getpid());

   for (;;)
    {
      pid_t client_pid;
      char *num1, *num2;
      if (!answer) pause();
      client_pid=cmd->pid();
      num1=cmd->data();
```

```
      num2=num1+strlen(num1)+1;

      res->lock();
      if (!isnumber(num1)||!isnumber(num2))
         strcpy(res->data(),"illegal operands");
      else
       {
         char *num3=long_mul(num1,num2);
         strcpy(res->data(),num3);
         free(num3);
       }
      cmd->unlock();
      answer=false;
      kill(client_pid,SIGUSR1);
    }
 }
```

Das Signal `SIGUSR1` wird mit einem Handler abgefangen, der hinter den `pause`-Aufruf zurückkehrt. Der Server läuft ewig und muss mit `CTRL-C` abgebrochen werden. Daher wird auch `SIGINT` abgefangen und führt zum Abbruch. Mit `atexit` wird noch die Funktion `cleanup` eingehängt, die vor dem Programmende noch aufräumt (Speichersegmente und damit die Semaphore freigibt und die Lockfiles löscht).

Der Client ist der Einfachheit halber so aufgebaut, dass er am Anfang des Programms die Speichersegmente ankoppelt und bis zum Ende behält. Wenn der Server zwischendurch abgebrochen wird, bekommt der Client es nicht mit und erzeugt erst beim Zugriff auf die nicht mehr vorhandenen Segmente einen Fehler.

Wenn man es ganz sauber realisieren wollte, müsste der Server alle Clients zunächst *registrieren* und sie bei seinem Abbruch mit einem Signal benachrichtigen.

```
// include sys/ipc sys/shm iostream stdlib string signal fcntl unistd
// include lmcs comseg semarray

comseg *cmd, *res;
bool answer=false;

void cleanup() { delete cmd; delete res; }

void error(char *str) { cerr << "client error: " << str << endl; exit(0); }

void sigint_handler(int s) { cerr << "client interrupted" << endl; exit(0); }

void sigusr1_handler(int s) { answer=true; }

int main()
 {
   atexit(cleanup);
   signal(SIGINT,sigint_handler);

   static struct sigaction sa;
   sa.sa_handler=sigusr1_handler;
   sa.sa_flags=SA_RESTART;
   sigaction(SIGUSR1,&sa,&sa);
```

```
  cmd=new comseg(LM_NAME1,0);
  res=new comseg(LM_NAME2,0);
  res->lock();
  pid_t server_pid=res->pid();
  res->unlock();

  for (;;)
   {
     static char num1[2048], num2[2048];
     int l1,l2;
     cout << "1. Zahl: "; cin.getline(num1,2048);
     cout << "2. Zahl: "; cin.getline(num2,2048);

     num1[l1=strlen(num1)]=num2[l2=strlen(num2)]=0;
     if (l1+l2>DATA_SIZE-2)
        cerr << "combined operands too long" << endl;
     else
      {
        cmd->lock();
        cmd->setpid(getpid());
        strcpy(cmd->data(),num1);
        strcpy(cmd->data()+(l1+1),num2);

        answer=false;
        kill(server_pid,SIGUSR1);
        if (!answer) pause();

        cout << "\n  " << num1
             << "\n* " << num2
             << "\n= " << res->data() << endl << endl;

        res->unlock();
      }
   }
 }
```

Schließlich folgt noch die versprochene Implementation der eigentlichen Multiplikation:

```
// include stdlib string

char *long_mul(const char *f1, const char *f2)
 {
   int n1,n2,nr,i,j;
   char *result,*Pr,*P3;
   const char *P1, *P2;

   nr=(n1=strlen(f1))+(n2=strlen(f2));
   if ((result=(char *)calloc(nr+1,sizeof(char)))==0) return 0;

   for ( i=n2-1 , P2=f2 ; i>=0 ; --i , ++P2 )
    {
      int cyph, carry=0, fac=*P2-'0';
```

```
    for ( j=n1 , Pr=result+nr-i , P1=f1+n1 ; j>0 ; --j )
     {
       cyph=(*--P1-'0')*fac+*--Pr+carry;
       for ( carry=0 ; cyph>=10 ; ++carry ) cyph-=10;
       *Pr=cyph;
     }
    while (carry)
     {
       cyph=*--Pr+carry;
       for ( carry=0 ; cyph>=10 ; ++carry ) cyph-=10;
       *Pr=cyph;
     }
  }

  P2=P3=result;
  while ( *P2==0 && nr>1 ) { ++P2; --nr; }
  for ( i=nr ; i>0 ; --i ) *P3++='0'+*P2++;
  *P3=0;
  return result;
}
```

9.6 Dateisperren

Um Inkonsistenzen in gemeinsam benutzten *Dateien* (z.B. bei Datenbank-ähnlichen Anwendungen) zu vermeiden, gibt es einen POSIX-Mechanismus namens **Record Locking**. Auf diese Weise lässt sich der Einsatz von Semaphoren vermeiden (so denn welche zur Verfügung stehen), Blockade-Situationen müssen aber von den Prozessen selbst gemeistert werden.

Ein Record Lock ist eine Blockierung für einen beliebigen zusammenhängenden *Teil* einer Datei, ggf. natürlich auch einer ganzen Datei.

Es gibt zwei Arten von Locks („Sperren"), „*shared*" und „*exclusive*". Solange auf ein Dateistück ein exklusives Lock existiert, kann es nicht ein zweites Mal gelockt werden. Lock-Bereiche dürfen einander überlappen. Wenn ein Byte Teil eines exklusiv gelockten Bereichs ist, ist dieses Lock das einzige.

Record Locking kann mit einem speziellen `fcntl`-Aufruf erfolgen:

UNIX
int fcntl**(int** fd, F_GETLK**, struct** flock ***flp);** Anfrage, ob das durch `flp` angegebene Lock durch ein bereits bestehendes verhindert würde. Wenn nicht, wird `l_type` auf `F_UNLCK` gesetzt, ansonsten wird `flp` mit den Daten des bestehenden Locks gefüllt.
int fcntl**(int** fd, F_SETLK**, struct** flock ***flp);** Einrichten eines Locks, das durch `flp` definiert wird (oder bei `l_type`=`F_UNLK` Aufheben eines bestehenden Locks). Wenn bereits ein exklusives Lock auf den Bereich besteht, kehrt die Funktion mit -1 zurück.
int fcntl**(int** fd, F_SETLKW**, struct** flock ***flp);** wie `F_SETLK`, aber blockierend ('W'=wait): Wenn bereits ein exklusiver Lock auf den Bereich besteht, wartet die Funktion, bis dieser aufgehoben wird.

Die verwendete Struktur `flock` hat folgenden Aufbau:

```
struct flock
 {
   short l_type;       // Typ: F_RDLCK (shared), F_WRLCK (excl.), F_UNLCK
   short l_whence;     // wie bei lseek: SEEK_SET, SEEK_CUR, SEEK_END
   off_t l_start;      // Offset relativ zu whence
   off_t l_len;        // Länge in Bytes, 0 bedeutet bis Dateiende
   pid_t l_pid;        // wird von F_GETLK eingetragen
 };
```

- Wenn eine Datei durch `l_len`=0 bis zum Ende gelockt wird, soll sich nach der POSIX-Definition das Lock automatisch in der Größe anpassen, wenn die Datei wachsen oder schrumpfen sollte. Das ist aber nicht unbedingt so verwirklicht.
- Bei blockierenden Aufrufen (`F_SETLKW`) ist Vorsicht geboten, da es eventuell zu Deadlocks kommen kann (siehe 8.3.3). Es sollte dann auf jeden Fall mit einem Prozess-Synchronisationsmechanismus (z.B. mit Semaphoren) gearbeitet werden.
- Bei Beendigung eines Prozesses werden automatisch alle von ihm errichteten Locks freigegeben.
- Locks werden *nicht* an Kindprozesse vererbt (Locks sind etwas prozess-spezifisches). Es ist in POSIX nicht festgelegt, was nach einem `exec`-Aufruf mit bestehenden Locks geschieht.

Beispiel: Das folgende simple Programm kann nur in einer Instanz gleichzeitig laufen. Es lockt mit `F_SETLK` eine (existierende) Datei vollständig und exklusiv und wartet dann auf ein Signal. Ein zweiter Aufruf erhält dann keinen Lock mehr, fragt mit `F_GETLCK` genauer nach und gibt die PID des blockierenden Prozesses aus:

```
int main(int argc, char *argv[])
 {
   struct flock flbuf;
   int fd=open("lockme",O_RDWR);
   if (fd<0) { perror("lockme"); exit(0); }

   flbuf.l_type=F_WRLCK;
   flbuf.l_whence=SEEK_SET;
   flbuf.l_start=flbuf.l_len=0;

   if (fcntl(fd,F_SETLK,&flbuf))
    {
      fcntl(fd,F_GETLK,&flbuf);
      printf("schon gelockt von pid %d!\n",flbuf.l_pid);
    }
   else pause();

   close(fd);
 }
```

9.7 Sockets

Unter UNIX kommunizieren Prozesse auf verschiedenen Rechnern standardmäßig über sogenannte **Sockets** („Steckdosen“). Ein Socket ist dabei einfach das Ende eines Kommunikationswegs – die Verbindungsdetails werden vom Kernel und Treibern übernommen. Sockets wurden zuerst in BSD eingeführt, aber in die meisten anderen UNIX-Geschmacksrichtungen (auch L*i*nux) übernommen.

In diesem Kapitel interessieren uns natürlich nicht die technischen Grundlagen dieser Verbindung, sondern die Handhabung der Sockets in Benutzerprogrammen. Die unterliegenden Mechanismen wollen wir hier nur kurz betrachten.

Die Kommunikation findet auf drei unterschiedlichen Ebenen statt (man kann noch feinere Unterteilungen treffen):

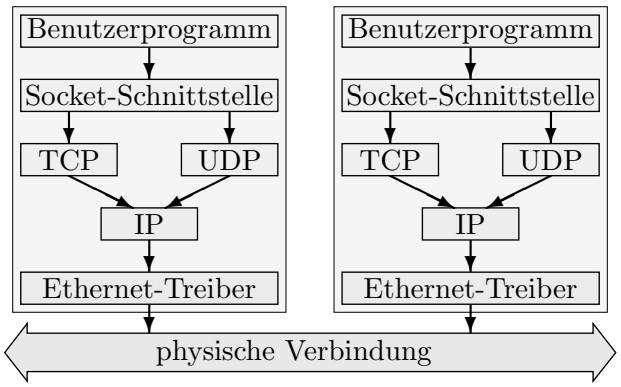
Geräte-Ebene: Das ist die Hardware-nächste Ebene, auf der die Verständigung z.B. der beteiligten Netzwerkkarten abläuft. Daran beteiligt sind die Hardware und deren Treiber im Kernel. In vielen Netzwerken wird „**Ethernet**“ verwendet, eine Sammlung von Hardware- und Übertragungs-Spezifikationen. Es wird so festgelegt, durch welche physischen Signale Bits dargestellt werden, in welchen Gruppierungen, mit welchen Checksummen sie übertragen werden sollen, etc.

Protokoll-Ebene: Auf dieser Ebene werden Datenstrukturen festgelegt, um Übertragungen in einem Netzwerk mit vielen Rechnern richtig zu interpretieren. Die Daten der unterliegenden Ebene werden „eingepackt“ und mit zusätzlichen Informationen (wie Rechner-IDs, Kontrollstrukturen, etc.) versehen.

Sehr häufig wird als ein solches Protokoll **TCP/IP** (zwei Ebenen in sich) eingesetzt. TCP (*Transfer Control Protocol*) auf der „Transport-Ebene“ beschäftigt sich mit der Verwaltung eingehender Datenpakete (Aufteilung in kleine Pakete, Sortieren von Paketen, Checksummen-Kontrolle und ggf. mehrfaches Senden). IP (*Internet Protocol*) auf der „Netzwerk-Ebene“ beschäftigt sich mit dem Routing (Wegfindung) und dem tatsächlichen Verschicken.

Wenn der *Benutzer* die Verwaltung der Pakete übernimmt, erhält man die Kombination **UDP**/IP (*User Datagram Protocol*). TCP ist die sicherere Wahl, da es selbst die Verantwortung für verlorengegangene Pakete etc. übernimmt. UDP ist simpler und kann im Ernstfall schneller sein. Außerdem können mit UDP größere Pakete (bis 8K) verschickt werden, während TCP Daten vergleichsweise klein zerteilt.

Programm-Ebene: Die Benutzerprogramme sollen von den unterliegenden Details natürlich nichts mitbekommen müssen. Dazu dienen die Sockets, die wie Files per Deskriptor gelesen, beschrieben und kontrolliert werden können, und deren Daten vom Kernel transparent an die verwendeten Protokolle und Treiber übergeben werden.



Auf dieser Ebene sind weitere High-Level-Protokolle definiert, wie etwa **HTTP** (*Hyper-Text Transfer Protocol*) zum Austausch von Dokumenten im WWW, **FTP** (*File Transfer Protocol*), Mail-Protokolle, etc.

Die Kommunikation über Sockets spiegelt den Austausch von kleinen Paketen (Datagrammen) auf den unteren Ebenen wieder. Sie ähnelt daher einem Message Passing mit Send- und Receive-Aufrufen. Wenn eine FIFO-ähnliche Verbindung gewünscht ist, müssen die beteiligten Prozesse das selbst verwalten.

9.7.1 Anlegen von Sockets

Die Include-Dateien, die bei Benutzung von Sockets wichtig sind, sind `sys/socket.h` (Socket-Definitionen), `netdb.h` (allgemeine Netzkommunikation), `netinet/in.h` (Internet-Definitionen).

Ein Socket wird meist angelegt durch folgenden Systemaufruf:

	UNIX
`int socket(int domain, int type, int protocol);` legt einen Socket an; gibt einen passenden File-Deskriptor zurück	

Die drei zu übergebenden Attribute bedeuten dabei folgendes:

`domain:` Ein Socket ist immer fest an einen „Lebensbereich“ gebunden, der globale Einstellungen (wie Protokolle, Adressformate, etc.) vorgibt. Innerhalb eines UNIX-Systems kann ein effektiveres Format verwendet werden als in einem heterogenen Netz. Die wichtigsten hier erlaubten Konstanten sind `AF_UNIX` (UNIX-intern) und `AF_INET` (Internet).

`type:` Der Typ legt fest, welcher Art die Verbindung sein soll. Ein Stream-Socket („virtueller Kreis“, `SOCK_STREAM`) ist eine sehr zuverlässige bidirektionale Verbindung, durch die dazu notwendigen Sicherheitsmechanismen aber auch ein wenig langsam. Weitere Typen sind „Datagramm“ (`SOCK_DGRAM`, Verschicken kleiner benutzerdefinierter Packete) und „Roh“ (`SOCK_RAW`, unter Umgehung von Netzwerk-Protokollen).

`protocol:` Wenn in einer Domäne mehr als das Standard-Protokoll erlaubt ist, kann eines ausgewählt werden. Ansonsten wird hier 0 angegeben.

Geschlossen wird ein Socket mit `close`, ein vorzeitiges einseitiges Einstellen der Kommunikation kann mit `shutdown` erfolgen.

9.7.2 Socket-Adressen

Ein Socket muss noch an eine bestimmte Adresse *gebunden* werden (s.u.). Eine solche Adresse besteht aus der Definition des Rechners (z.B. Internet-Adresse) und eines „Ports“. Ein Port ist einfach eine ganze Zahl, die den Adressaten innerhalb des Rechners identifiziert. Die Zahlen 0 bis 1023 sind für Systemdienste reserviert (z.B. Port 80 für den HTTP-Dämon, 25 für Mail-Dienste, etc.).

Ein Server-Prozess (z.B. Webserver-Programm) definiert seinen eigenen Port und „hört“ ihn dann ab, d.h. er wartet auf Daten, die unter Angabe dieser Nummer an seinen Rechner geschickt werden. Ein Client-Prozess (z.B. Webbrowser) muss die Port-Nummer des Servers kennen, den er ansprechen will.

Zur Darstellung von Adressen für Sockets dient die Struktur `sockaddr`, bzw. speziellere Strukturen wie `sockaddr_in` für Internet-Sockets, etc.:

```
struct sockaddr_in
 {
   short            sin_family;     // Domain
   unsigned short   sin_port;       // Port-Nummer
   struct in_addr   sin_addr;       // Internet-Adresse
   unsigned char    __pad[...];     // Auffüllen auf sockaddr-Größe
 };
```

9.7.3 Datenaustausch über Sockets

Viele Datei-Operationen mit File-Deskriptoren sind auch mit den Deskriptoren für Sockets möglich; beispielsweise schließt man ja mit `close` eine Verbindung und gibt den Socket frei.

Für das Abschicken von Daten ist die Funktion `sendto`, für das Empfangen `recvfrom` zuständig.

Wenn man mit *Stream-Sockets* arbeitet, besteht eine permanente Verbindung. In diesem Fall verwendet man `send` und `recv`, es funktionieren aber genausogut `write` und `read` (wodurch z.B. Standard-Ein-/Ausgabe auf Sockets umgelenkt werden können). `send` und `recv` bieten lediglich zusätzliche Steuermöglichkeiten (Protokolle umgehen, etc.):

UNIX
int send**(int** s**, const void ***msg**, int** len**, unsigned int** flags**);** schickt Daten der Länge `len` Bytes ab der Adresse `msg` über den Socket `s`
int recv**(int** s**, void ***buf**, int** len**, unsigned int** flags**);** empfängt max. `len` Bytes Daten über den Socket `s` und schreibt sie nach `buf`

Die Funktionen liefern die Anzahl der tatsächlich geschriebenen/empfangenen Bytes zurück (-1 bei Fehler).

Der `recv`-Aufruf blockiert normalerweise, wenn keine Daten vorliegen, es sei denn, man hat das mit einem `fcntl`-Aufruf anders festgelegt:

```
fcntl(s,F_SETFL,O_NONBLOCK);
```

Es ist aber dennoch nicht gewährleistet, dass ein `recv`-Aufruf soviele Bytes liefert wie gewünscht. Folgende Funktion ruft so lange `recv` auf, bis die geforderte Menge angekommen ist (funktioniert natürlich auch für normale Files):

```
int x_recv(int fd, void *data, int size)
 {
   int left=size,bytes;
   char *P=(char *)data;
   do
```

```
    {
      if ((bytes=read(fd,P,left))<0) return 0;
      P+=bytes; left-=bytes;
    }
  while (left>0);
  return size;
}
```

Wenn man bei Kommunikation mit mehreren anderen Rechnern auf *mehrere Sockets gleichzeitig* warten möchte, sollte man den Systemaufruf `select` verwenden. Er ist für File-Deskriptoren ausgelegt und funktioniert auch mit normalen Dateien (einzubinden ist `sys/time.h`):

UNIX
`int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);` wartet auf Status-Änderungen bei beliebig vielen File-Deskriptoren

Es werden drei Deskriptor-Mengen `readfds`, `writefds` und `exceptfds` angegeben. Die Funktion blockiert so lange, bis über einen der Deskriptoren Daten *gelesen* bzw. *geschrieben* werden können, bzw. bis eine Datei ihren Status ändert. `n` ist der numerisch größte File-Deskriptor +1.

Zurückgegeben wird die Anzahl der Deskriptoren, die das Ende von `select` ausgelöst haben. In `timeout` (siehe auch Seite 156) kann man eine maximale Wartezeit angeben. Bei `timeout=0` blockiert `select` unbegrenzt lange.

Um die Deskriptor-Mengen zu manipulieren, sollte man nur die Makros `FD_ZERO` (leeren), `FD_SET` (einen Deskriptor eintragen) und `FD_CLR` (austragen) verwenden. Nach einem `select` kann man mit dem Makro `FD_ISSET` fragen, ob ein bestimmter Selektor Auslöser war:

```
FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

Das ausführliche Beispiel am Ende des Kapitels verwendet diesen Mechanismus.

9.7.4 Server- und Client-Sockets

Es gibt zwei (bzw. drei) Arten, einen Socket einzusetzen:

Client-Sockets:

Hier dient ein Socket hauptsächlich dazu, einen speziellen Host anzusprechen (einen Server), ihm Kommandos zu schicken und Daten als Antwort von ihm entgegenzunehmen. Auf dem entfernten Rechner muss bereits ein Socket existieren, mit dem der neue verbunden wird.

Folgende Schritte sind erforderlich, um so eine Verbindung herzustellen:

- ein Socket für diese Domain und diesen Typ wird erzeugt
- die (numerische) Adresse des entfernten Rechners muss ermittelt werden (beispielsweise durch `gethostbyname`);

- eine Adressstruktur wie `sockaddr_in` wird mit den Informationen über den Rechner gefüllt;
- mit dem Systemaufruf `connect` wird die Verbindung hergestellt.

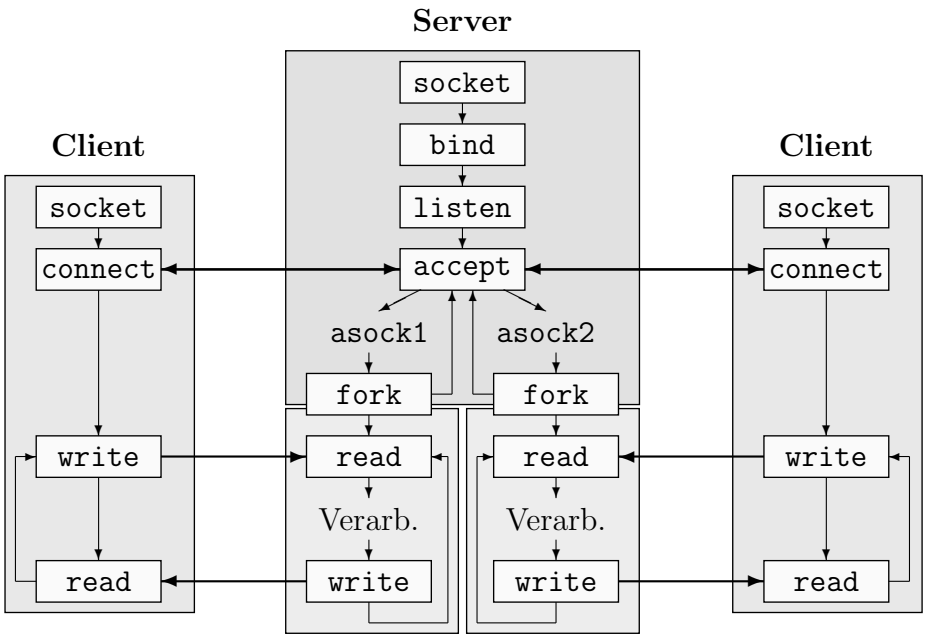
Server-Sockets:

Ein „**Listen-Socket**“ ist eine Anlaufstelle für Client-Anfragen. Er wird wie folgt erzeugt:

- ein Socket für die gewünschte Domain-/Typ-Kombination wird erzeugt;
- in einer Adressstruktur wird ein spezieller Client eingetragen – oder die Angabe, dass auf beliebige Anfragen gehorcht werden soll;
- mit dem Systemaufruf `bind` wird der Socket an die Adressdefinition gebunden;
- bei Stream-Sockets wird mit dem Systemaufruf `listen` die Maximalgröße der internen Warteschlange für Anfragen angegeben;
- mit dem Systemaufruf `accept` wird eine Anfrage eines Client-Sockets akzeptiert und ein „**Accept-Socket**“ erzeugt.

Über den Accept-Socket kann nun die Kommunikation mit dem Client erfolgen, während über den Listen-Socket weiterhin Anfragen anderer Clients entgegengenommen werden können. Wenn die Bearbeitung des Client-Auftrags länger dauert, kann man sie am besten mit `fork` abspalten und direkt wieder mit `accept` auf weitere Clients horchen.

Das folgende Bild fasst die einzelnen Abläufe (für Stream-Sockets mit ständiger Verbindung) noch einmal zusammen:



Im Folgenden sind die benötigten Funktionen aufgeführt:

UNIX
struct hostent *gethostbyname**(const char** ***name);** liefert eine (statisch angelegte) Struktur `hostent` mit Informationen über den Host mit dem Namen `name` (alphanumerisch oder numerisch mit 4 oder 6 Feldern); der Eintrag `h_addr` der Struktur liefert die numerische Adresse im internen Format
int connect**(int** fd**, struct** sockaddr ***serv_addr, int** addrlen**);** öffnet eine Verbindung vom Socket `fd` zu einem passenden Socket auf dem Server mit der Adresse `serv_addr`; es können unterschiedliche Strukturen bei `serv_addr` angegeben werden (beispielsweise `sockaddr_in` für Internet, daher muss die Größe der Struktur in `addrlen` übergeben werden
int bind**(int** fd**, struct** sockaddr ***my_addr, int** addrlen**);** bindet den Socket `fd` an die lokale Adresse `my_addr` (Länge `addrlen`)
int listen**(int** s**, int** backlog**);** definiert für den Server-Socket `s` die Länge der Warteschlange für eingehende Verbindungen
int accept**(int** s**, struct** sockaddr ***addr, int ***addrlen**);** akzeptiert eine Verbindung über den Server-Socket `s`, erzeugt einen passenden Accept-Socket und gibt dessen Deskriptor zurück; `*addrlen` muss die Länge der Adressstruktur enthalten; füllt außerdem `addr` mit Informationen über den anfragenden Client (und `addrlen` mit der echten Adresslänge)

Wenn nicht anders angegeben, liefern die Funktionen 0 bei okay zurück, -1 bei Fehlern.

Beispiel: Als Client-/Server-Beispiel implementieren wir wieder die lange Multiplikation. Wir verwenden der Einfachheit halber Stream-Sockets.

Der Client `sclient` erwartet als Aufrufparameter die Adresse des Server-Rechners, die Port-Nummer und die beiden Faktoren (um den Quelltext abzukürzen, werden einige Sicherheitsabfragen ausgelassen). Ein Aufruf wäre also: „`sclient vulcan 4711 1000000 12345678`“.

```
// include netinet/in netdb stdio stdlib string iostream unistd fcntl

void sorry(const char *s) { perror(s); exit(1); }

int main(int argc, char *argv[])
 {
   char buffer[1024];
   int len;

   if (argc!=5) { cerr << "Usage: sclient host port num1 num2"; exit(1); }

   struct hostent *host=gethostbyname(argv[1]);
   if (host==0) sorry("host");
```

```
  int host_sock=socket(AF_INET,SOCK_STREAM,0);
  if (host_sock<0) sorry("socket");

  static struct sockaddr_in host_in;
  host_in.sin_family=AF_INET;
  host_in.sin_addr.s_addr=((struct in_addr*)(host->h_addr))->s_addr;
  host_in.sin_port=htons(atoi(argv[2]));

  if (connect(host_sock,(struct sockaddr*)&host_in,sizeof(host_in))<0)
     sorry("connect");

  sprintf(buffer,"%s %s",argv[3],argv[4]);
  len=strlen(buffer)+1;
  if (send(host_sock,buffer,len,0)!=len) sorry("send");

  if (recv(host_sock,buffer,1024,0)>0)
     cout << argv[3] << '*' << argv[4] << '=' << buffer << endl;
  else sorry("recv");

  close(host_sock);
 }
```

Die Bibliotheksfunktion `htons` (Host-To-Network/Short) dient hier dazu, die Byte-Reihenfolge der Port-Nummer vom Rechner-Format ins Netzwerk-Format umzuwandeln (Intel-Prozessoren speichern längere Integer-Zahlen in umgekehrter Reihenfolge ab als der Rest der Welt). Auf Rechner, auf denen die Reihenfolgen übereinstimmen, tut die Funktion also nichts.

Das Server-Gegenstück `sserver` erhält als Parameter nur die Port-Nummer, die er verwenden soll. Er empfängt von beliebigen Clients *einmal* zwei Zahlen als Nachricht und schickt das Produkt zurück. Damit er sofort für andere Clients empfangsbereit ist, spaltet er die Berechnung und Beantwortung in einen Kind-Prozess ab:

```
// include sys/socket unistd stdio stdlib netinet/in ctype iostream string

void sorry(const char *s) { perror(s); exit(1); }

ostream & operator << (ostream &o, const struct in_addr &adr)
 {
  unsigned char c[4];
  memcpy(c,&adr,4);
  return o << (int)c[0]<<'.'<<(int)c[1]<<'.'<<(int)c[2]<<'.'<<(int)c[3];
 }

int main(int argc, char *argv[])
 {
  if (argc!=2) { cerr << "Usage: sserver port\n"; exit(1); }

  int lsocket=socket(AF_INET,SOCK_STREAM,0);
  if (lsocket<0) { perror("socket"); exit(1); }

  static struct sockaddr_in s_in;
```

```
  s_in.sin_family=AF_INET;
  s_in.sin_addr.s_addr=INADDR_ANY;
  s_in.sin_port=htons(atoi(argv[1]));

  if (bind(lsocket,(struct sockaddr*)&s_in,sizeof(s_in))<0) sorry("bind");
  if (listen(lsocket,10)<0) sorry("listen");

  for (;;)
   {
     static struct sockaddr_in incoming;
     int addrlen=sizeof(incoming);
     int asocket=accept(lsocket,(struct sockaddr*)&incoming,&addrlen);
     if (asocket<0) sorry("accept");

     if (fork()==0)
      {
        cout << "request from " << incoming.sin_addr << endl;

        char buffer[1024];
        int bytes;
        bytes=recv(asocket,buffer,1024,0);
        if (bytes>=3)
         {
           char *s1=strtok(buffer," "), *s2=strtok(0," ");
           char *s3=long_mul(s1,s2);
           int len=strlen(s3)+1;
           if (send(asocket,s3,len,0)!=len) sorry("send");
           free(s3);
         }
        close(asocket);
        _exit(0);
      }
     else close(asocket);
   }
}
```

9.7.5 Ausführliches Beispiel

Als komplexere Anwendung schreiben wir eine Client-/Server-Kombination zur verteilten Berechnung von Ausschnitten der **Mandelbrotmenge**.

Die Mandelbrotmenge ist die Menge Zahlen $c \in \mathbb{C}$, für die die Folge $m_c \in \mathbb{C}^{\mathbb{N}}$ mit $m_{c,0} := c$ und $m_{c,i+1} = m_{c,i}^2 + c$ $(\forall i \geq 0)$ nicht divergiert. Als Divergenzkriterium verwenden wir, dass $|m_{c,i}| \geq 2$ wird.

Wir stellen einen rechteckigen Ausschnitt der komplexen Ebene dar. Für jeden Pixel berechnen wir die ersten Glieder der zugehörigen Folge. Wenn wir Divergenz feststellen, färben wir den Pixel entsprechend der Nummer des erreichten Folgenglieds unterschiedlich ein. Wenn wir nach einer Maximalzahl von Folgengliedern noch keine Divergenz festgestellt haben, färben wir den Pixel schwarz.

Für die Darstellung in einem Fenster verwenden wir eine einfache `C++`-Klasse `SimpleWindow`, deren Interna hier nicht relevant sind. Wir benötigen hauptsächlich Konstruktor/Destruktor und eine Funktion zum Zeichnen eines Pixels.

- In unserer verteilten Implementation haben wir einen Client, den wir hier „**Master**“ nennen wollen, und beliebig viele Server für die eigentlichen Berechnungen (hier „**Servants**“ genannt).
- Es gibt hier eine einfache Möglichkeit, den Datenverkehr zu kontrollieren und die Lasten der Rechenleistung der Servants entsprechend zu verteilen. Die Berechnungen werden *zeilenweise* verteilt. Wenn ein Servant eine Zeile berechnet hat, schickt er sie an den Master, der ihm die nächste Zeile zuteilt. Zu Beginn werden die ersten n Zeilen direkt an die n Servants verschickt.
- Wir haben zwei Arten von Nachrichten: Der Master schickt einem Servant seine Aufgabe (hauptsächlich die Nummer der zu berechnenden Zeile), und der Servant schickt die fertig berechnete Zeile zurück an den Master. Um den Datenverkehr klein zu halten, schränken wir die Anzahl von Farben auf maximal 256 ein. Die Servants schicken dann nicht ein Array mit den erreichten *Iterationstiefen* in der Zeile, sondern direkt mit den *Farbcodes* zurück (ein Byte pro Pixel).
- Wir brauchen zwei Ports (`MASTERPORT` und `SERVANTPORT`), da wir es erlauben wollen, dass ein Servant und ein Master auf demselben Rechner existieren. Der Master startet der Einfachheit halber nicht mit `fork` einen eigenen Berechnungsprozess, mit dem er ja schneller über Shared Memory o.ä. kommunizieren könnte.
- Um die Servants anzusprechen, kennt der Master ihre *Namen*. Er muss ihre *Adressen* zunächst selbst ermitteln. Die Servants dagegen kennen ihren Master nicht und erfahren seine Adresse erst beim Zustandekommen der Verbindung über ihren Server-Socket.
- Der Master erzeugt für jeden Servant einen Client-Socket `send_sock[i]`, über den er die Aufgaben abschickt, insgesamt also n Stück. Der verwendete Port soll `SERVANTPORT` heißen. Für den Empfang der Daten richtet er einen Listen-Socket mit dem Port `MASTERPORT` ein. Bei jeder Rückmeldung bei `accept` erzeugt er einen Accept-Socket `recv_sock[i]`, wiederum n Stück.
- Jeder Servant richtet einen Server-Socket `in_sock` ein, mit dem er den Port `SERVANTPORT` abhorcht, um den Master zu finden. Über `accept` erhält er einen Accept-Socket, über den er vom Master die Aufgaben bekommt (`recv_sock`). Außerdem legt er einen Client-Socket `send_sock` an (mit der Adresse aus dem Accept-Socket), über den er die Aufgaben zurückschickt.
- Als erstes erhalten die Servants eine „Hello“-Nachricht, die die Realteile, Breite, Iterationstiefe und Farbanzahl enthält, und die sie zur Bestätigung zurückschicken. Diese Daten werden auch bei jeder weiteren Übertragung mitgeschickt, in dieser Version aber ignoriert. Vor allem erfahren sie hier ihre Nummer, d.h. ihren Index in den Master-Arrays. Sie wird bei jeder Rückmeldung zur Identifikation zurückgeschickt.

Die Port-Nummern und die Strukturen für die beiden Nachrichtentypen sind in der gemeinsamen **Header-Datei** `multimandel.h` definiert:

```
#define MASTERPORT 6666
#define SERVANTPORT  6667

struct cmd_msg                  // vom Master zum Servant
 {
   unsigned short num;          // Nummer der Servants
   char command,dummy;          // Kommando: q(uit), s(tart), c(ompute)
   unsigned short line;         // Nummer der Zeile
   double y,x1,x2;              // Imaginärteil der Zeile, Realteil-Begrenzungen
   unsigned short width;        // Bildbreite
   unsigned short depth;        // maximale Iterationstiefe
 };

struct data_msg                 // vom Servant zum Master
 {
   unsigned short num;          // Nummer des Servants
   unsigned short line;         // Nummer der Zeile
   unsigned char it[0];         // Farbdaten, eigentlich variabel it[width]
 };
```

Die zweite Struktur ist „variabel groß“ (abhängig von der Bildbreite). Ihre tatsächliche Größe in Bytes wird per `sizeof` im Programm berechnet.

Es folgt der Code des **Servants** (`servant.cpp`. Die Funktionen `sorry`, `x_recv` und der Ausgabe-Operator `<<` für Adressen sind aus den vorhergehenden Programmen übernommen.

```
// include sys/socket unistd stdio stdlib netinet/in ctype iostream iomanip
// include string signal multimandel

int in_sock,recv_sock,send_sock;
struct cmd_msg msgbuf;
struct data_msg *data;
size_t data_size;

double x_1,x_2,dx,*xval,*xsq;
int width,numcolors,maxit,*modtab;

void sigpipe_handler(int) { raise(SIGINT); }       // "Broken Pipe" ⇒ Abbruch

int one_point(double x0, double y0, double x2, double y2)
 {                                                 // einen Punkt berechnen
   double x=x0,y=y0;
   int it=0,cmp=msgbuf.depth;
   if (x2+y2>=4.0) return 0;
   for (;;)
    {
      if (++it==cmp) return it;
      y+=(y*=x)+y0;
      x=x2-y2+x0;
      if ((x2=x*x)+(y2=y*y)>=4.0) return it;
```

```
    }
 }
void one_line()                                    // eine Zeile berechnen
 {
   unsigned char *P=data->it;
   double y2=msgbuf.y*msgbuf.y,*xP=xval,*x2P=xsq;

   for (int i=0;i<width;++i)
      *P++=modtab[one_point(*xP++,msgbuf.y,*x2P++,y2)];
 }
void init_mandel()                                 // Initialisierungen
 {
   width=msgbuf.width; numcolors=msgbuf.line; maxit=msgbuf.depth;
   x_1=msgbuf.x1; x_2=msgbuf.x2; dx=(x_2-x_1)/width;
   xval=new double[width]; xsq=new double[width];
   for (int i=0;i<width;++i) { xval[i]=x_1+i*dx; xsq[i]=xval[i]*xval[i]; }

   modtab=new int[maxit+1];
    for (int i=0,j=1;i<maxit;++i)
     {
       modtab[i]=j;
       if (++j>=numcolors) j=1;
     }
 }
int main()
 {
   signal(SIGPIPE,sigpipe_handler);

   in_sock=socket(AF_INET,SOCK_STREAM,0);
   if (in_sock<0) sorry("socket");

   static struct sockaddr_in s_in;
   s_in.sin_family=AF_INET;
   s_in.sin_addr.s_addr=INADDR_ANY;
   s_in.sin_port=htons(SERVANTPORT);

   if (bind(in_sock,(struct sockaddr*)&s_in,sizeof(s_in))<0) sorry("bind");
   if (listen(in_sock,10)<0) sorry("listen");

   struct sockaddr_in incoming;
   int addrlen=sizeof(incoming);
   recv_sock=accept(in_sock,(struct sockaddr*)&incoming,&addrlen);
   if (recv_sock<0) sorry("accept");
   cout << "accepted from " << incoming.sin_addr << endl;

   send_sock=socket(AF_INET,SOCK_STREAM,0);
   if (send_sock<0) sorry("socket");
   incoming.sin_port=htons(MASTERPORT);

   if (connect(send_sock,(struct sockaddr *)&incoming,sizeof(incoming))<0)
      sorry("connect");
```

```
  for (;;)
   {
     if (x_recv(recv_sock,&msgbuf,sizeof(msgbuf))==0) sorry("recv");

     switch (msgbuf.command)
      {
        case 'q':           // quit: Stop-Signal vom Master erhalten, STOP
           exit(0);

        case 's':           // start: Initialisierung, "hello" zurückschicken
           init_mandel();
           data_size=sizeof(data_msg)+sizeof(unsigned char[width]);
           data=(data_msg*)new char[data_size];
           data->num=msgbuf.num;
           if (send(send_sock,data,data_size,0)<0) sorry("send");
           break;

        case 'c':           // compute: eine Zeile berechnen & zurückschicken
           one_line();
           data->line=msgbuf.line;
           if (send(send_sock,data,data_size,0)<0) sorry("send");
           break;

        default:
           cerr << "unknown command " << msgbuf.command << endl;
      }
   }
}
```

Es folgt der Code des **Masters**, zu dem zunächst einige Bemerkungen vorangestellt werden.

- Wir befassen uns in diesem Kapitel nur mit der eigentlichen Kommunikation zwischen Prozessen, nicht damit, wie wir bei den *Programmcode* auf entfernte Rechner *transportieren* und dort *starten*! Wir gehen hier davon aus, dass der Code (z.B. per `ftp`) auf allen Rechnern vorliegt. Wenn die Rechner ohnehin ein gemeinsames Dateisystem haben, ist das natürlich überflüssig.

- Die Servants müssen alle laufen, bevor der Master beginnt, die Verbindungen aufzubauen. Wir haben zwei Möglichkeiten, die Servants zu starten:
  - per Hand (mit `rlogin`, Starten aus der Shell heraus)
  - automatisch vom Master aus

  Zweiteres erzwingen wir mit der Option '`+`' an unser Master-Programm. Es erzeugt dann für jeden Servant einen Kind-Prozess, der sich mit dem `rsh`-Kommando (Remote-Shell) überlädt. Diese Shell startet auf dem entfernten Rechner das Kommando, das ihr als Parameter übergeben wird. Hier geben wir den absoluten Pfad des Servant-Codes an (in unserer Version muss er auf allen Rechnern an derselben Stelle liegen).

  Achtung: Damit die Remote-Shell nicht nach unserem Passwort fragt, müssen wir auf den entfernten Rechnern in der Datei `.rhosts` den Master-Rechner eintragen.

- Vorsicht: Die Internet-Verbindungen haben eine kurze „Nachlaufzeit“, d.h. die Ports sind noch kurze Zeit nach dem Auflösen der Verbindung reserviert. Die Funktion `bind` in Master und Servant meldet dann einen Fehler. In diesem Fall müssen wir die Remote-Shells durch ein `kill`-Kommando selbst beseitigen (aus Platzgründen hier nicht ins Programm aufgenommen)!
- Am Schluss gibt der Master aus, welcher Servant wieviele Zeilen berechnet hat, und wie lange der gesamte Vorgang (inklusive Herstellen der Verbindungen) bzw. nur die Berechnung gedauert hat.

```
// include netinet/in netdb stdio stdlib string iostream sys/types sys/time
// include sys/timeb unistd fcntl math signal multimandel swindow

// ------------------------------------------------------------------------

/**  die beteiligten Rechner (Servants): **    */
char *servant[]={ "vulcan.solar.system", "qonos.solar.system" };

/**  Bildbreite und Hohe: **  */
static int width=800,height=600;

/**  Berechnungs- und Darstellungs-Parameter **    */
static double x_1=-2.0,x_2=0.5,y_1=-0.9375,y_2=0.9375;
static const int maxit=1500, numcolors=32;

// ------------------------------------------------------------------------

const int num_servants=(sizeof(servant)/sizeof(servant[0]));
int send_sock[num_servants],master_sock,recv_sock[num_servants];
bool alive[num_servants];
int num_alive;
int compline[num_servants],countlines[num_servants];
pid_t rsh[num_servants];

struct sockaddr_in s_in;
struct cmd_msg msgbuf;
struct data_msg *data;
size_t data_size;

double dx;
SimpleWindow *pwin;
static int (*colors)[3];

void sorry(const char *s) { extern void fini(int); perror(s); fini(0); }

void send_msgbuf(int i)
 { if (send(send_sock[i],&msgbuf,sizeof(msgbuf),0)<0) sorry("send"); }

void fini(int)
 {
   delete pwin;
   msgbuf.command='q';
   for (int i=0;i<num_servants;++i)
    {
```

```
      if (send_sock[i]>0) send_msgbuf(i);
      kill(rsh[i],SIGTERM);
    }
   signal(SIGINT,SIG_DFL); raise(SIGINT);
 }

// Rückruf-Funktionen der Fensterklasse
static void expose_callback(const int &w, const int &h)
 { pwin->copy_pixmap(); }
static void keypress_callback(const int &key) { if (key=='q') fini(0); }

// Darstellung
void make_colors() { ... }        // Farb-Berechnung, hier ausgelassen!

void show_one_line(int ypix, unsigned char *line)
 {
   for (int i=0;i<width;++i)
    {
      pwin->set_color(*line++);
      pwin->draw_point_pm(i,ypix);
    }
   pwin->copy_pixmap_line(ypix);
 }

// Socket-Definitionen

int make_server_socket(int port)
 {
   int sock=socket(AF_INET,SOCK_STREAM,0);
   if (sock<0) sorry("socket1");

   memset(&s_in,0,sizeof(s_in));
   s_in.sin_family=AF_INET;
   s_in.sin_addr.s_addr=INADDR_ANY;
   s_in.sin_port=htons(MASTERPORT);
   if (bind(sock,(struct sockaddr*)&s_in,sizeof(s_in))<0) sorry("bind");
   if (listen(sock,10)<0) sorry("listen");

   return sock;
 }

int make_client_socket(const char *name, int port)
 {
   struct hostent *host=gethostbyname(name);
   if (host==0) sorry("host");

   for (int i=5;;)
    {
      int sock=socket(AF_INET,SOCK_STREAM,0);
      if (sock<0) sorry("socket");
      memset(&s_in,0,sizeof(s_in));
      s_in.sin_family=AF_INET;
      s_in.sin_addr.s_addr=((struct in_addr*)(host->h_addr))->s_addr;
```

```
      s_in.sin_port=htons(port);
      if (connect(sock,(struct sockaddr*)&s_in,sizeof(s_in))>=0) return sock;
      if (--i==0) sorry(name);
      close(sock);
      sleep(1);
    }
 }

// Verschicken und Empfangen

void send_task(int to, int line)
 {
   if (to<0||to>=num_servants) { cerr << "illegal send" << endl; exit(1); }

   msgbuf.num=to;
   msgbuf.command='c';
   msgbuf.line=line;
   msgbuf.y=y_1+line*dx;
   send_msgbuf(to);
   compline[to]=line;
 }

void init_mandel()                                    // Initialisierung
 {
   if (x_2<x_1) swap(x_1,x_2);
   if (y_2<y_1) swap(y_1,y_2);
   dx=(x_2-x_1)/(double)width;
   double f=0.5*dx*height,g=0.5*(y_1+y_2);
   y_1=g-f; y_2=g+f;
 }

int main(int argc, char *argv[])                      // Master-Hauptprogramm
 {
   int i;

   struct timeb t1,t2,t3;
   ftime(&t1);
   signal(SIGINT,fini);
   init_mandel();

   if (argc>=2 && argv[1][0]=='+')
      for (i=0;i<num_servants;++i)
       {
         if ((rsh[i]=fork())==0)
          {
            char buffer[80];
            execlp("rsh","rsh","-n",servant[i],"/home/axel/servant",0);
          }
       }

   msgbuf.x1=x_1; msgbuf.x2=x_2;
   msgbuf.width=width; msgbuf.depth=maxit;
   msgbuf.line=numcolors;
```

```
data_size=sizeof(data_msg)+sizeof(unsigned char[width]);
data=(data_msg*)new char[data_size];

master_sock=make_server_socket(MASTERPORT);

for (i=0;i<num_servants;++i)
 {
   send_sock[i]=make_client_socket(servant[i],SERVANTPORT);
   msgbuf.num=i;
   msgbuf.command='s';
   send_msgbuf(i);
 }

num_alive=0;
for (i=0;i<num_servants;++i)
 {
   int addrlen;
   int sock=accept(master_sock,(struct sockaddr*)&s_in,&addrlen);
   if (sock<0) sorry("accept");
   cout << "connected to " << s_in.sin_addr << endl;

   if (x_recv(sock,data,data_size)==0) sorry("recv");

   recv_sock[data->num]=sock;
   if (alive[data->num]) exit(99);
   alive[data->num]=true;
   ++num_alive;

   send_task(data->num,i);
 }

if (num_alive==0) { cerr << "nobody wants to talk to me" << endl; exit(1); }

make_colors();
pwin=new SimpleWindow(width,height,"MandelMaster",numcolors,colors);
pwin->set_expose_func(expose_callback);
pwin->set_keypress_func(keypress_callback);

int lines_to_receive=height, next_line=num_servants-1;

ftime(&t2);

do
 {
   fd_set wait_set;
   FD_ZERO(&wait_set);
   int max=0;
   for (i=0;i<num_servants;++i)
      if (alive[i])
       {
         FD_SET(recv_sock[i],&wait_set);
         if (recv_sock[i]>max) max=recv_sock[i];
       }

   struct timeval tv_wait;
```

```
   tv_wait.tv_sec=20; tv_wait.tv_usec=0;
   if (select(max+1,&wait_set,0,0,&tv_wait)==0)
      cerr << "warning: no message in 20 seconds" << endl;

   for (i=0;i<num_servants;++i)
      if (FD_ISSET(recv_sock[i],&wait_set))
       {
         int bytes=x_recv(recv_sock[i],data,data_size);
         if (bytes<=0)
          {
            cerr << i << " died" << endl;
            alive[i]=false;
            if (--num_alive==0) { cerr << "all dead" << endl; exit(1); }
          }
         else
          {
            if (data->num>=num_servants)
               cerr << "something's obviously wrong..." << endl;
            else
             {
               if ( bytes!=data_size || data->line!=compline[data->num] )
                {
                  cerr << "received some garbage..." << endl;
                  send_task(data->num,compline[data->num]);
                }
               else
                {
                  ++countlines[data->num];
                  if (--lines_to_receive>0 && ++next_line<height)
                     send_task(data->num,next_line);
                  show_one_line(data->line,data->it);
                  if (lines_to_receive==0) break;
                }
             }
          }
       }

   pwin->handle_events(false);
 }
while (lines_to_receive>0);

ftime(&t3);
cout << "\nsuccessfully completed!\n";

for (i=0;i<num_servants;++i)
   cout << "received " << countlines[i] << " lines from "
        << servant[i] << endl;

cout << "time: " << (t3.time-t1.time)+(t3.millitm-t1.millitm)/1000.0
     << " s ("
     << (t3.time-t2.time)+(t3.millitm-t2.millitm)/1000.0
     << " s computing time)" << endl;
```

```
    pwin->handle_events(true);
    fini(0);
  }
```

Bei acht Test-Rechnern ergab sich beispielsweise folgende Ausgabe:

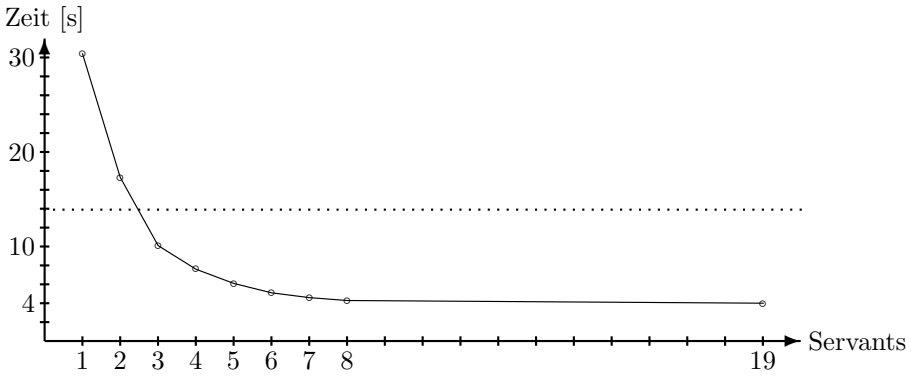
```
received 87 lines from wmpi01.math.uni-wuppertal.de
received 58 lines from wmpi02.math.uni-wuppertal.de
received 82 lines from wmpi03.math.uni-wuppertal.de
received 74 lines from wmpi04.math.uni-wuppertal.de
received 69 lines from wmpi05.math.uni-wuppertal.de
received 77 lines from wmpi06.math.uni-wuppertal.de
received 75 lines from wmpi07.math.uni-wuppertal.de
received 78 lines from wmpi08.math.uni-wuppertal.de
time: 5.964 s (4.343 s computing time)
```

Die Rechner sind in etwa gleich schnell, waren aber unterschiedlich stark anderweitig beschäftigt. Auf `wmpi01` lief auch der Master, sodass es nicht verwunderlich ist, dass von dort die meisten Zeilen stammen.

Ein gleichwertiges Programm auf einem Rechner alleine ohne jegliches Verschicken benötigt 13.9 Sekunden, mit Verschicken an sich selbst dagegen 30.4 Sekunden! Bei den vergleichsweise kurzen Berechnungszeiten ist der Verwaltungsaufwand des Datentransports dominant. Erst bei drei Rechnern sind wir wieder schneller als die 13.9 Sekunden.

Servant	1	2	3	4	5	6	7	8	...	19
Zeit [s]	30.4	17.3	10.1	7.6	6.1	5.1	4.6	4.3	...	4.0

Bei mehr als 8 Rechnern ist kein wesentlicher Geschwindigkeitsvorteil mehr festzustellen. Bis zu den getesteten 19 Rechnern stieg die Berechnungszeit aber auch nicht wieder an.



10 Deadlocks

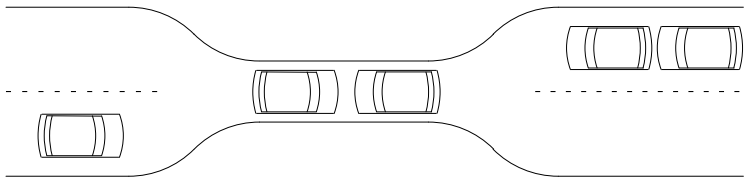
Ein Deadlock (eine *Verklemmung*) ist eine Situation, in der jeder Prozess einer Gruppe von Prozessen auf ein Ereignis wartet, das nur von einem anderen Prozess der Gruppe ausgelöst werden kann.

Prozesse geraten meist im Zusammenhang mit gemeinsamen Ressourcen in Deadlocks, d.h. wenn sie auf eine solche Weise auf die Ressourcen warten, dass keiner von ihnen jemals mehr den Wartezustand verlassen kann. Die schon gebundenen Ressourcen werden nie mehr freigegeben.

Andere Deadlocks entstehen durch Fehler bei der Synchronisation (z.B. mit zählenden Semaphoren). Sie sind allerdings immer auf falsche Verwendung der Synchronisationsmittel zurückzuführen. Wir beschäftigen uns hier hauptsächlich mit den Deadlocks, die mit Ressourcen zu tun haben.

Wir hatten bereits in 8.3.3 (vertauschte Semaphore) und 8.3.5 (Philosophenproblem) gesehen, wie leicht man eine solche Situation heraufbeschwören kann.

Beispiel 1: Deadlocks können oft mit Situationen im Straßenverkehr verglichen werden. Beispielsweise betrachten wir eine Brücke, auf der nur eine Fahrspur Platz findet:



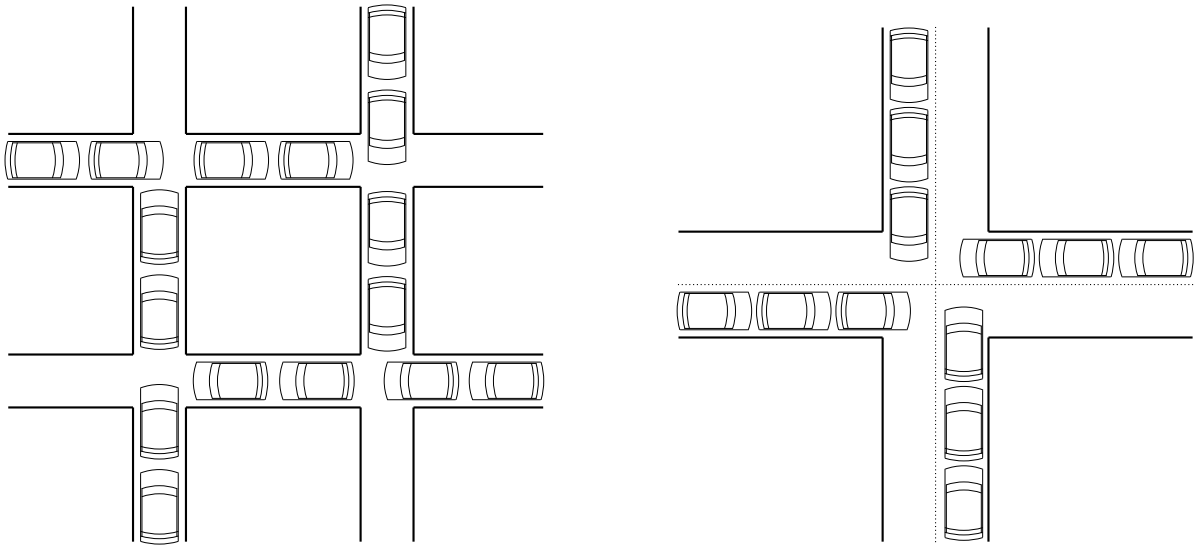
Die Ressourcen sind hier die einzelnen Teile der Straßenkonstruktion, beispielsweise Brückensektionen der Länge in etwa einer Wagenlänge. Die Wagen stellen die Prozesse dar.

Um die Brücke komplett von links zu überqueren, muss ein Wagen zunächst auf die erste Sektion von links fahren und dann sukzessive eine Sektion weiter. (In unserem Bild gibt es nur eine linke und eine rechte Sektion.) Er kann dabei einen Wagen blockieren, der dasselbe von rechts versucht. Der Wagen hat keine Möglichkeit, eine Sektion freizugeben, bevor er die nächste Sektion erreicht hat.

- Wenn ein Deadlock aufgetreten ist, kann er nachträglich **entfernt** werden, wenn die Prozesse gezwungen werden, Ressourcen wieder abzugeben. Hier würde das bedeuten, dass ein oder mehrere Wagen rückwärts von der Brücke fahren müssen.
- Der Deadlock kann schon vorab **verhindert** werden, wenn es einen Straßenwächter gibt, der die Wagen genügend weit vor der Brücke anhält, warten lässt und immer nur einen auf die Brücke lässt.
- Prozesse können dabei allerdings **verhungern** – nämlich wenn der Straßenwächter beispielsweise von der linken Seite kommende Wagen bevorzugt und die auf der rechten Seite unbeachtet lässt.

Beispiel 2: Oben blockieren sich direkt zwei Wagen gegenseitig. Unübersichtlicher kann die Situation werden, wenn die Blockade über den Umweg anderer Prozesse zustande kommt. Es gibt dann eine Kreisblockade wie beim folgenden Verkehrsstau.

Nachträglich kann der Deadlock nur dadurch aufgelöst werden, dass die Wagen an einer Stelle vor der Kreuzung zurücksetzen. Verhindern kann man ihn durch Verkehrsampeln, die natürlich so geschaltet sein müssen, dass die Wagen aus keiner Straße „verhungern“.



Es gibt vier Möglichkeiten, mit Deadlocks umzugehen:

Deadlock-Vermeidung (*Deadlock Avoidance*): Die Freiräume der Prozesse bei der Ressourcen-Anforderung werden eingeschränkt, so dass es niemals zu einem Deadlock kommen kann.

Deadlock-Verhinderung (*Deadlock Prevention*): Vor jeder Ressourcen-Zuteilung wird geprüft, ob sie zu einem Deadlock führen *würde* (dann wird sie aufgeschoben, und der Prozess wird blockiert).

Deadlock-Erkennung (*Deadlock Detection*): Entstandene Deadlocks müssen erkannt und durch Ressourcen-Entzug o.ä. nachträglich aufgelöst werden.

Deadlock-Ignorieren (*Ostrich Algorithm*, Vogel-Strauß-Algorithmus): Zugrunde liegt die Annahme, dass Deadlocks *vergleichsweise* selten auftreten. Im Ernstfall muss der Administrator eingreifen und einige beteiligte Prozesse entfernen.

Es ist eigentlich eine Aufgabe des Betriebssystems, Deadlocks zu verhindern oder aufzulösen. Die wenigsten Betriebssysteme (auch UNIX nicht) stellen aber einen solchen Service zur Verfügung.

Das hängt vor allem damit zusammen, dass Prozesse in ihrer Freiheit eingeschränkt werden müssten oder die Überprüfung mit einigem Zeitaufwand verbunden wäre. Entsprechende Mechanismen müssen dann bei Bedarf daher in die Prozesse selbst eingebaut werden.

10.1 Zustandsdiagramme

Bei der Betrachtung des Zusammenspiels mehrerer Prozesse ist es oft nützlich, das System ähnlich wie einen (nicht-deterministischen) endlichen Automaten darzustellen.

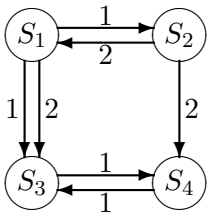
Bezeichnungen: Wir nennen die Menge aller überhaupt denkbaren Systemzustände (Statūs) $\Sigma = \{S_1, S_2, \ldots\}$ und die Menge aller aktuellen Prozesse $\Pi = \{P_1, P_2, \ldots\}$.

Einen Prozess fassen wir als Abbildung $P : \Sigma \to \wp(\Sigma)$ auf. $P(S)$ gibt dabei die Menge der *möglichen* Systemzustände an, die für den Prozess vom Zustand $S \in \Sigma$ aus erreichbar sind. (Die Abbildung wird immer angewandt, wenn der Prozess aktiv werden kann.) Ein Wert $P(S) = \emptyset$ deutet also an, dass der Prozess beim Systemzustand S *nicht* weiterarbeiten kann.

Der *Zustandsgraph* (Σ, E_s) des Systems hat als Punktmenge gerade die Menge Σ der Systemzustände. In $E_s \subset \Sigma \times \Sigma \times \Pi$ gibt es für alle $S_j \in P_k(S_i)$ eine mit P_k (oder k) beschriftete gerichtete Kante von S_i nach S_j.

Beachte: Wenn man ein reales *komplettes Rechnersystem* so modellieren will, wird die Anzahl der Zustände extrem groß (sie wächst exponentiell mit der Anzahl von Ressourcen)! Man beschränkt sich meistens auf eine relevante kleine Teilmenge.

Beispiel:



Die Werte der Prozessfunktionen sind hier als Tabelle angegeben:

$P_1(S_1) = \{S_2, S_3\}$	$P_2(S_1) = \{S_3\}$
$P_1(S_2) = \emptyset$	$P_2(S_2) = \{S_1, S_4\}$
$P_1(S_3) = \{S_4\}$	$P_2(S_3) = \emptyset$
$P_1(S_4) = \{S_3\}$	$P_2(S_4) = \emptyset$

- Einen Zustandsübergang von S_i nach S_j durch P_k schreiben wir als $S_i \xrightarrow{k} S_j$.
- Eine Folge von $n-1$ „passenden“ Zustandsübergängen („Pfad“), verursacht durch ggf. unterschiedliche Prozesse, schreiben wir als $S_{i_1} \xrightarrow{k_1} S_{i_2} \xrightarrow{k_2} \ldots \xrightarrow{k_{n-2}} S_{i_{n-1}} \xrightarrow{k_{n-1}} S_{i_n}$.
- Wenn es eine solche Folge von Zustandsübergängen gibt, die bei S_i startet und in S_j endet, schreiben wir $S_i \xrightarrow{*} S_j$.

10.2 Charakterisierung von Deadlocks

Mit Hilfe der Zustandsdiagramme führen wir einige weitere Begriffe ein:

- Ein Prozess P_i heißt **blockiert** im Zustand S_j, wenn $P_i(S_j) = \emptyset$. (Da andere Prozesse den Systemzustand ändern können, muss der Prozess natürlich nicht für immer blockiert bleiben.)
- Man sagt, ein Prozess P_k befindet sich in einem **Deadlock-Zustand** S_i, wenn er in S_i blockiert ist, und in allen Zuständen S_j mit $S_i \xrightarrow{*} S_j$ immer noch blockiert ist. (Auch Änderungen durch andere Prozesse können die Blockade also nicht aufheben.)

- S_i heißt **Deadlock-Zustand des Systems**, wenn es ein P_k gibt, das sich in S_i in einem Deadlock-Zustand befindet.
- S_i heißt **totaler Deadlock-Zustand** des Systems, wenn es für alle Prozesse P_k ein Deadlock-Zustand ist.
- S_i heißt **sicherer Zustand**, wenn es kein Deadlock-Zustand ist und für alle Übergänge $S_i \xrightarrow{*} S_j$ auch S_j kein Deadlock-Zustand ist.

Im Beispiel oben ist S_3 ein Deadlock-Zustand für P_2, da $P_2(S_3) = \emptyset$ und der einzige Übergang $S_3 \xrightarrow{1} S_4$ wieder in einen Zustand mit $P_2(S_4) = \emptyset$ führt. Analog ist auch S_4 ein Deadlock-Zustand für P_2. Es gibt keine totalen Deadlock-Zustände.

Beispiel: Wir wollen nun den Standardfall der beiden Prozesse P_1 und P_2 untersuchen, die sich um dieselben beiden Ressourcen R_a und R_b streiten, die nur exklusiv gehalten werden können. Die Prozesse sollen die Ressourcen in zueinander umgekehrter Reihenfolge anfordern (das ist genau die Situation aus 8.3.3).

Wir bezeichnen die relevanten Systemzustände mit S_{ijkl}, wobei $i, k \in \{0, a, A, *\}$ und $j, l \in \{0, b, B, *\}$.

```
void P_1()
 {
   for (;;)
    {
      ...
      down(sem_a);
      down(sem_b);
      ...
      up(sem_b);
      up(sem_a);
    }
 }
```

```
void P_2()
 {
   for (;;)
    {
      ...
      down(sem_b);
      down(sem_a);
      ...
      up(sem_a);
      up(sem_b);
    }
 }
```

Die Indizes i, j stehen für P_1, die Indizes k, l für P_2. Die Buchstaben bedeuten folgendes:

0	–	der Prozess hält die Ressource nicht (und ist nicht interessiert),
a/b	–	der Prozess fordert die Ressource an,
A/B	–	der Prozess hält die Ressource,
$*$	–	der Prozess hat die Ressource gerade freigegeben.

Die verschiedenen Buchstaben wären nicht nötig, machen die Sache aber lesbarer.

Der normale Gang der Dinge für P_1 ist also (für P_2 analog):

$$S_{00\_\_} \to S_{a0\_\_} \to S_{A0\_\_} \to S_{Ab\_\_} \to S_{AB\_\_} \to S_{A*\_\_} \to S_{00\_\_}.$$

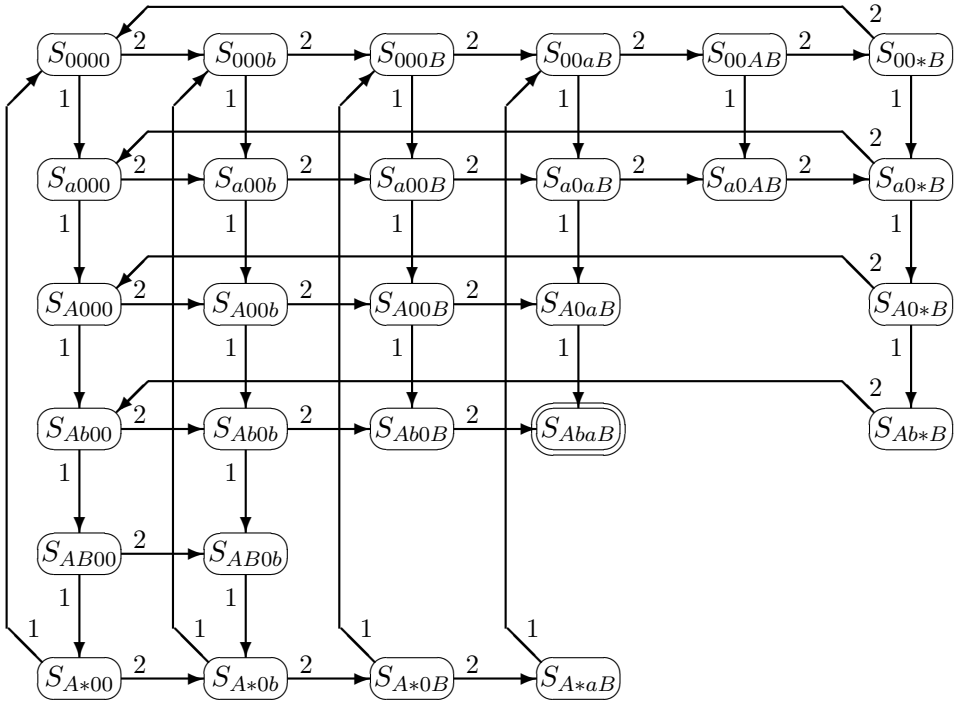
Das Sternchen ist notwendig. Wir müssen nicht nur den Belegungszustand der Ressourcen, sondern auch den internen Zustand der Prozesse mitcodieren. Das hier relevante Detail ist, ob der Prozess nur eine Ressource hält, weil er die andere noch nicht bekommen hat, oder weil er von zweien erst eine freigegeben hat:

Wenn P_1 nacheinander R_b und R_a freigibt, könnte man einen Übergang $S_{AB\_\_} \to S_{A0\_\_} \to S_{00\_\_}$ verwenden. Beim *Anfordern* der Ressourcen möchte P_1 aber vom Zustand $S_{A0\_\_}$ nur in den Zustand $S_{Ab\_\_}$ übergehen. Die Zustände der *Ressourcen* mögen also bei $S_{A*\_\_}$ und $S_{A0\_\_}$ gleich sein, nicht aber die Zustände der *Prozess-Wünsche* (also die Position im Code). Zustände $S_{**\_\_}$ bzw. $S_{\_\_**}$ können dagegen problemlos mit $S_{00\_\_}$ bzw. $S_{\_\_00}$ identifiziert werden.

Im Folgenden ist das Zustandsdiagramm dargestellt, das sich ergibt.

Der Zustand S_{AbaB} ist ein **totaler Deadlock-Zustand**. Seine beiden Vorgänger S_{Ab0B} und S_{A0aB} sind Deadlock-Zustände, von denen aus ein Deadlock schon unausweichlich ist. Es gibt keine sicheren Zustände, da die Deadlocks von allen Zuständen aus erreichbar sind.

Außerdem ist P_1 in S_{a0AB} und S_{Ab*B} blockiert, und P_2 in S_{AB0b} und S_{A*aB}.



Es gibt folgende notwendige Voraussetzungen für das Auftreten eines Deadlocks:

Exklusiver Zugriff: Mindestens eine der beteiligten Ressourcen wird per exklusivem Zugriff gehalten, d.h. nur maximal ein Prozess kann zur selben Zeit Zugriff haben.

Ressourcen, auf die mehrere Prozesse problemlos gleichzeitig zugreifen können, können bei einer Deadlock-Untersuchung außer acht gelassen werden.

Freiwillige Abgabe: Ressourcen müssen von Prozessen freiwillig wieder abgegeben werden, d.h. es gibt keinen Preemption-Mechanismus.

Die CPU kann einem Prozess per Zeitscheiben-Mechanismus entzogen werden, Hauptspeicher per Auslagerung. Dagegen kann man bei gestartetem Ausdruck einem Prozess nicht den Drucker entziehen oder eine (gemeinsam benutzte) Datei während eines Schreibvorgangs.

Zyklisches Warten: Es muss einen Zyklus $(P_0, P_1, \ldots, P_{n-1})$ von wartenden Prozessen geben, sodass jedes P_i auf eine von $P_{(i+1) \bmod n}$ gehaltene Ressource wartet.

Der Zyklus im letzten Fall kann natürlich aus nur zwei Prozessen bestehen. Dann haben zwei Prozesse bereits eine Ressource erhalten, die zweite benötigte wird vom jeweils anderen gehalten. Das ist die kleinste klassische Deadlock-Situation. Sie entspricht dem Zustand S_{AbaB} aus dem Diagramm von oben.

10.3 Ressourcengraphen

Die Abhängigkeiten zwischen Prozessen und Ressourcen kann man gut in Form von „Ressourcengraphen“ (*General Resource Graphs*) modellieren, die 1972 von Holt eingeführt wurden.

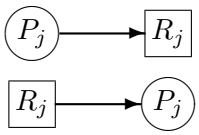
Der Ressourcengraph ist ein gerichteter Graph $G = (V, E)$ ('V' für *Vertices*=Punkte, 'E' für *Edges*=Kanten), der wie folgt definiert ist:

Punktmenge: Die Punktmenge $V = P \cup R$ ist partitioniert in die Menge der beteiligten Prozesse und die der Ressourcen, $P = \{P_1, \ldots, P_n\}$, $R = \{R_1, \ldots, R_m\}$.

Request-Edges: Für jedes Paar (P_i, R_j) aus einem Prozess und einer von ihm angeforderten Ressource, die er noch nicht erhalten hat, enthält der Graph eine gerichtete Kante $P_i \to R_j$.

Assignment-Edges: Für jedes Paar (P_i, R_j) aus einem Prozess und einer ihm bereits zugeteilten Ressource enthält der Graph eine gerichtete Kante $R_j \to P_i$.

Grafisch werden die Ressourcen meist als Kästchen, die Prozesse als Kreise (und die gerichteten Kanten als Pfeile) dargestellt:



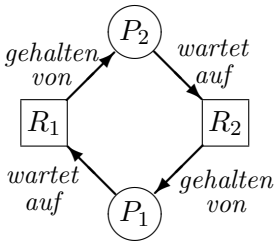
angeforderte Ressource

zugeteilte Ressource

Ein Prozess, aus dem ein Pfeil herausführt, ist momentan blockiert, einer, aus dem keine Pfeile herausführen, ist „ready“, kann also jederzeit vom Scheduler aktiviert werden.

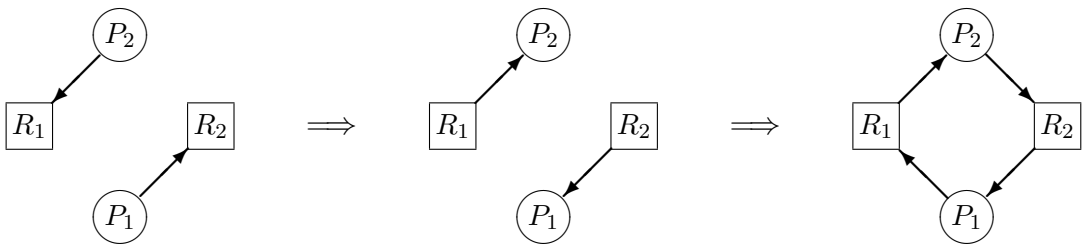
Wenn ein Pfeil in eine Ressource führt, aus der keine Pfeile herausführen, kann das System der dargestellten Anforderung entsprechen. Die Pfeilrichtung dürfte sich also innerhalb kurzer Zeit umkehren.

Rechts ist die kleinste mögliche Deadlock-Situation dargestellt. P_1 wartet auf die Zuteilung von R_1, das bereits P_2 zugeteilt wurde. P_2 kann R_1 aber nie freigeben, da es auf R_2 wartet, das im Besitz von P_1 ist.



Allgemein gilt, dass (wenn die ersten beiden Deadlock-Bedingungen erfüllt sind) das Auftreten eines Deadlocks äquivalent dazu ist, dass der Ressourcengraph einen gerichteten Zyklus enthält (siehe die Zyklus-Bedingung von Seite 259).

Der obige Graph kann auf folgende Weise nacheinander entstanden sein:

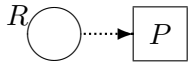


Folgende Verfeinerungen des Modells machen Sinn:

- Die Ressourcen R werden disjunkt aufgeteilt in **wiederverwendbare** Ressourcen R_r (*reusable*) und **verbrauchbare** Ressourcen R_c (*consumable*), also $R = R_r \cup R_c$.

  Die wiederverwendbaren sind von Anfang an in ihrer Größe konstant, die verbrauchbaren Ressourcen müssen von Prozessen aus P produziert werden. Im Ressourcengraph gibt es spezielle permanente gerichtete Kanten von den Ressourcen aus R_c zu den Prozessen, von denen sie produziert werden („**Producer-Edges**"). In den Grafiken werden sie gepunktet dargestellt.

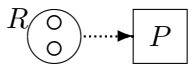
  Während sich im Lauf der Zeit die anderen Kanten ändern können, bleiben die Producer-Edges permanent erhalten. Der Fall, dass sich die Ressourcen zusammen mit ihrem Produzenten ganz verabschieden, wird (hier) nicht betrachtet.

- Man betrachtet Ressourcen mit mehreren „Instanzen" oder „**Units**". Beispielsweise hat die Ressource „Drucker" so viele Units, wie (gleichberechtigte) physische Drucker im System vorhanden sind. Falls es nicht ganz gleichgültig ist, welcher Drucker einer Anfrage zugeteilt wird, muss man diese Ressourcenklasse aufteilen, etwa in „Textdrucker" und „Grafikdrucker".

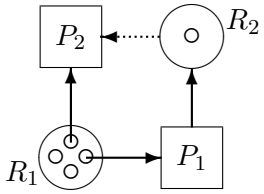
  Die Units werden in der Grafik meist durch Punkte innerhalb des Ressourcen-Kreises dargestellt. Aus einer Ressource mit k Units können im Graphen also maximal k Pfeile herausführen. (*In* jede Ressource können dagegen beliebig viele Pfeile hineinführen.)

  Verbrauchbare Ressourcen können unbegrenzt viele Units haben, die neu produziert und wieder verbraucht werden können.

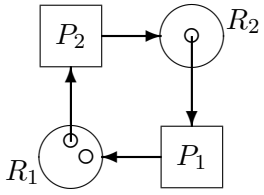
- Synchronisationsmittel wie zählende Semaphore können als verbrauchbare Ressourcen aufgefasst werden: Ein `up` produziert eine Unit, ein `down` verbraucht eine. Wenn gerade keine Unit vorhanden ist, blockiert ein `down`. Damit kann man dann auch solche Deadlocks mit diesem Modell behandeln, die durch fehlerhafte Synchronisation entstehen.

Im Beispiel rechts gibt es eine permanente Ressource R_1 mit vier Units, von denen jeweils eine von P_1 bzw. P_2 gehalten wird. Die verbrauchbare Ressource R_2 enthält momentan eine Unit. Sie wird von P_2 produziert und wurde im Moment von P_1 angefordert, ist ihm aber noch nicht zugeteilt.



Vorsicht: Mit Units ist das Enthaltensein eines Zyklus im Graphen nicht mehr *hinreichend* für das Auftreten eines Deadlock (nur noch *notwendig*, Beweis weiter unten)!

Beispielsweise enthält der Ressourcengraph rechts einen Zyklus, obwohl die dargestellte Situation keinen Deadlock erzeugt. P_1 kann als nächstes die noch freie Unit von R_1 erhalten, wodurch der Zyklus beseitigt ist. Es kann seine Arbeit beenden, R_2 freigeben, wodurch dann auch P_2 weiterarbeiten kann.



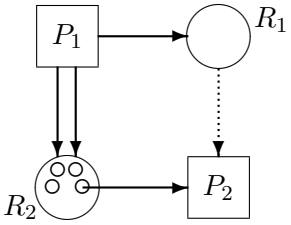
Die Prozesse verändern durch ihre Aktivität den Systemzustand, und dazu gehört auch der Verlauf der Kanten im Ressourcengraph. Die relevanten Operationen sind folgende:

Anfragen (Requests): Ein Prozess P_i im Ready-Zustand kann (atomar) eine Serie von Ressourcen R_j, R_k, ... anfragen. Dadurch werden Kanten (Request-Edges) $P_i \to R_j$, $P_i \to R_k$, ... in den Graphen eingefügt. Beachte, dass ein Prozess nur dann „ready“ ist, wenn nicht noch alte Anfragen unbeantwortet sind.

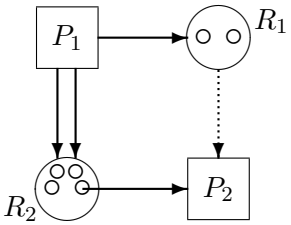
Zuteilungen (Acquisitions): Wenn alle von P_i angefragten Ressourcen verfügbar sind, kann P_i alle diese Ressourcen zugeteilt bekommen. Bei *permanenten* Ressourcen werden die Request-Edges von oben werden umgekehrt (sie werden zu Assignment-Edges). Eine Request-Edge in Richtung einer *verbrauchbaren* Ressource verschwindet (und die Ressource wird „einmal verbraucht“, eine Unit verschwindet).

Freigaben (Releases): Ein Prozess P_i im Ready-Zustand kann beliebig viele Ressourcen, die er hält, freigeben (es verschwinden die entsprechenden Assignment-Edges). Außerdem kann er für jede Ressource, für die er Produzent ist, beliebig viele Units produzieren.

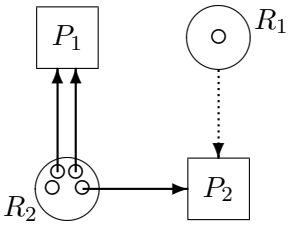
In der rechts dargestellten Situation hält P_2 eine Unit der wiederverwendbaren Ressource R_2 und ist nicht blockiert. Außerdem ist es der Produzent der verbrauchbaren Ressource R_1, von der es im Moment noch keine Units gibt.



P_1 fordert gerade zwei Units der wiederverwendbaren Ressource R_2 an, außerdem eine Unit von R_1. Durch die letzte Anforderung ist P_1 momentan blockiert.



Links hat P_2 gerade zwei Units der Ressource R_1 produziert. Dadurch ist P_1 nicht mehr blockiert. Rechts sieht man die Situation, nachdem seine Requests gewährt wurden. Die Request-Edges an R_2 sind zu Assignment-Edges geworden; eine Unit von R_1 wurde verbraucht.

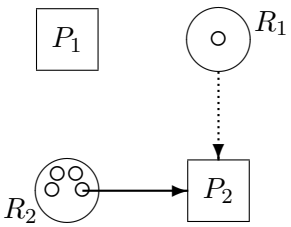


Unter der **Reduktion** eines Ressourcengraphen in einem nicht isolierten und nicht blockierten Punkt P_i versteht man folgendes:

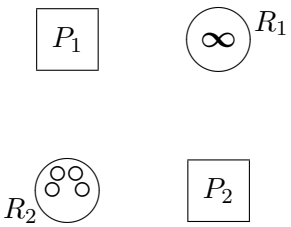
- die Durchführung aller denkbaren Zuteilungen, d.h. alle Kanten $P_i \to R_j$ werden entfernt. Die Anzahl der Units in verbrauchbaren Ressourcen R_j wird dabei um 1 erniedrigt.
- die Durchführung aller denkbaren Freigaben, d.h. alle Kanten $R_j \to P_i$ werden entfernt. *Formal* dürfen auch Production-Edges entfernt werden, die entsprechende wiederverwendbare Ressource erhält dafür *unendlich viele* Units.

Beachte, dass der betroffene Prozess nach dieser Prozedur *isoliert* ist.

Wenn keine Reduktion möglich ist (alle Prozesse sind isoliert oder blockiert), heißt der Zustand **irreduzibel**.



Im Beispiel von oben dürfen wir zu Beginn nicht in P_1 reduzieren, da der Prozess noch blockiert ist. Wenn wir dagegen später (in der mittleren oder der Endsituation) den Graphen zunächst nach P_1 und dann nach P_2 reduzieren, erhalten wir die Situationen links und rechts.



Wenn wir zu Beginn erst in P_2 reduzieren, können wir danach in P_1 reduzieren (R_1 hat ja dann unendlich viele Units) und kommen zum selben Schlussbild.

Mit Hilfe des Begriffs der Reduktion kann man nun folgenden Satz formulieren, der allerdings hauptsächlich für theoretische Betrachtungen nützlich ist:

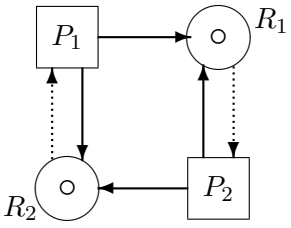
Satz 1: Ein (blockierter) Prozess P_i ist genau dann *nicht* an einem Deadlock beteiligt, wenn man den entsprechenden Graphen so weit reduzieren kann, dass P_i nicht blockiert ist.

Wenn insbesondere der Ressourcengraph des Systems vollständig in isolierte Punkte reduzierbar ist, existiert kein Deadlock im System. (Die Umkehrung hiervon gilt nicht!)

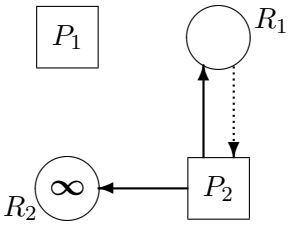
Der Satz folgt direkt aus der Definition der Reduktion. Eine Folge von Reduktionen entspricht ja offenbar genau einer Folge von Zustandswechseln des Systems, durch die der Prozess wieder lauffähig gemacht werden könnte.

Der Erfolg einer Reihe von Reduktionen hängt aber natürlich von ihrer Reihenfolge ab, und der Satz liefert aber keinen Algorithmus, der festlegt, wann welche Reduktion stattzufinden hat. Das Verfahren, alle Möglichkeiten durchzuprobieren, hat leider die Ordnung $\mathcal{O}(|R| \cdot |P|!)$ und ist in der Praxis nicht durchführbar.

Bemerkung: Wenn der Graph keinen Zyklus erhält, liefert er aber eine *lineare* Ordnung auf der Menge der Prozesse (bei Verzweigungen gibt es mehrere, in diesem Zusammenhang gleichwertige Ordnungen). Aus dem ersten (oder letzten) Punkt in dieser Ordnung führen keine Pfeile heraus, und der Prozess kann nicht blockiert sein. Wenn man hier beginnt und in der Reihenfolge der Ordnung reduziert, kann man den Graph vollständig reduzieren (insbesondere liegt kein Deadlock vor, also „Deadlock ⇒ Zyklus“).



Es gibt leider Situationen ohne Deadlock, deren Ressourcengraph nicht vollständig reduzibel ist, beispielsweise der linke. Wenn man nach P_1 reduziert, erhält man den Graphen rechts, der nicht weiter reduzierbar ist (P_2 ist blockiert). Reduziert man zuerst nach P_2, ergibt sich die symmetrische Situation.



Wir benötigen noch folgende Begriffe zu gerichteten Graphen:

- Ein Punkt P_j heißt von P_i aus **erreichbar**, wenn es eine Folge von Pfeilen $P_i \to \ldots \to P_j$ gibt. (Ein Punkt ist von sich selbst aus erreichbar, wenn er in einem Zyklus liegt.)
- Eine **Senke** ist ein Punkt, aus dem keine Pfeile herausführen.
- Ein **Knoten** ist eine nichtleere Teilmenge $K \subset V$, sodass von allen $v \in K$ aus *genau* alle Punkte in K erreichbar sind.

Beachte:

- In einem Knoten gibt es keine Senke (von hier aus ist kein Punkt erreichbar). Insbesondere ist ein isolierter Punkt kein Knoten.

- Ein Zyklus ist nicht notwendigerweise ein Knoten (es könnten Pfeile herausführen). Ein Knoten enthält aber immer einen Zyklus (jeder Punkt im Knoten muss sich selbst erreichen können).
- Ein Graph enthält genau dann keinen Knoten, wenn alle Punkte Senken sind oder von ihnen aus eine Senke erreichbar ist.

  ($\Leftarrow$ ist klar, zu $\Rightarrow$ überlegt man sich, dass eine Nicht-Senke, die keine Senke erreicht, einen Zyklus erreichen muss, der sogar in einem Knoten liegt, Widerspruch).

Ein Zustand, in dem alle Prozesse, die eine belegte Ressource anfragen, blockiert sind, heißt **Normalzustand**. Wenn eine unbelegte Ressource angefragt wird, wird sie sofort vergeben.

In den meisten Betriebssystemen sind gar keine anderen Situationen erlaubt. Wir setzen daher immer Normalzustände voraus.

Satz 2: Ein **Zyklus** im Ressourcen-Graph ist eine **notwendige** Bedingung für einen Deadlock (Deadlock $\Rightarrow$ Zyklus, kein Zyklus $\Rightarrow$ kein Deadlock).

Bei einem Normalzustand ist ein **Knoten** im Ressourcengraph eine **hinreichende** Bedingung für einen Deadlock (Knoten $\Rightarrow$ Deadlock, kein Deadlock $\Rightarrow$ kein Knoten).

Beweis der zweiten Teils: Ein Knoten besteht aus mindestens zwei Punkten, enthält also Prozesse und Ressourcen. Ein Prozess im Knoten ist keine Senke, erreicht also (mindestens) eine Ressource im Knoten direkt, wartet also auf sie (per Request-Edge). Die Prozesse, die diese Ressource produzieren oder freigeben könnten, sind von der Ressource aus erreichbar (per Producer- oder Assignment-Edge), liegen also auch im Knoten. Also warten alle Prozesse im Knoten auf Aktionen anderer Prozesse im Knoten (und sind aufgrund des Normalzustands blockiert). Daher liegt ein Deadlock vor.

10.4 Deadlock-Vermeidung

Um einem Deadlock überhaupt aus dem Weg zu gehen, muss das System dafür sorgen, dass eine der drei notwendigen Bedingungen von Seite 259 nicht eintritt.

Unter Deadlock-*Vermeidung* wollen wir hier verstehen, dass das System *grundlegende* Voraussetzungen schafft, dass es in keiner Situation zu einem Deadlock kommen kann – nicht, dass *spezielle* Deadlock-Situationen kurz vorher vorausgesehen und verhindert werden.

10.4.1 Exklusiver Zugriff

Ressourcen, auf die problemlos gleichzeitig zugegriffen werden kann, spielen keine Rolle für Deadlocks. Leider gibt es aber solche, die von ihrer Natur her nicht gemeinsam benutzbar sind, wie Dateien, in die geschrieben wird, Drucker, Bänder, Netz-Datenpakete, etc.

Immerhin kann man die Gefahr von Deadlocks verringern, wenn man die Ressourcen, bei denen das möglich ist, nur zum Lesen (*Read-Only*) hält.

10.4.2 Freiwillige Abgabe

Diese Voraussetzung kann für solche Ressourcen umgangen werden, die einem Prozess wieder entzogen werden können, indem ein spezielles Protokoll verwendet wird. Ein solches Entziehen ist beispielsweise möglich bei Seitendruckern nach Beendigung einer kompletten Seite, etc.

Die erste Möglichkeit ist, einem Prozess, der bereits Ressourcen hält und eine weitere anfordert, die nicht frei ist, die bereits gehaltenen automatisch zu entziehen. Sie werden in die Liste der Ressourcen eingefügt, auf die der Prozess wartet. Der Prozess läuft erst weiter, wenn er alle angeforderten Ressourcen auch erhalten kann.

Als zweite Möglichkeit könnten Ressourcen einem Prozess sofort entzogen werden, wenn ein anderer sie anfordert. Sie werden dann in die Warteliste des ersten Prozesses eingefügt.

Da dieses Vorgehen nicht für beliebige Ressourcen sinnvoll ist, kann hiermit wiederum nur die Deadlock-Gefahr vermindert, aber nicht vollständig beseitigt werden. Außerdem kann es hier eventuell zu Verhungern kommen.

10.4.3 Zyklisches Warten

Das zyklische Warten impliziert, dass es mindestens einen Prozess gibt, der bereits Ressourcen hält und auf eine weitere warten muss. Mit speziellen Anforderungs-Protokollen kann hier ein Deadlock vermieden werden. Die Prozesse werden aber so in ihrer Freiheit eingeschränkt.

Zusammengefasste Anforderung: Beispielsweise kann man verlangen, dass alle Ressourcen, die ein Prozess in einem Teil seines Codes benötigt, zusammen in einem Systemaufruf angefordert und später auch gemeinsam wieder zurückgegeben werden müssen. Danach kann er eine neue Anforderung absetzen, verschachtelte Anforderungen sind aber einfach verboten. Die Anforderungen werden jeweils erst dann erfüllt, wenn alle betroffenen Ressourcen frei sind.

Vorab-Anforderung: Noch einschränkender ist die Forderung, dass ein Prozess alle überhaupt während seiner gesamten Laufzeit benötigten Ressourcen bei seinem Start, also vor der ersten eigentlichen Anweisung, belegen muss. (Wenn sich die Ressource erst aus Eingabedaten ergibt, ist dieses Vorgehen natürlich nicht durchführbar.)

Beide Protokolle beinhalten allerdings die Gefahr des Verhungerns, besonders, wenn viele Ressourcen auf einmal angefordert werden, und die Chance groß ist, dass eine davon von irgendeinem anderen Prozess gehalten wird.

Vorgeschriebene Reihenfolge: Man kann eine totale Ordnung auf der Menge der Ressourcen einführen, z.B. in Form einer bijektiven Abbildung $f : R \to \{0, \ldots, n-1\}$.

Ein Prozess darf nun seine Ressourcen nur in der Reihenfolge *aufsteigender Nummerierung* anfordern. Wenn mehrere Units einer Ressource benötigt werden, müssen sie zusammen angefordert werden. Die Nummerierung sollte sinnvoll gewählt werden. Ressourcen, die üblicherweise zuerst benötigt werden, sollten also kleine Nummern erhalten.

Mit diesem Protokoll kann es nie zu zyklischem Warten kommen. Angenommen, es gäbe im Ressourcen-Graphen doch einen Zyklus $P_0 \to R_0 \to P_1 \to \ldots \to P_{m-1} \to R_{m-1} \to P_0$.

Dann gilt an jeder Stelle, dass P_i bereits im Besitz der Ressource $R_{(i-1) \bmod m}$ ist und die Ressource R_i zusätzlich anfordert. Nach dem Protokoll gilt also $f(R_{(i-1) \bmod m}) < f(R_i)$, also $f(R_0) < f(R_1) < \ldots < f(R_{m-1}) < f(R_0)$, ein Widerspruch.

In allen drei Fällen kann es dazu kommen, dass Geräte schlecht ausgenutzt werden. Sie sind eventuell nur zur Einhaltung des Protokolls schon oder noch von einem Prozess belegt, obwohl er im Moment gar keinen Gebrauch von ihnen macht.

Die Einschränkungen der Prozesse bei der Deadlock-Verhinderung sind darin begründet, dass wir jeweils an einem einzigen Prozess ansetzen. Wenn wir möglichst viel Freiheit gewähren wollen, müssen wir Deadlocks kurz vor ihrem Eintreten erkennen und abwenden. Das wollen wir unter *Deadlock-Verhinderung* verstehen.

Grundlage dazu sind allerdings Methoden, Deadlock-Situationen eindeutig zu erkennen.

10.5 Deadlock-Erkennung

Methoden zur Deadlock-Erkennung können auf zwei Weisen angewandt werden:

- Zur Vorab-Erkennung eines **drohenden** Deadlocks bei der Anforderung einer belegten Ressource. Der Deadlock wird dann **verhindert**, indem der entsprechende Prozess blockiert wird, bis seine Wünsche ohne Deadlock-Gefahr erfüllt werden können.
- Zur Erkennung eines schon **aufgetretenen** Deadlocks. Diese Überprüfung kann alle paar Sekunden oder Minuten durchgeführt werden. Danach sollten natürlich Maßnahmen eingeleitet werden, den Deadlock **aufzulösen**.

In einigen speziellen Fällen (bei eingeschränkten Ressourcen-Typen) gibt es Resultate, die zu besonders effizienten Algorithmen zur Deadlock-Erkennung führen. Diese wollen wir daher als erstes betrachten.

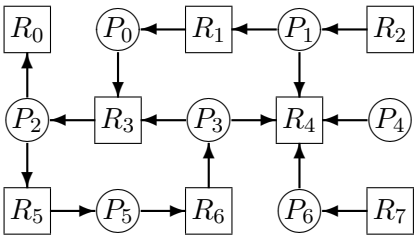
10.5.1 Einfache Ressourcen

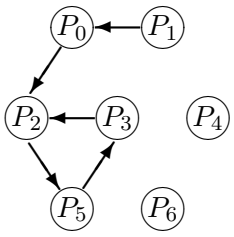
Wenn Ressourcen im einfacheren Sinn (also mit jeweils genau einer Unit) betrachtet werden, entspricht die Deadlock-Erkennung ja dem Aufspüren eines Zyklus im Ressourcen-Graphen. Das ist mit $\mathcal{O}((n+m)^2)$ Zeit möglich.

Man kann aus dem Ressourcen-Graph G_R sogar die Ressourcen-Punkte herausnehmen. Es entsteht dadurch ein neuer Graph G_W, der „**Wait-For Graph**“ (Warte-Graph), der angibt, welcher Prozess auf welchen anderen Prozess wartet (d.h. darauf, dass er eine Ressource freigibt). Es gibt in G_W genau dann eine Kante $P_i \to P_j$, wenn es in G_R eine Kantenfolge $P_i \to R_k \to P_j$ gibt.

Man kann die Betrachtung auf G_W beschränken, da es genau dann einen Zyklus in G_R gibt, wenn es einen Zyklus in G_W gibt. Damit sind $\mathcal{O}(n^2)$-Algorithmen möglich.

Unten ist ein Ressourcen-Graph und der zugehörige Wait-For-Graph dargestellt, in dem der vorhandene Zyklus besonders schnell zu finden ist:





Unten ist ein einfacher rekursiver Algorithmus in C++ notiert (Funktion `has_cycle`), der in der Zeit $\mathcal{O}(n^2)$ einen Zyklus findet. Der Graph wird ihm als Matrix präsentiert – es ist `arrow[i][j]=true` genau dann, wenn es einen Pfeil $P_i \to P_j$ gibt.

Die Matrix des Graphen von oben ist (der Lesbarkeit halber mit 0 und 1) rechts angegeben.

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

```
#define N 7
const bool arrow[N][N]= { ... };   // s.o.
bool been_there[N];

static bool in_cycle(int i)
 {
   been_there[i]=true;
   for (int k=0;k<N;++k)
      if ( arrow[i][k] && ( been_there[k] || in_cycle(k) ) )
         return true;
   return false;
 }

bool has_cycle()
 {
   for (int i=0;i<N;++i)
    {
      for (int j=0;j<N;++j) been_there[j]=false;
      if (in_cycle(i)) return true;
    }
   return false;
 }
```

10.5.2 Mehrfache Ressourcen, einfache Anfragen

Wir betrachten Systeme (mit Normalzuständen), in denen die Ressourcen zwar mehrere Units haben dürfen, diese aber nicht beliebig angefragt werden dürfen. Ein Prozess darf auf maximal eine Unit derselben Ressource warten. Wenn er mehrere Units benötigt, kann er natürlich nacheinander auf sie warten, sodass wir hier keine große Einschränkung vornehmen.

Satz 3: In solchen Systemen ist ein Knoten im Ressourcen-Graphen äquivalent zu einem bestehenden Deadlock.

Beweis: Nach Satz 2 gilt schon „Knoten ⇒ Deadlock“, wir müssen noch zeigen: „kein Knoten ⇒ kein Deadlock“.

Ohne Knoten gibt es von jedem Prozess P_i aus einen Weg in eine Senke: $P_i \to R_j \to \ldots P_k \to R_\ell \to P_m$, wobei (wegen des Normalzustands) die Senke ein Prozess ist. Dann ist P_m nicht blockiert, und wir können den Graph in P_m reduzieren. Dadurch vergrößert sich die Anzahl freier Units von R_ℓ echt. Daher kann nun P_k nicht blockieren. Wir können den Graph also immer weiter reduzieren, bis wir bei P_i ankommen. P_i ist nicht blockiert, und da die Betrachtung für alle Prozesse gilt, gibt es keinen Deadlock.

Auf dieser Beobachtung basiert der Algorithmus auf der rechten Seite (in Pseudo-Code), der erkennt, ob der Zustand S ein Deadlock-Zustand im Ressourcen-Graph (V, E) ist. Er arbeitet mit einer geordneten Menge M von Punkten.

In den ersten Durchgängen der **while**-Schleife werden alle direkten Vorgänger von Senken in die Menge eingefügt, danach deren Vorgänger, etc. Genau dann, wenn die Menge zum Schluss alle Punkte im Graph enthält, gibt es von allen Punkten aus einen Pfad in eine Senke, d.h. es gibt keinen Knoten im Graph, also keinen Deadlock.

```
bool is_deadlocked(V,E)
 {
   M= Senken in V;
   i=0;
   while i < |M|
    {
      for all p : p → M_i
          M.append(p);
      ++i;
    }
   return ( M ≠ V );
 }
```

Es gibt keine Kanten zwischen zwei Prozessen bzw. zwei Ressourcen, und in den hier betrachteten Systemen gibt es maximal eine Kante von einem Prozess zu einer Ressource. Bei n Prozessen und m Ressourcen hat der Graph also maximal $2nm$ Kanten. In der inneren Schleife werden maximal alle diese Kanten in die Menge eingefügt. Das Testen, ob eine Kante bereits enthalten ist, kann in $\mathcal{O}(1)$ erfolgen (z.B. per Bitfeld). Daher hat der Algorithmus die Ordnung $\mathcal{O}(nm)$.

10.5.3 Mehrfache Ressourcen, nur wiederverwendbar

Auch in Systemen, in denen alle Ressourcen wiederverwendbar sind, gibt es effiziente Deadlock-Algorithmen.

Satz 4: In solchen Systemen ist das Nichtvorhandensein eines Deadlocks äquivalent dazu, dass der Ressourcen-Graph vollständig reduzibel ist.

Wenn es keine verbrauchbaren Ressourcen gibt, ist es gleichgültig, in welcher Reihenfolge die Reduktionen vorgenommen werden: Die Anzahl verfügbarer Units ändert sich bei einer Reduktion nicht. Man erhält immer denselben irreduziblen Graphen.

Satz 1 bedeutet gerade, dass ein vollständig reduzibler Graph keinen Deadlock darstellt. Wenn es umgekehrt keinen Deadlock gibt, kann man für jeden Prozess den Graphen so weit reduzieren, dass dieser Prozess nicht blockiert ist. Von dort aus kann man den Graphen weiter reduzieren und landet immer beim selben irreduziblen Graphen, der also nur aus isolierten Punkten besteht, d.h. der Ressourcen-Graph ist vollständig reduzibel.

Auf diesem Satz beruht der Algorithmus rechts. Er arbeitet mit einer Menge (z.B. als Liste) von Prozessen. Die am Ende in der Menge verbliebenen Prozesse befinden sich in einem Deadlock.

```
M=P;
do
 {
   reducible=false;
   for all p ∈ M
      if ( !blocked(p) )
       {
         reducible=true;
         reduce_graph(p);
         M=M \ {p};
         break;
       }
 }
while (M ≠ ∅ && reducible);
```

Im ungünstigsten Fall benötigt er $\mathcal{O}(mn^2)$ Zeit (möglicherweise n Durchgänge beim **`while`**, darin n Durchgänge beim **`for`**, m beteiligte Ressourcen-Kanten beim Reduzieren). Es gibt effektivere Versionen ($\mathcal{O}(mn)$), die aber kompliziertere Datenstrukturen benötigen.

10.6 Deadlock-Auflösung

Wenn ein Deadlock aufgetreten (und erkannt worden) ist, kann man versuchen, ihn nachträglich aufzulösen.

10.6.1 Ressourcen-Entzug

Manchmal ist es möglich, einem Prozess eine gehaltene Ressource wieder zu entziehen und einem anderen Prozess zuzuteilen. Das bietet sich allerdings nur bei wenigen Ressourcen an, wie beim Seitendrucker nach Beendigung einer Seite. Es muss natürlich gewährleistet sein, dass die dann gemischt ausgegebenen Seiten mehrerer Prozesse letztendlich den richtigen Besitzer erreichen. Eventuell ist auch der Eingriff eines Administrators nötig oder sinnvoll.

10.6.2 Rollback

Ein anderer Ansatz ist es, einen der an einem Deadlock beteiligten Prozesse anzuhalten und in der Zeit zurückzuversetzen („**Rollback**“). Dazu ist natürlich ein spezielles Protokoll notwendig, das von jedem Prozess in gewissen Zeitabständen ein Abbild aller notwendigen Informationen anfertigt („**Checkpointing**“).

In diesem Abbild müssen alle Informationen des Prozess-Kontrollblocks, ein Speicherabbild und Daten zu den gerade gehaltenen Ressourcen enthalten sein.

Wenn das Rollback eines einzelnen Prozesses nicht ausreicht, kann man versuchen, weitere Prozesse zurückzusetzen. Wenn alle beteiligten Prozesse zurückgesetzt sind, der Deadlock aber immer noch besteht, hat man ihn nicht rechtzeitig genug erkannt. Dem kann man begegnen, indem man mehrfaches Checkpointing betreibt: Das jeweils zurückliegende Abbild wird nicht überschrieben, sondern eine Serie von Abbildern aufbewahrt.

Checkpointing ist offensichtlich sehr zeit- und speicheraufwendig, dafür aber sehr effektiv. Es kann für sehr viele Arten von Ressourcen und in beliebigen Situationen eingesetzt werden.

10.6.3 Prozess-Termination

Das ist ein Spezialfall von Rollback: Der Prozess wird gewissermaßen zum Zeitpunkt seiner Geburt zurückversetzt, d.h. gewaltsam beendet und ggf. später neu gestartet. Bei einigen speziellen Prozessen bietet sich dieses Verfahren an – beispielsweise bei der Compilation eines Programms, wo der Ablauf allein von den Eingabe-Quelltexten abhängt und jederzeit wiederholt werden kann.

10.7 Deadlock-Verhinderung

Viele Algorithmen zur Vermeidung von Deadlocks im allgemeinen Fall werden dadurch ermöglicht, dass man von den Prozessen zusätzliche Vorab-Informationen einholt, wie sie ihre Ressourcen-Anforderungen zukünftig gestalten werden.

Meistens wird verlangt, dass a priori bekannt ist, wieviel Units welcher Ressource jeder Prozess **maximal** (gleichzeitig) benötigen wird. Diese Angabe muss jeder Prozess bei seinem Start selbst machen. Er kann natürlich nie mehr deklarieren, als das System insgesamt zur Verfügung stellt.

Obwohl auf diese Weise gut verwendbare Algorithmen möglich sind, ist diese Forderung ein wenig realitätsfern. Kein bekanntes Betriebssystem arbeitet jedenfalls mit solchen Deklarationen.

Ein Nachteil fast all solcher Algorithmus ist es, dass nur Deadlocks verhindert werden, die sich durch den gemeinsamen Zugriff auf Ressourcen ergeben – nicht solche, die bei Synchronisationsversuchen (etwa mit zählenden Semaphoren) auftreten.

10.7.1 Sichere Zustände

Die meisten Algorithmen arbeiten mit dem Begriff von „sicheren Zuständen“. Ein Zustand wird als sicher angesehen, wenn es noch möglich ist, den Prozessen *in einer bestimmten Reihenfolge* alle ihre (als maximal deklarierten) Ressourcen zuzuteilen, genauer:

Ein Zustand heißt **sicher**, wenn es eine Anordnung $(P_1, \ldots, P_n)$ aller Prozesse gibt, sodass (für alle i) die von P_i benötigten Ressourcen frei sind oder von Prozessen P_j mit $j < i$ gehalten werden.

Die Idee hierbei ist folgende: Falls die Bedürfnisse von P_i momentan nicht erfüllt werden können, kann P_i einfach warten, bis alle P_j mit $j < i$ beendet sind. Dann erhält es die Ressourcen, arbeitet damit, beendet sich und ermöglicht damit P_{i+1} den Zugriff auf seine Ressourcen, etc.

Offensichtlich garantiert ein solcher sicherer Zustand, dass kein Deadlock auftritt. Ein unsicherer *kann* dagegen einen Deadlock andeuten. Die Algorithmen, die nur sichere Zustände garantieren, sind also vorsichtiger als nötig.

Beispiel: Wir betrachten zunächst eine Situation, in der es nur eine Ressource (mit mehreren Units) gibt. In den Tabellen wird in die Spalte „max“ die Anzahl der Units einer Ressource eingetragen, die ein Prozess als maximal deklariert hat. In der Spalte „hat“ sind die momentan gehaltenen Units aufgeführt:

Es sollen 10 Units der Ressource existieren. Hier besitzen die Prozesse $3 + 2 + 2 = 7$ Units, sodass 3 Units frei sind.

	max	hat
P_1	9	3
P_2	4	2
P_3	7	2
	10	3

Dieser Zustand ist sicher: (P_2, P_3, P_1) ist eine mögliche Reihenfolge, in der die Prozesse ablaufen können. Der Zeile des jeweils aktiven Prozesses ist fett dargestellt. Alle Prozesse erhalten jeweils ihre Maximalanforderung, sind irgendwann beendet und geben ihre Ressourcen zurück:

	max	hat	max	hat	max	hat	max	hat	max	hat	max	hat	max	hat
P_1	9	3	9	3	9	3	9	3	9	3	9	9	–	–
P_2	4	2	**4**	**4**	–	–	–	–	–	–	–	–	–	–
P_3	7	2	7	2	7	2	**7**	**7**	–	–	–	–	–	–
	10	3		1		5		0		7		1		10

Wenn der Anfangszustand sicher ist, führt jede beliebige Prozess-Reihenfolge zum Ziel, vorausgesetzt, dass immer eine Maximalforderung erfüllt wird. Es wird ja davon ausgegangen, dass der Prozess irgendwann terminiert und alle Ressourcen zurückgibt. Die Anzahl freier Units kann so also höchstens größer werden (die schon belegten Ressourcen werden auch zurückgegeben).

Das gilt nicht, wenn auch Teilforderungen gewährt werden! Wir probieren so beispielsweise $(P_1, P_2, \ldots)$:

	max	hat	max	hat	max	hat
P_1	9	3	**9**	**4**	9	4
P_2	4	2	4	2	**4**	**4**
P_3	7	2	7	2	7	2
	10	3		2		0

Der letzte Zustand ist unsicher. P_1 und P_3 sind blockiert. Wenn P_2 beendet ist, sind wieder 4 Ressourcen frei, was aber die Maximalanforderungen von P_1 und P_3 nicht befriedigt. Wenn nichts freigegeben wird, kommt es zu einem Deadlock.

Beispiel: Wir betrachten noch eine Situation mit drei Ressourcen mit 10, 5 und 7 Units, um die 5 Prozesse konkurrieren. Es erscheint eine Spalte für jede Ressource unter „max“ bzw. „hat“. Der Anfangszustand ist sicher, eine mögliche Prozess-Reihenfolge ist $(P_1, P_3, P_0, P_2, P_4)$:

	max			hat			max			hat			max			hat			max			hat		
P_0	7	5	3	0	1	0	7	5	3	0	1	0	7	5	3	0	1	0	7	5	3	0	1	0
P_1	3	2	2	2	0	0	**3**	**2**	**2**	**3**	**2**	**2**	–	–	–	–	–	–	–	–	–	–	–	–
P_2	9	0	2	3	0	2	9	0	2	3	0	2	9	0	2	3	0	2	9	0	2	3	0	2
P_3	2	2	2	2	1	1	2	2	2	2	1	1	2	2	2	2	1	1	**2**	**2**	**2**	**2**	**2**	**2**
P_4	4	3	3	0	0	2	4	3	3	0	0	2	4	3	3	0	0	2	4	3	3	0	0	2
	10	5	7	3	3	2				2	1	0				5	3	2				5	2	1

	max			hat			max			hat			max			hat			max			hat		
P_0	7	5	3	0	1	0	**7**	**5**	**3**	**7**	**5**	**3**	–	–	–	–	–	–	–	–	–	–	–	–
P_1	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
P_2	9	0	2	3	0	2	9	0	2	3	0	2	9	0	2	3	0	2	**9**	**0**	**2**	**9**	**0**	**2**
P_3	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
P_4	4	3	3	0	0	2	4	3	3	0	0	2	4	3	3	0	0	2	4	3	3	0	0	2
	10	5	7	7	4	3				0	0	0				7	5	3				1	5	3

	max			hat			max			hat			max			hat		
P_0	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
P_1	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
P_2	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
P_3	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
P_4	4	3	3	0	0	2	**4**	**3**	**3**	**4**	**3**	**3**	–	–	–	–	–	–
	10	5	7	10	5	5				6	2	4				10	5	7

Algorithmus (Test auf sicheren Zustand):

Wenn die Maximalforderung eines Prozesses erfüllt werden kann, wird sie (nur rein rechnerisch) erfüllt, und es wird davon ausgegangen, dass der Prozess irgendwann beendet ist und alle seine Ressourcen zurückgibt. Insgesamt kehren bei diesem Vorgang also die ursprünglich schon reservierten Ressourcen in den Ressourcen-Pool zurück.

Welchen Prozess man jeweils auswählt, ist gleichgültig, da sich bei der obigen Rechnung die Ressourcen-Zahl ja höchstens vergrößern kann.

Es wird so lange auf diese Weise verfahren, bis alle Prozesse befriedigt sind (dann ist der Zustand sicher) oder kein Prozess mehr bedient werden kann (dann ist der Zustand unsicher, da es zu einem Deadlock kommen könnte).

10.7.2 Der Bankiers-Algorithmus

Der Bankiers-Algorithmus (*Banker's Algorithm*, von Dijkstra 1965) ist der wohl bekannteste zur Deadlock-Verhinderung im allgemeinen Fall. Er hat den Namen daher, dass man ihn abgewandelt auch in einer Bank verwenden könnte (Kunden ≈ Prozesse, Geld ≈ Ressourcen), um ständige Zahlungsfähigkeit zu gewährleisten.

Er erfordert die bereits bekannten Maximal-Deklarationen der Ressourcen. Er wird jedesmal zu Rate gezogen, wenn ein Prozess Ressourcen anfordert, und überprüft jeweils, ob sich durch die Zuteilung ein sicherer Zustand ergeben würde. In diesem Fall wird die Zuteilung so vorgenommen, ansonsten muss der Prozess warten (bis er geweckt wird, wenn eine der angeforderten Ressourcen frei wird).

Es sind einige Datenstrukturen notwendig, die permanent (also über einen Aufruf hinaus) gespeichert werden müssen.

Diese Strukturen müssen sich eigentlich vergrößern bzw. verkleinern, wenn Prozesse starten bzw. beendet sind, neue Typen z.B. von verbrauchbaren Ressourcen hinzukommen, etc. Der Algorithmus kann so formuliert werden – der Übersicht wegen verwenden wir hier aber statische Arrays, d.h. es gibt immer konstant viele Prozesse (`n`) und Ressourcen-Typen (`m`).

Bei den zweidimensionalen Arrays steht eine einfache Indizierung (wie in `C` üblich) für den Zugriff auf eine Zeile, z.B. `Max[i]`. Der Operator $\leq$, auf Arrays bezogen, ist elementweise zu verstehen.

`const int n;`	Anzahl der Prozesse
`const int m;`	Anzahl der Ressourcen
`int avail[m];`	`avail[j]` legt die Anzahl der verfügbaren Units der Ressource `j` fest.
`int max[n][m];`	`max[i][j]` legt die Maximalzahl der von Prozess `i` zur selben Zeit anforderbaren Units vom Typ `j` fest. Jeder Prozess muss zu Beginn „deklarieren“, wieviel er später maximal zu verbrauchen gedenkt.
`int alloc[n][m];`	`alloc[i][j]` gibt die Anzahl der momentan dem Prozess `i` zugeteilten Units der Ressource `j` an.
`int need[n][m];`	gibt an, welche Prozesse noch auf maximal wieviele Units der Ressourcen warten könnten. Es handelt sich um ein Hilfsarray, denn es ist immer `need=max-alloc`.

Bankiers-Algorithmus:

- Es sei nun `int request[m]` der Vektor der Anforderungen, die gerade von Prozess `i` getätigt werden.
- Falls nicht `request`$\leq$`need`, hat der Prozess seine Maximal-Deklaration missachtet. Die Anfrage wird mit einem Fehler beendet.

- Falls nicht `request`$\leq$`avail`, sind nicht alle angeforderten Ressourcen verfügbar. Der Prozess wird hier blockiert, bis er durch eine Freigabe aufgeweckt wird.
- Ansonsten sind alle angeforderten Ressourcen verfügbar. Das System ändert die Datenstrukturen nun so, als würde es die Ressourcen auch tatsächlich vergeben:

  ```
  avail -= request;
  alloc[i] += request;
  need[i] -= request;
  ```

  Wenn der so erreichte Zustand sicher ist, werden die Ressourcen tatsächlich so vergeben. Ansonsten wird die Änderung an den Arrays rückgängig gemacht, und der Prozess legt sich schlafen. Wenn er aufgeweckt wird, wird dieser Schritt des Algorithmus wiederholt.

Zur besseren Übersicht ist der Algorithmus noch in Pseudocode beschrieben:

```
void bankers_allocate(int i, int request[m])
 {
   if (request > need[i] error();
   while (request[i]>avail) wait();

   for (;;)
    {
      avail-=request;  alloc[i]+=request;  need[i]-=request;

      if (safe_state()) { do_allocate(i, request); return; }

      avail+=request;  alloc[i]-=request;  need[i]+=request;

      wait();
    }
 }
```

Der folgende „Unter-Algorithmus“, mit dem der Bankiers-Algorithmus testet, ob der neue Zustand sicher ist („`safe_state`“), hält sich an die Beschreibung von Seite 271:

Sicherheits-Test:

- Es gibt folgende Hilfsvariablen:

  ```
  int work[m]=avail;                          // Kopie von avail zum Testen
  bool finished[n]={ false, ... , false };    // Prozess bedient & beendet?
  ```

- solange es ein `i`$\in \{0,\ldots,$`n`$-1\}$ gibt mit `!finished[i]` und `need[i]`$\leq$`work`:

  ```
  finished[i]=true;                 // Prozess wird bedient und beendet sich
  work+=alloc[i];                   // ⇒  vorher reservierte Ressourcen frei
  ```

- Der Zustand ist genau dann sicher, falls am Schluss `finished[i]=true` für alle `i`.

Der Algorithmus arbeitet in der Zeit $\mathcal{O}(\mathtt{mn}^2)$ (maximal `n` Durchgänge, ein weiteres `n` durch das Suchen nach einem Prozess in jedem Durchgang, `m` durch die Vektor-Addition). Es gibt aufwendigere Algorithmen der Ordnung $\mathcal{O}(\mathtt{mn})$.

Wenn wir ihn auf unser 3-Ressourcen-Beispiel von Seite 271 anwenden, erhalten wir die dort schon angegebene Reihenfolge.

11 Speicherverwaltung

11.1 Grundlagen

Nach dem Prozessor ist der Hauptspeicher das wichtigste Betriebsmittel. Es handelt sich um den Speicher, den die CPU direkt (ohne I/O-Vorgang) erreichen kann. Die Zugriffsgeschwindigkeit ist gegenüber den Sekundärspeichern wie Platte um Größenordnungen höher.

In allen gängigen Rechnerarchitekturen (die auf von Neumann basieren) ist der Hauptspeicher **linear** angeordnet: jede Speicherzelle hat eine Nummer ≥ 0 (ihre **Adresse**). Die Menge aller im System gültigen Adressen heißt (physischer) **Adressraum**. Je nach Hardware muss er allerdings nicht unbedingt zusammenhängend sein.

Im einem von-Neumann-ähnlichen System liegen diverse unterschiedliche Arten von Daten im Speicher, u.a.:

Programmcode,	Strukturspeicher (Stack),
konstante Daten,	Bildschirminhalte,
Zwischenergebnisse,	I/O-Puffer,
Verwaltungsstrukturen.	

Daher wird der Hauptspeicher ja auch als „von-Neumannscher *Flaschenhals*“ bezeichnet. Nicht nur die Hardware muss entsprechend schnell konzipiert sein, auch das Betriebssystem muss den Speicher möglichst effizient verwalten.

11.1.1 Virtueller Speicher

In Multitasking-Systemen liegen fast immer viele Prozesse gleichzeitig im System vor, sodass Speicherbereiche mehrerer Prozesse nebeneinander im Hauptspeicher liegen. Bei Speichermangel muss zeitweise auf Bereiche von Sekundärspeichern (*backing store*, meist auf einer schnellen Festplatte) zurückgegriffen werden:

Swapping: Ganze Prozesse werden aus dem Hauptspeicher auf den Sekundärspeicher „*ausgelagert*“.

Paging: Der Speicher wird in „Seiten“ aufgeteilt, und nur einzelne Seiten werden ausgelagert.

Zugriffe auf Sekundärspeicher sind aber um Größenordnungen langsamer als Hauptspeicherzugriffe. Für die Performance des gesamten Systems ist daher die Verwirklichung der Speicherverwaltung von großer Bedeutung und war früher, bei teurem und vergleichsweise kleinen Hauptspeicher, geradezu existentiell.

Der Begriff „Virtueller Speicher“ ist mit dem Begriff „Multitasking“ konzeptionell verwandt:

Virtualisieren des Betriebsmittels CPU $\Rightarrow$ Multitasking
Den Prozess braucht es i.Allg. nicht zu interessieren, mit welchen anderen Prozessen er gleichzeitig lebt. Er kann die CPU so behandeln, als wäre er allein im System.

Virtualisieren des Betriebsmittels Speicher $\Rightarrow$ Virtueller Speicher
Jeder Prozess verwendet einen linearen Adressraum ab der Adresse 0. Der wirklich verwendete Speicher kann wild im Hauptspeicher oder auf Platte verstreut liegen. Die Umsetzung wird per Hardware vorgenommen.

11.1.2 Adressräume und Memory Management

Durch die Einführung von virtuellem Speicher muss man nun zwei Adressräume voneinander unterscheiden:

Logischer Adressraum: die Menge von Adressen, die die CPU (in einem bestimmten Prozess-Kontext) verwenden kann.

Ein Prozess kann fast beliebig viel Hauptspeicher adressieren, der nicht wirklich gerade im System frei oder überhaupt physisch vorhanden zu sein braucht. Der Bereich dieser Adressen wird *logischer Adressraum* genannt, die Adressen heißen *logische Adressen* oder *virtuelle Adressen*. Ein Prozess verwendet üblicherweise Adressen in wenigen zusammenhängenden Bereichen.

Physischer Adressraum: die Menge von Adressen, über die der physische Hauptspeicher tatsächlich angesprochen wird.

Die Speicherverwaltung des Systems ist „transparent“ für das Benutzerprogramm. Das System bildet automatisch die *tatsächlich angesprochenen* Teile des großen logischen Adressraums stückweise auf Bereiche im physischen Hauptspeicher ab – oder ggf. eben zwischenzeitlich auf Plattenspeicher.

Die Maßnahmen zur Speicherzuteilung und -verwaltung (und oft auch den entsprechenden Betriebssystem-Teil) nennt man **Memory Management**. Die Hardware zur Umsetzung zwischen den beiden Adressräumen heißt **Memory Management Unit** (**MMU**, *address translation device*). Wenn Teile ausgelagert werden, spricht man auch von einem dritten Adressraum („*auxiliary address space*“, auf dem Sekundärspeicher).

- Zu beachten ist, dass jeder *Prozess* einen *eigenen* logischen Adressraum haben darf, der sich mit denen anderer Prozesse überlappen kann. Das gelingt durch die Veränderung der Parameter der MMU bei einem Context Switch.
- Die Größen des logischen Adressraums und des physisch möglichen Adressraums stimmen meist überein und sind durch die Anzahl von Adressleitungen der CPU und die Länge der internen Adressregister festgelegt – bei 32 Bit z.B. 2^{32} Byte = 4 GByte.
- Obwohl ein Prozess theoretisch den logischen Adressraum ausnutzen und seine Daten wild darüber verstreuen könnte, sind seine Zugriffe auf realen Systemen auf wenige Bereiche beschränkt. Anders wäre eine Speicherverwaltung nicht denkbar.

Die wichtigsten Aufgaben des Memory Managements sind folgende:

- Mehrere Prozesse sollen „**gleichzeitig**“ ausführbar sein, auch wenn nicht genügend physischer Hauptspeicher vorhanden ist, um alle aufzunehmen.
- Trotz Speicherbewegungen durch wechselnde Prozess-Situationen soll die **System-Performance** nicht deutlich absinken.

- Die Speicherbereiche der einzelnen Prozesse sollen gegeneinander **abgeschottet** sein, d.h. ein Prozess soll vor unbeabsichtigtem oder böswilligen Auslesen oder Überschreiben seiner Daten geschützt werden (*memory protection*).
- In sinnvollen Situationen sollen aber Speicherbereiche geschaffen werden können, die sich **mehrere Prozesse** geordnet **teilen** (*shared memory*).
- Die Speicherverwaltung soll für den Programmierer **transparent** sein. Er soll nur mit einem großen linearen logischen Adressraum arbeiten müssen.

11.1.3 Segmente

Die Daten eines Prozesses sind von ganz unterschiedlicher Natur. Die gröbste Unterteilung ist die in Code und Daten. Weiterhin ist aber der Code logisch unterteilt in eigenen Code und Bibliotheks-Code, in Hauptprogramm und Unterprogramme, etc. Diese Struktur kann (mit gewissen Vorteilen) auf den logischen Adressraum übertragen werden, was man als „**Segmentierung**“ bezeichnet.

Je nach Hardware kann es ausgefeilte Segmentierungs-Strukturen geben (später genauer). Fast immer gibt es aber zumindest die folgende Aufgliederung des logischen Adressraums:

- **Code-Segment** (CS, für den Maschinencode, bei UNIX „Text-Segment“)
- **Daten-Segment** (DS, ggf. unterteilt, s.u.)
- **Stack-Segment** (SS, für den Laufzeit-Stack, also für Rücksprung-Adressen, Funktions-Parameter und lokale Variablen)

Bei den Daten muss noch eine zweifache logische Unterteilung gemacht werden:

Statische Daten sind die, die aus statischen Variablen-Definitionen des Quelltexts resultieren. In `C` werden sie durch globale oder in Unterprogrammen statische Definitionen festgelegt.

Diese Daten können vorinitialisiert sein (wie bei `int table[]={1,2,3};`) oder nicht (`int i,j,k;`). Die entsprechenden Konstanten für erstere können direkt aus dem Objektcode geladen werden, der Platz für zweitere *kann* bei der Initialisierung gelöscht werden (muss aber nicht). Das Segment wird meist entsprechend unterteilt in einen Teil mit konstanten Daten und einen nicht-initialisierten Teil (BSS, *binary storage segment*).

Dynamische Daten sind die, die durch Aufrufe wie `malloc` während des Programmlaufs angefordert und danach über Pointer verwaltet werden. Mittels `free` wieder freigegebene Bereiche sollen günstig wiederverwendet werden, wozu teilweise komplizierte und effiziente Datenstrukturen eingesetzt werden. Auf Systemebene wächst der insgesamt für diesen Zweck vorgesehene Speicherbereich („Heap“) aber üblicherweise in eine Richtung, wenn jeweils nötig.

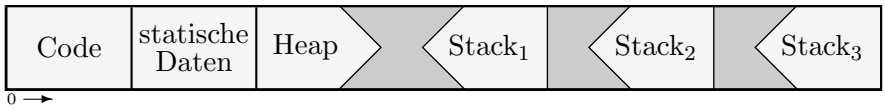
Von den oben angegebenen Segmenten behalten einige während des Programmlaufs ihre Größe strikt bei (Code und statische Daten), einige wachsen und schrumpfen (Heap und Stack).

Meistens wird jedem Prozess ein zusammenhängender logischer Adressbereich zugewiesen, bei dem die beiden variabel großen Bereiche aufeinander zu wachsen:

Bei den gängigen Prozessor-Architekturen ist die Wachstumsrichtung des Stacks durch die Funktionsweise der Stack-Befehle festgelegt. Meistens wächst der Stack zu niedrigeren Adressen hin wie im Bild angedeutet.

Normalerweise wird eine mehrere Megabyte große Lücke zwischen Heap und Stack gelassen, sodass es fast nie zu einer Überschneidung kommt (z.B. Stack Overflow bei zu tiefer Verschachtelung). Tritt der Fall doch einmal ein, muss er unbedingt erkannt werden, da es sonst sofort zu Daten-Inkonsistenzen und sehr schnell zu einem Absturz kommt. Sinnvollerweise muss so eine Überprüfung durch Hardware vorgenommen werden, was heutzutage fast immer der Fall ist (siehe spätere Abschnitte).

Bei Systemen mit mehreren Threads innerhalb eines Prozesses gibt es einen Heap, aber mehrere Stacks innerhalb eines Adressraums. Die Verteilung ist üblicherweise wie folgt:



Die Hardware-Verwaltung wird dann etwas komplizierter, da sich nun auch benachbarte Stacks nicht überschneiden dürfen.

11.1.4 Adressen in ausführbaren Programmen

Meistens entsteht ausführbarer Programmcode aus der Compilation eines Programms, das in einer höheren Programmiersprache (wie `C` oder `C++`) geschrieben ist (oder ggf. in Assemblersprache).

Dort wird nicht direkt mit Speicheradressen, sondern mit symbolischen Adressen oder abstrakten Strukturen gearbeitet: mit Hilfe von **Variablen** und **Pointern** (für die Datenbereiche), bzw. **Kontrollstrukturen** und **Unterprogramme** (für den Codebereich).

11.1.4.1 Compiler

Der Compiler oder Assembler übernimmt eine Umsetzung in eine bestimmte Art von logischen Adressen:

- Wenn (bei alten Systemen, insbesondere ohne virtuellen Speicher) das entstehende Programm ausschließlich an einem bestimmten Platz im Hauptspeicher liegen darf, kann bereits der Compiler feste physische Adressen für alle Objekte vergeben und diese in den erzeugten Maschinenbefehlen verwenden (*absolute code*).

  Prozesse an festgelegten Stellen blockieren aber ggf. das System. Wenn der Platz bereits teilweise belegt ist, kann der Prozess nicht starten. Prozesse, die teilweise denselben Platz belegen, können nicht gleichzeitig im Speicher liegen. Wenn das Programm an einer anderen Stelle liegen soll, muss es (mit der entsprechenden Angabe) neu compiliert werden.

- Üblicherweise sollen die Prozesse an beliebigen Stellen ausführbar sein. Daher erzeugt der Compiler nur relative Adressen. Das wird teilweise auf Maschinensprache-Ebene auf bestimmte Weise unterstützt:

  - Bei kurzen Sprüngen wird nicht direkt das Sprungziel, sondern die Sprungdistanz (zum aktuellen Befehl) angegeben.
  - Lokale Variablen liegen auf dem Stack, der mit dem Stack-Pointer (einem Prozessor-Register) verwaltet wird. Die Adresse ist überhaupt erst zur Laufzeit festgelegt. Sie wird im Befehl als die Distanz zum Stack-Pointer codiert.
  - Bei Daten kann mit einem oder mehreren weiteren Registern gearbeitet werden, zu denen relativ adressiert wird (dazu später).

  Solche Befehle können beliebig im Speicher verschoben werden, ohne dass sich die Adressbezüge ändern müssten. (Außerdem sind diese Befehle kürzer als solche mit großen absoluten Adressen.)

- Bei Daten-Zugriffen ohne Spezialregister, bei längeren Sprüngen und Unterprogramm-Aufrufen gibt es aber keine so handhabbaren Befehle.

  Bei ihnen erzeugt der Compiler relative Adressen, bezogen auf den Anfang des ganzen Programms (oder ggf. zum Anfang des Daten-Teils).

  Für Bezüge auf andere Code-Teile oder Bibliotheken kann noch gar keine Adresse erzeugt werden. Es wird eine Tabelle mit abgespeichert, die beispielsweise die Namen der externen Unterprogramme enthält, zusammen mit allen Stellen im Code, an denen sie per Adresse angesprochen werden.

  Der entstehende Code heißt *relozierbarer Objektcode* (*relocatable object code*) und ist so nicht direkt lauffähig.

11.1.4.2 Linker

Der normale (statische) Linker dient dazu, mehrere einzeln übersetzte Programmteile und Bibliotheken (statisch) miteinander zu verbinden – zu einer festen großen Einheit.

Jeder Teil besitzt *eigene relative* Adressen, bezogen jeweils auf *seinen* Anfang. Der Linker hängt alle Teile aneinander und muss an jedem Befehl mit solchen Adressen entsprechende Umsetzungen vornehmen.

So können nun zwar alle beim Compilieren „freigelassenen“ Adressen (für externe Bezüge) festgelegt werden. Die Adressen, die schließlich entstehen, sind aber immer noch relativ, nämlich bezogen auf den Anfang des neuen Gesamtcodes. Der Code kann immer noch nicht direkt ausgeführt werden.

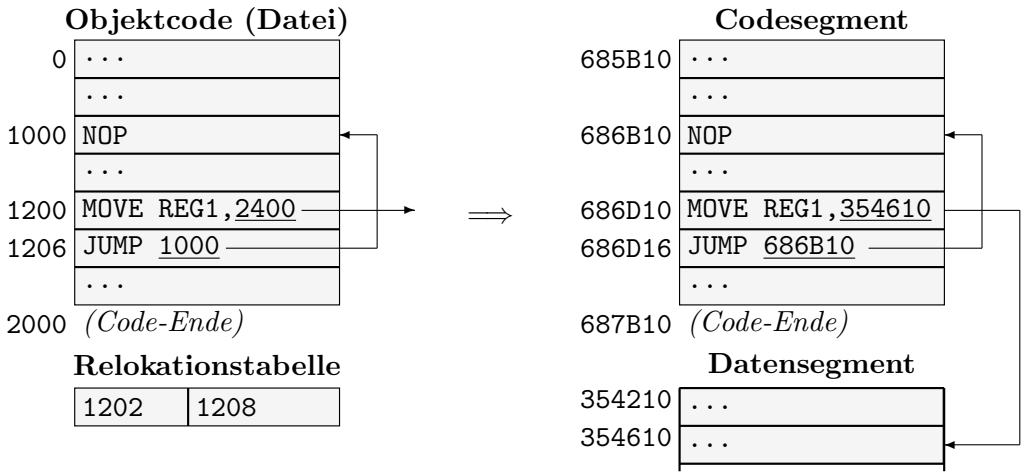
11.1.4.3 Loader

Der Loader ist der Betriebssystem-Teil, der ausführbare Programme in Form von Dateien einliest und im Hauptspeicher lauffähig macht.

Der belegte Speicher kann irgendwo im Hauptspeicher zu liegen kommen. Wenn es keine Hardwareunterstützung gibt, muss der Loader die relativen Adressen aus dem Objektcode in echte physische Adressen umsetzen. Dazu muss er die Anfangsadresse des Speicherbereichs auf alle relativen Angaben im ganzen Code addieren. Dieser Vorgang heißt **Relokation**.

Es muss dafür im Objektcode zusätzlich zum eigentlichen Programm eine Tabelle („Relokationstabelle“) abgelegt werden. In ihr sind all die Positionen *im Objektcode* angegeben, an denen Veränderungen vorgenommen werden müssen. (Befehle, die in sich relative Angaben machen wie oben besprochen, fallen nicht in diese Kategorie!)

Das folgende Bild (mit fiktiver Assemblersprache) soll dies verdeutlichen:



Im Beispiel wird von einem Datensegment ausgegangen, das im Objektcode Adressen direkt oberhalb des Codebereichs (ab `$2000`) erhält. Dieser Bereich wird erst nach dem Laden überhaupt physisch realisiert. Die physische Adresse zur logischen Adresse `$2400` liegt dann mit einem Offset von `$400` im Datensegment.

11.1.4.4 Linker-Loader

Ein spezieller Linker-Loader ist notwendig, wenn mit dynamischer Bindung gearbeitet wird. Es brauchen dann nicht sofort alle Teile geladen zu werden, nur der Start-Teil, insbesondere nicht die Bibliotheken. Bibliotheken, die für solche Verwendung gedacht sind, heißen *shared libraries* oder *dynamic link libraries* (z.B. die `*.DLL`s unter Windows).

Wenn eine Routine aus einer solchen Bibliothek das erste Mal angesprochen wird, wird ein spezieller Zwischencode (*Stub*) aufgerufen, der den angeforderten Code (oder die ganze Bibliothek) nachlädt, wenn er nicht (durch ein anderes Programm) bereits im Speicher steht. Dazu benutzt er meist den Loader, der ggf. Adressumsetzungen wie oben vornehmen muss. Dann modifiziert sich der Zwischencode selbst, sodass bei weiteren Aufrufen direkt die neue Routine erreicht wird. Schließlich ruft er die Routine das erste Mal auf.

Durch dieses *dynamische Linken* können sich mehrere Prozesse dieselbe Bibliothek teilen, ohne dass sie mehrfach im Speicher zu stehen braucht (daher das „*shared*“). Außerdem wird eine Bibliothek gar nicht geladen, wenn sie in einem speziellen Programmlauf gar nicht benötigt wird. Beispielsweise könnte der Code auch zwischen zwei Bibliotheken *auswählen*.

Außerdem lassen sich solche Bibliotheken leicht gegen neue Versionen *austauschen* (die natürlich vollständig aufwärtskompatibel sein sollten). Die Programme, die sie benutzen, brauchen nicht neu (statisch) gelinkt zu werden.

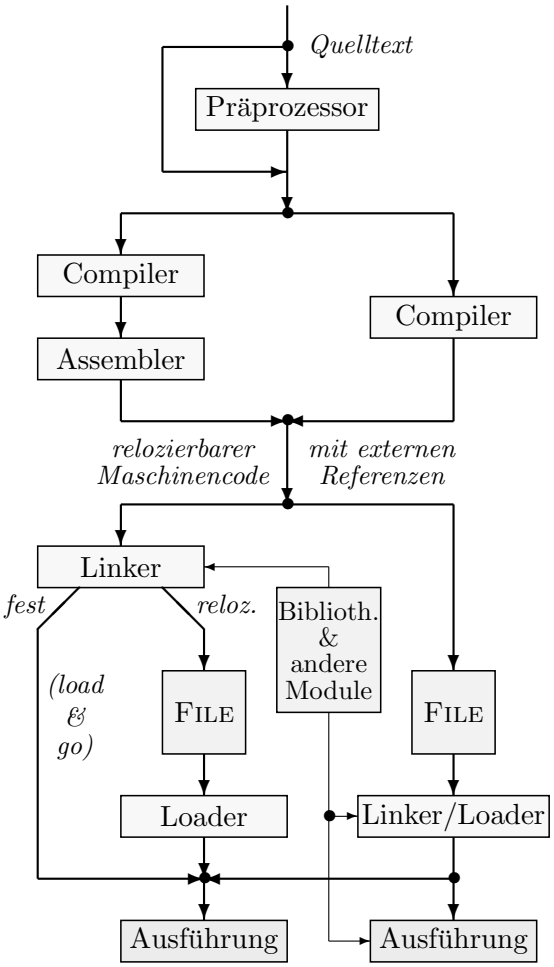
Der Name des Standard-UNIX-Linkers `ld` kommt übrigens von seiner Teilaufgabe auch als *Loader*.

11.1.4.5 Laufzeit

Wenn die vom Loader erzeugten Adressen immer noch keine physischen Adressen sind, findet zur Laufzeit eine Adressumrechnung per Hardware statt – es liegt virtueller Speicher vor. Die logischen Adressen sind dann beispielsweise immer noch auf den Code-Anfang (=0) bezogen und können direkt denen aus dem Objektcode entsprechen.

Auf diese Weise können beispielsweise (in Multitasking-Systemen) Prozesse nachträglich verschoben werden, ohne dass eine Änderung der Maschinenbefehle (wie beim Loader) notwendig wäre. Außerdem braucht die Adressumsetzung nicht mehr linear zu sein. Je nach Speicherlage kann der Speicherbereich des Prozesses zerrissen und wild im Hauptspeicher verteilt werden, bei Speichermangel sogar auf Sekundärspeicher ausgelagert werden.

Das folgende Bild stellt noch einmal die wichtigsten Abläufe zwischen Compilation und Programmlauf zusammen:



Wenn das System den Objektcode eines Programms lädt und als neuen Prozess startet, geschieht also u.a. folgendes:

- Der Code wird auf Gültigkeit überprüft (z.B. mit „Magic Numbers“, die am Anfang der Datei stehen müssen).
- Ein Header mit Informationen wird eingelesen (Größe und Position der weiteren logischen Teile der Datei).
- Der benötigte Speicher wird angelegt und der Code, bzw. die Daten hineinkopiert. Auf Systemen mit virtuellem Speicher wird ggf. weniger physischer Speicher reserviert und nur ein Teil eingeladen, z.B. der Start des Code-Segments (genaueres später).
- Wenn es ein Segment für nicht-initialisierte Daten gibt, wird dieser (meistens) mit Nullen vorbelegt. Manche Compiler speichern diesen Bereich auch unnötigerweise in der Datei mit ab, sodass er mitgeladen werden muss!
- Ebenso wird ggf. der Stack mit Nullen initialisiert (nicht unter UNIX und Windows).

11.1.5 Compiler- und Maschinensprache-Ebene

Wir schauen uns noch kurz an, wie die unterschiedlichen Klassen von Adressbezügen im Objektcode in Compilaten entstehen.

11.1.5.1 Statische Objekte

Bei Variablen (Objekten) sorgt der Compiler dafür, dass zur Laufzeit genügend Speicher für das Objekt zur Verfügung steht. Dieser Speicherbereich kann über den Variablennamen angesprochen werden. Die Adressierung geht ganz in der Abstraktion als Objekte auf.

Bei Assemblern und Assembler-Sprache kann man immerhin logische Namen für Adressen vergeben, sodass man sich nicht um die Details der Speicherpositionen (wohl aber um die einzelnen Speicher*größen*) kümmern muss.

Wir betrachten kurze Ausschnitte aus einem `C-` (bzw. `C++`)-Programm und das Compilat des GNU-Compilers unter L*i*nux-x86. (Dazu kann man entweder mit „`-S`“ Assembler-Quelltext erzeugen lassen oder den Objektcode z.B. mit `gdb` disassemblieren). Die Notation ist dabei *nicht* die Intel-Notation, sondern die von AT&T V/386– insbesondere ist die Operandenreihenfolge „Quelle,Ziel“!

```
int func(int a, int b)
 {
   int c=a-b;     // Umweg mit Absicht
   return c;
 }

int i,j;          // global mit Absicht
```

```
int main()
 {
   i=1;
   j=i+1;
   return func(i,j);
   ...
 }
```

Der Code-Teil für die Zuweisungen in `main` sieht wie folgt aus:

```
0x8048423 <main+3>:   movl   $0x1,0x8049558    ; Konstante 1 nach i
0x804842d <main+13>:  movl   0x8049558,%edx    ; i in das Register edx
0x8048433 <main+19>:  incl   %edx              ; edx um 1 erhöhen
0x8048434 <main+20>:  movl   %edx,0x8049554    ; edx nach j
```

Wir sehen, dass sich Adressen im Code- und Datensegment um ca. 4 KByte unterscheiden. Außerdem sind die Adressen für `i` und `j` in absteigender Reihenfolge vergeben worden.

11.1.5.2 Automatische Objekte

Automatische Variablen, die in Unterprogrammen in mehreren Instanzen existieren können, liegen auf dem Stack. Dazu gehören hier die beiden Parameter an die Funktion `func`. Der Aufruf von `func` sieht entsprechend so aus:

```
0x804843a <main+26>:  movl   0x8049554,%eax    ; j nach eax
0x804843f <main+31>:  pushl  %eax              ; eax auf den Stack
0x8048440 <main+32>:  movl   0x8049558,%eax    ; i nach eax
0x8048445 <main+37>:  pushl  %eax              ; eax auf den Stack
0x8048446 <main+38>:  call   0x8048410 <func>  ; func aufrufen
0x804844b <main+43>:  addl   $0x8,%esp         ; Stack-Pointer um 8 korrigieren
```

Aus technischen Gründen (um variabel lange Parameterlisten in `C` zu ermöglichen), werden die Parameter in umgekehrter Reihenfolge auf den Stack geschoben. Sie werden nicht explizit wieder (einzeln) heruntergenommen, stattdessen wird einfach der Stack-Pointer durch eine arithmetische Operation entsprechend korrigiert.

Wir haben in diesem Stück

- zwei Bezüge auf das Datensegment (beim Zugriff auf `i` und `j`),
- zwei Bezüge auf das Stacksegment (beim Erzeugen der Übergabe-Argumente, d.h. bei den beiden `pushl`; die Adresse ergibt sich u.a. aus dem Register `esp`)
- und einen Bezug auf das Code-Segment (die Anfangsadresse von `func`).

In `func` gibt es direktere Adressbezüge im Stack-Segment, nämlich für die Übergabe-Parameter `a` und `b`, sowie die lokale Variable `c`:

```
0x8048410 <func>:     pushl  %ebp           ; ebp retten (auf den Stack)
0x8048411 <func+1>:   movl   %esp,%ebp      ; Stack-Frame anlegen
0x8048413 <func+3>:   subl   $4,%esp        ; Platz fur 4 lokale Bytes (c)
0x8048416 <func+6>:   movl   8(%ebp),%edx   ; a nach edx
0x8048419 <func+9>:   subl   12(%ebp),%edx  ; b von edx subtrahieren
0x804841c <func+12>:  movl   %edx,-4(%ebp)  ; edx nach c
0x804841f <func+15>:  movl   -4(%ebp),%eax  ; c nach eax
                      ...                   ; technisch (Alignment)
0x8048430 <func+32>:  movl   %ebp,%esp      ; Stack-Frame auflösen
0x8048432 <func+34>:  popl   %ebp           ; Stack aufräumen
0x8048433 <func+35>:  ret                   ; Rückkehr aus der Funktion
```

Die Funktion sieht so etwas länglich aus, weil ein „Stack-Frame“ angelegt wird, der Platz für die zusätzliche lokale Variable `c` bietet. Die Adressbezüge innerhalb des Stack-Segments sind unterstrichen (jeweils relativ zum Stack-Pointer).

11.1.5.3 Pointer

Beim Arbeiten mit Pointern hat man es etwas direkter mit Adressen zu tun. Die erste Zuweisung an einen Pointer entsteht aber immer durch die Referenzierung eines Objekts (oder aus einem anderen Pointer). Weitere Veränderungen können sich zusätzlich arithmetisch (durch Objekt-Abstände) ergeben. Man braucht nie wirklich mit den physischen Adressen zu tun zu haben.

```
int main()
 {
   int *p=malloc(2*sizeof(int));
   *p++=1;  *p=2;
   free(p-1);
 }
```

Der erzeugte Code für die beiden mittleren Zuweisungen ist folgender:

```
0x80484c3 <main+19>:   movl   -4(%ebp),%eax  ; p nach eax
0x80484c6 <main+22>:   movl   $1,(%eax)      ; 1 an die Stelle, auf die p zeigt
0x80484cc <main+28>:   addl   $4,-4(%ebp)    ; p um 4 (Bytes) erhöhen
0x80484d0 <main+32>:   movl   -4(%ebp),%eax  ; (verändertes) p nach eax
0x80484d3 <main+35>:   movl   $2,(%eax)      ; 2 an die Stelle, auf die p zeigt
```

11.1.5.4 Code-Adressen

Im Codebereich kommen Adressen genau bei Sprungbefehlen vor. Sprünge werden in höheren Programmiersprachen zu Kontrollstrukturen (wie Schleifen) oder zu Aufrufen von Unterprogrammen abstrahiert.

```
int main()
 {
   for (int i=5;i>0;--i) ;
 }
```

```
0x8048417 <main+7>:    movl   $5,-4(%ebp)          ; 5 nach i
0x804841e <main+14>:   cmpl   $0,-4(%ebp)          ; Vergleich mit 0
0x8048422 <main+18>:   jg     0x8048430 <main+32>  ; i>0, dann Sprung
0x8048424 <main+20>:   jmp    0x8048440 <main+48>  ; Schleifenende anspringen
                       ...                         ; technisch (Alignment)
0x8048430 <main+32>:   decl   -4(%ebp)             ; i um 1 erniedrigen
0x8048433 <main+35>:   jmp    0x804841e <main+14>  ; Schleifenkörper anspringen
```

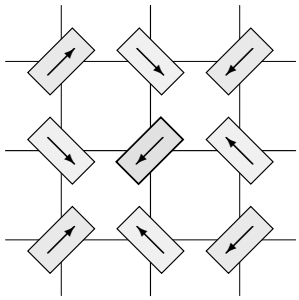
In diesem Stück kommen drei relative Sprünge vor (unterstrichen), die in der Disassemblierung nicht als solche herauskommen (man erkennt sie daran, dass der Maschinenbefehl nur 2 Bytes lang ist). Im Befehl wird die Sprungdistanz (relativ zum Programmzähler) angegeben, der Disassembler zeigt die sich so ergebende absolute logische Adresse an.

11.2 Hauptspeicher-Hardware

Die verbreiteten unterschiedlichen Arten von organisiertem Digitalspeicher haben folgende Gemeinsamkeiten:

- Die kleinsten Elemente speichern ein Bit, sind aber manchmal lokal oder global in Gruppen zu 4,8,... Bits organisiert.
- Die Elemente werden in Form eines zweidimensionalen Arrays angeordnet, die Adressen dementsprechend in zwei Teile aufgeteilt. So wird die Schaltungslogik für das Ermitteln der Zelle zu einer gegebenen Adresse verringert.
- Wenn der Speicher in Worten zu n Bits organisiert ist, wird er meist in n gleich große Teile aufgeteilt, aus dem für ein Wort jeweils ein Bit gelesen wird. Dadurch, dass jedes Bit innerhalb eines Teils an derselben Stelle liegt, wird die Adressierungslogik vereinfacht.

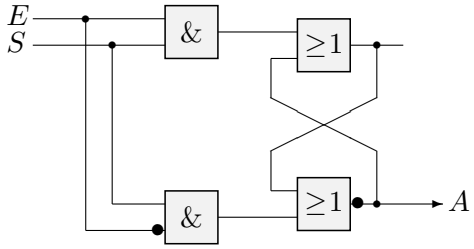
11.2.1 Halbleiterspeicher



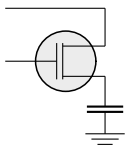
Die erste Generation von Hauptspeicher (in den 50er Jahren) war der sogenannte **Kernspeicher**. Als Speicherelemente verwendete er ringförmige Ferritkerne, auf ein Draht-Array aufgefädelt. Ein Kern kann jeweils zwei mögliche Magnetisierungsrichtungen aufweisen. Er wurde durch Strom durch beide beteiligten Drähte angesprochen: je nach sich ergebendem Stromfluss wurde auf die Magnetisierungsrichtung geschlossen.

Mit dieser Hardware war natürlich keine Miniaturisierung möglich. Nach diesen altertümlichen Zeiten wurden Halbleiterspeicher verwendet, von denen es zwei Typen gibt:

Das Grundelement eines **statischen RAMs** ist das „**Flip-Flop**“, das aus Logikschaltungen und damit letztendlich aus Dioden- oder Transistor-ähnlichen Baugruppen aufgebaut ist.



Jedes Flip-Flop hat zwei Eingänge: einen Wert-Eingang (das Bit), einen zweiten, der die Übernahme des Werts in den Speicher bewirkt. Links ist das Prinzip eines Flip-Flops dargestellt. (Um Rückkopplungen in größeren Netzen zu verhindern, sind echte Flip-Flops komplizierter aufgebaut.)



Die Wirkung von **dynamischen RAMs** beruht auf dem Kondensator-Effekt; zum Einschreiben und Auslesen ist eine Transistor-Einheit vorgeschaltet. Die (auf Mikrochips) winzigen Kondensatorplatten können ihre Ladung aber nur kurz halten. Deshalb müssen die Inhalte solcher RAMs in sehr kleinen Zeitabständen wieder aufgefrischt werden (daher das „dynamisch“).

Wie an den Bildern oben schon zu erkennen, ist dynamischer Speicher einfacher aufgebaut, einfacher zu miniaturisieren und wesentlich billiger als statischer Speicher. Allerdings braucht er nach jedem Zugriff eine „Erholungszeit“, bis er wieder ausgelesen werden kann. Meist ist er wesentlich langsamer als die CPU. Innerhalb der CPU wird (für Register) statischer Speicher verwendet.

Sehr häufig wird auf aufeinanderfolgende Speicherzellen zugegriffen (z.B. bei der Abarbeitung eines Programms). Um solche Zugriffe bei dynamischen RAMs zu beschleunigen, wird der Speicher meist in mehrere „**Bänke**“ aufgeteilt (*interleaving*). Bei n Bänken enthält Bank Nummer

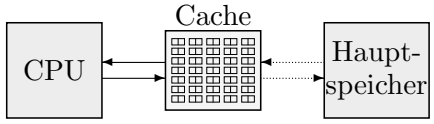
i die Speicherzellen mit Adressen $\equiv i \bmod n$. Nach n Zugriffen ist die zuerst angesprochene Bank (hoffentlich) wieder bereit.

Eine weitere Möglichkeit zur Beschleunigung ist der Einsatz von Caches.

11.2.2 Caches

Ein Cache ist ein *Pufferbereich* zwischen CPU und Hauptspeicher. Er ist wesentlich kleiner als der Hauptspeicher, aber im Zugriff schneller (und erheblich teurer).

Die Idee ist, dass einmal gelesene Daten im Cache gepuffert werden. Bei einem weiteren Lesezugriff auf dasselbe Datum wird es aus dem (schnelleren) Cache geholt. Daten, auf die lange nicht zugegriffen wurde, werden zugunsten neuer Daten aus dem Cache geworfen.



Dadurch werden besonders Schleifen (Zugriff auf Maschinencode) und Operationen mit lokalen Variablen von Hochsprachefunktion (Zugriff auf Daten) beschleunigt. Prozessoren mit relativ wenigen CPU-Registern (z.B. ältere Intels) profitieren von letzterem erheblich.

Jede Hauptspeicher-Operation der CPU geht zunächst an den Cache. Beim Lesen gibt es dabei zwei Fälle:

Cache Hit: das angeforderte Datum liegt schon im Cache und kann hieraus gelesen werden.

Cache Miss: das angeforderte Datum liegt nicht im Cache und muss (leider) aus dem Hauptspeicher gelesen werden. Es wird bei dieser Gelegenheit aber für eventuelle weitere Zugriffe im Cache untergebracht.

- Die Strategie, welche Cache-Daten zugunsten neuerer entfernt werden, ist natürlich in Hardware implementiert. Es wird in irgendeiner Weise markiert, wie lange der letzte Zugriff zurückliegt, wie oft bereits auf ein Datum zugegriffen wurde, etc. Fast immer wird eine LRU-Methode (*Least Recently Used*, dazu später beim Paging) verwendet.

- Wenn der Cache „gut eingestellt ist“, d.h. die richtigen Entscheidungen beim Austausch von Daten trifft, ist die Trefferquote gut, und man spricht von einem *heißen Cache* (bzw. bei niedriger Quote von einem *kalten Cache*).

- Wenn die durchschnittliche Wahrscheinlichkeit, dass ein zu lesendes Datum im Cache steht, $p \in [0, 1]$ ist, erhält man die durchschnittliche Speicherzugriffszeit t_{av} als

  $$t_{\text{av}} = p \cdot t_{\text{cache}} + (1 - p) \cdot t_{\text{memory}}.$$

- Der Hauptspeicherzugriff ist bei normalen Programmen „referenzlokal“. Es wird immer längere Zeit auf hintereinanderliegende Daten zugegriffen, z.B. bei Schleifen (Code) oder Arrays (Daten).

  Daher gibt es spezielle sehr schnelle Modi für den Transfer von Blöcken zwischen Cache und Hauptspeicher („**Bursts**“). Bei einem Cache Miss wird also üblicherweise nicht nur das angeforderte Datum, sondern (parallel zum CPU-Betrieb) vorausschauend auch einige dahinterliegende in den Cache gelesen.

  Mit normal dimensionierten Caches werden so Trefferquoten von 90 % und mehr erreicht.

- Der Cache ist meist unterteilt in einen Teil für Code (**Instruction Cache**) und einen für Daten (**Data Cache**). Diese beiden Arten von Zugriffen unterscheiden sich immer im Adressbereich und auch im durchschnittlichen Verhalten über einige Zeit hinweg. Es wäre ungünstig, wenn durch Datenzugriffe ständig Code-Teile aus dem Cache geworfen würden, die dann doch wieder gebraucht werden.

- Ein **Schreibzugriff** landet meist auch nicht direkt im Hauptspeicher, sondern im Cache. In den Cache-internen Strukturen wird die entsprechende Hauptspeicherzelle als verändert (*dirty*) markiert. Erst, wenn dieses Element aus dem Cache geworfen wird, wird das veränderte Datum tatsächlich in den Speicher geschrieben („*deferred write*“). Dabei kann oft von den schnellen Block-Operationen Gebrauch gemacht werden.

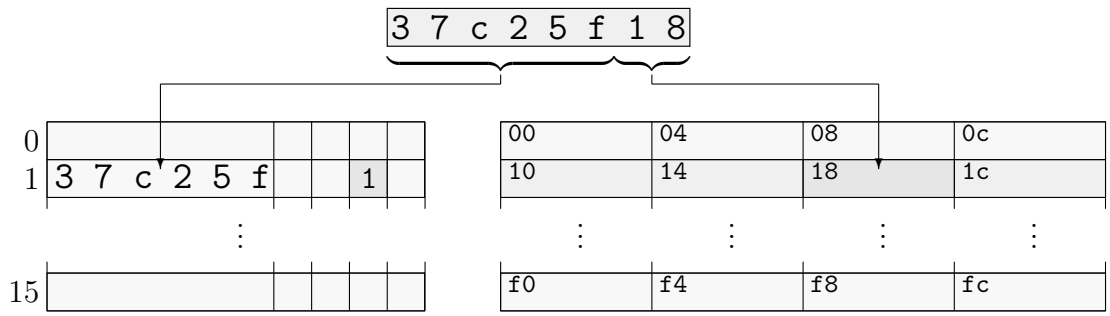
  Wenn bei jedem Schreibzugriff außer in den Cache auch direkt in den Speicher geschrieben wird, spricht man von einem „*Write-Through Cache*“.

- Es ist oft eine Hierarchie mehrerer Caches implementiert. Es gibt fast immer einen kleinen „**First-Level-Cache**“ schon auf dem Prozessor, der dadurch besonders schnell ansprechbar ist. Ein größerer „**Second-Level-Cache**“ kann dann auf dem Motherboard untergebracht werden und ist etwas langsamer. Beispielsweise hat ein Pentium-II 32 KByte Cache im Prozessorkern und 512 KByte noch im erweiterten Prozessor-Gehäuse.

Der Cache ist eine einfache Form von **Assoziativspeicher**. Bei solchen Speichern wird ein Element nicht über seine Nummer (Adresse), sondern über Teile seines Inhalts angesprochen. Die Elemente sind üblicherweise als Paare (*Schlüssel,Wert*) gegliedert. Das Ansprechen erfolgt über *Schlüssel*, und als Inhalt zurückgeliefert wird *Wert*. Beim Cache ist die Aufteilung naheliegenderweise (*Hauptspeicher-Adresse,Inhalt des Speicherzelle*).

- Die Cache-Hardware ist so aufgebaut, dass nicht nach dem Schlüssel (also der Adresse) *gesucht* werden muss. Das Element mit dem passenden Schlüssel „fühlt sich angesprochen“, die anderen nicht.

- Es wäre fast unmöglich, größere Caches zu bauen, die vollständige Adressen als Schlüssel speichern. Die Hardware für wenige KBytes könnte schon kaum noch auf dem Prozessor untergebracht werden. Daher sind die meisten Caches nur Teil-Assoziativspeicher. Es wird nur ein Teil der Adresse gespeichert, und der fehlende Teil ergibt sich aus der Position innerhalb des Caches. Als Einschränkung ergibt sich, dass nicht beliebige einzelne Speicherzellen, sondern kleine zusammenhängende Gruppen gecacht werden. Im praktischen Einsatz hat sich erwiesen, dass die Wirkung von Caches auf diese Weise fast nicht eingeschränkt wird.



Im Bild ist ein einfacher First-Level-Cache (der Instruction Cache des Motorola 68030) dargestellt (256 Bytes). Der Cache besitzt 16 „Lines“, jede Line fasst 16 *aufeinanderfolgende* Bytes in Form von 4 Langwörtern. Dadurch brauchen nur die obersten 24 Bits der Adresse (statt 32) gespeichert zu werden. Vorn ist eine Tabelle mit „Valid-Flags“ untergebracht, die anzeigen, ob an der entsprechenden Stelle im Cache auch ein gültiges Datum steht.

11.3 Speicherverwaltung ohne Auslagern

11.3.1 Multitaskinglose Systeme

Die einfachste vorstellbare Situation ist es, wenn nur ein Prozess zur selben Zeit im System existieren kann und bis zu seiner Beendigung im Speicher verbleibt (also kein Multitasking vorliegt). Weitere Prozesse „warten“ ggf. auf Platte, organisiert als „**Input Queue**“ (der Begriff wurde später auch in anderem Zusammenhang beibehalten).

An Großrechnern gab es solche Systeme nur in den fünfziger Jahren. Auf frühen Mikrocomputern (bis zum PC) feierte diese Regelung Wiederauferstehung.

Das Betriebssystem braucht hier keine eigentliche Speicherverwaltung zur Verfügung zu stellen. Der Benutzerprozess kann die gesamte Maschine übernehmen.

11.3.2 Eine Partition

Bei diesem Schema wird ein Teil des Hauptspeichers fest für Verwendung des Betriebssystems vorgesehen und ein fester anderer Bereich (Partition) einem Benutzer-Programm zur Verfügung gestellt. In dieser Partition darf genau ein Prozess zur gleichen Zeit liegen.

Wenn ein solches System Multitasking unterstützt, müssen also bei einem Context-Switch komplette Prozesse aus- und eingelagert werden (reinstes *Swapping*).

Die Lage des Betriebssystem-Teils orientiert sich dabei meist an den Vorgaben der Hardware. Wenn der Interrupt-Vektor an niedrigen Adressen liegt, wird man auch das System dorthin legen.

MS-DOS lag beispielsweise ursprünglich im niedrigsten Adressbereich. Das BIOS-ROM wird in den höchsten Adressbereich des 8086 (knapp unter 1MB) eingeblendet. Der Platz dazwischen war für „das“ Anwenderprogramm vorgesehen.

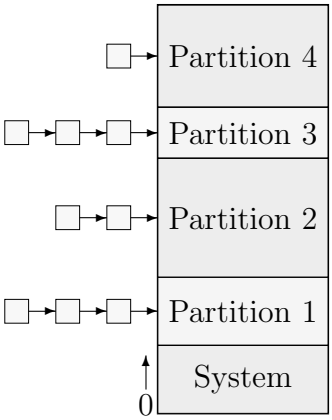
(Wie wir schon gesehen haben, verlagert DOS in späteren Versionen Teile seiner selbst in höhere Bereiche, und auf eingeschränkte Weise konnten mehrere Prozesse in der Benutzer-Partition existieren.)

11.3.3 Feste Partitionen

Bei Multitasking und mehreren Prozessen im Speicher ist es eine sehr einfache Möglichkeit, den Hauptspeicher in eine feste Anzahl von Partitionen einzuteilen, die nicht unbedingt dieselbe Größe haben müssen. Jede Partition kann maximal einen Prozess aufnehmen, der in ihr verbleibt, bis er beendet ist.

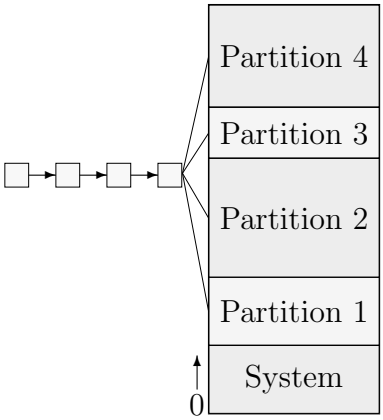
Die Partitionsgrößen werden beim Systemstart festgelegt oder können teilweise durch den Administrator zur Laufzeit korrigiert werden. Dieses Verfahren wurde von OS/360 verwendet (OS/MFT, *Multiprogramming with Fixed number of Tasks*).

Wenn ein neuer Prozess gestartet werden soll, ist es eine Dispatcher-Entscheidung, in welcher Partition er geladen wird. Normalerweise gehört er in die kleinste, die ihn aufnehmen kann.



Wenn die Prozesse *immer* in die für sie passendste Partition aufgenommen werden müssen, gibt es (explizit oder implizit) eine *Input Queue* wartender Prozesse für jede der Speicher-Partitionen (siehe links).

Wenn es dagegen erlaubt ist, einen Prozess in eine „zu große“ Partition aufzunehmen, wenn die passende gerade nicht frei ist, gibt es eine gemischte Input Queue (siehe rechts).



Im ersten Fall kann es passieren, dass kleine Prozesse lange liegengelassen werden, obwohl genügend Speicher frei wäre. Man kann dann mit *Alterungsprozessen* (siehe Seite 171 beim Scheduling) erreichen, dass sie irgendwann auf jeden Fall einmal eingeladen werden, auch in eine viel größere Partition.

Im zweiten Fall kann ein kleiner Prozess einen größeren blockieren und dabei viel Speicher unbenutzt lassen. Die Queue wird daher zumindest so organisiert, dass bei Freiwerden einer Partition der größte noch passende Prozess ausgewählt wird.

Dadurch, dass Prozesse ihre Partition nicht vollständig nutzen, liegt insgesamt eventuell viel Speicher brach. Er kann nicht für eine weitere Partition verwendet werden, da er über den Adressbereich verstreut ist. Dieser Erscheinung nennt man **interne Fragmentierung** des Speichers („intern“, da der verlorene Speicher im Zugriffsbereich der Prozesse liegt).

11.3.4 Variable Partitionen

Zu einem wesentlich flexibleren Schema kommt man, wenn die Aufteilung nicht beim Systemstart erfolgt und danach nicht mehr veränderlich ist:

- Frei gewordener Speicher kann vom Dispatcher beliebig an Prozesse vergeben werden, allerdings nur en-bloc.
- Ein Prozess befindet sich nach seinem Start bis zu seiner Terminierung immer vollständig im Hauptspeicher, und zwar an derselben Stelle. (Daher sind die Adressmodifikationen durch den Loader ausreichend.)

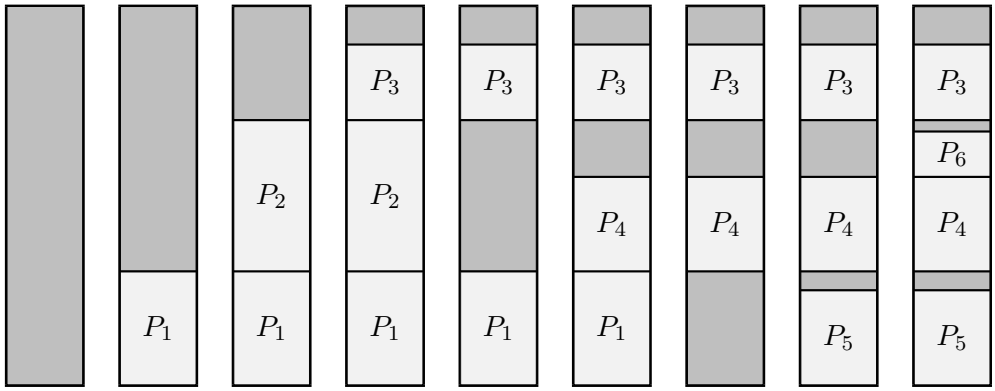
Ein Problem, mit dem solche Systeme (und *verwandte* Systeme wie Heap-Verwaltung, Platten-Speicherung etc.) zu kämpfen haben, ist die **externe Fragmentierung** des Speichers („extern“, da der Speicher außerhalb des Zugriffsbereichs aller Prozesse liegt).

Ein beendeter Prozess lässt ein *Loch* zurück, das in den seltensten Fällen wieder vollständig durch spätere Prozesse ausgenutzt werden kann. Löcher können nicht „gesammelt“ werden, da dazu das Verschieben von Prozessen notwendig wäre, was bei unserem einfachen Loader-Schema nicht möglich ist.

Das System führt eine **Liste** der momentan vorhandenen Löcher:

- Wenn ein Prozess zu Ende ist, wird das entstehende Loch in die Liste eingefügt. Wenn sich direkt vor und/oder hinter dem Bereich bereits Löcher befinden, werden sie ggf. zu einem großen Loch verschmolzen.
- Wenn ein neuer Prozess gestartet wird, wird ein passendes Loch gesucht. Üblicherweise ist es größer als der benötigte Bereich, so dass ein kleines Loch verbleibt und sich die Anzahl Löcher nicht verändert.
- Manchmal werden sehr kleine Löcher auch einfach dem direkt darunterliegenden Prozess mit zugeteilt, um den Verwaltungsaufwand zu verringern.

Unten ist die zeitliche Abfolge einiger solcher Speicherbelegungen durch neue Prozesse und Freigaben durch beendete Prozesse dargestellt. Die Löcher sind jeweils grau dargestellt.



Es kann so sehr leicht zu einer Situation kommen, in der ein Prozess momentan nicht geladen werden kann, obwohl insgesamt genügend freier Speicher vorliegt – allerdings nicht zusammenhängend. Durchschnittlich bleiben (nach einiger Laufzeit) 10 bis 30% des Speichers unbenutzbar, weil es in zu kleinen Blöcken vorliegt.

Bei der Auswahl des für einen zu ladenden Prozess *passenden* Lochs verfolgt man meist eine der folgenden simplen Strategien („*placement policies*“):

First-Fit (FF): Das erste genügend große Loch wird vergeben.

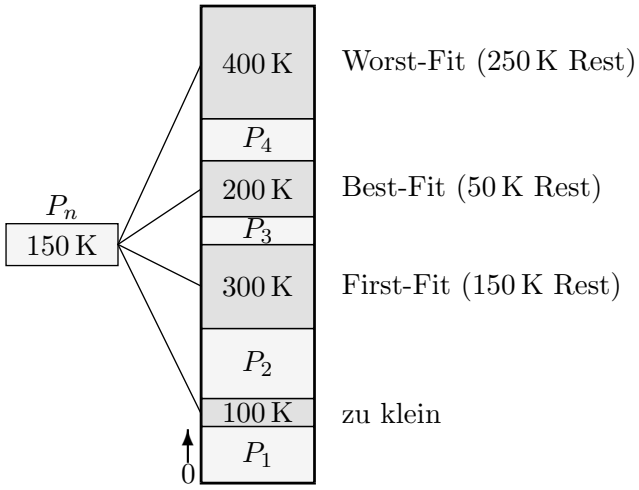
Rotating First-Fit (RFF): Wie FF, die Suche wird aber an der Stelle begonnen, an der die letzte Suche aufgehört hat.

Worst-Fit (WF): Das größte Loch wird vergeben.

Best-Fit (BF): Das kleinste noch passende Loch wird vergeben.

Wenn die Listenelemente nicht nach Größe sortiert werden, muss in den letzten beiden Fällen die gesamte Liste durchsucht werden. Dagegen kann die Suche bei FF und RFF beim ersten passenden Loch abgebrochen werden.

Rechts ist eine Situation dargestellt, in der ein neuer Prozess in einen fragmentierten Speicher eingepasst werden soll, und die Wahlen, die die verschiedenen Strategien treffen würden.



Es sind große Simulationen durchgeführt worden, um zu bestimmen, welche der Strategien für normale Systemsituationen am geeignetsten sind.

- Die Idee bei WF ist, dass das zurückbleibende Restloch noch möglichst groß bleibt. Es erweist sich aber als wesentlich schlechter als die anderen Möglichkeiten.
- FF erzeugt viele kleine Löcher am Anfang des Speichers. Wenn letztendlich Speicher in der Mitte vergeben wird, wird also zunächst sehr lange vorn in der Liste (erfolglos) gesucht.
- Aus diesem Grunde wurde RFF entwickelt, das die kleinen Löcher gleichmäßig über den Speicher verteilt.
- BF erzeugt oft sehr kleine Lücken, was sich aber nur selten wirklich nachteilig auf das Gesamtverhalten auswirkt.

Insgesamt verhalten sich RFF und BF in etwa gleich gut in der Praxis.

Es gibt nicht-lineare Verwaltungsstrukturen, beispielsweise die „Buddy-Systeme“ (auch in L*i*nux verwendet), die intern mit Bäumen arbeiten. Entsprechende Algorithmen arbeiten daher wesentlich schneller als die Listen-basierten, erzeugen aber in den Simulationen mehr unbenutzbare Speicherlücken.

Außerdem kann man die aktuelle Speichersituation auch in die Entscheidung des Schedulers eingehen lassen. Eventuell kann dieser bei Speichermangel für einen großen Prozess einen kleinen Prozess „vorlassen“, der (z.B. aufgrund niedrigerer Priorität) eigentlich noch nicht zum Laufen kommen würde. Der kleine Prozess kann dann natürlich (ähnlich wie auf Seite 288) den großen Prozess lange blockieren.

11.3.5 Basis-Register

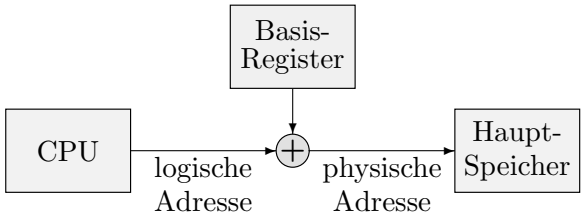
Die Hauptschwierigkeit in den oben angegebenen Schemata liegt darin, dass die Prozesse nicht mehr verschiebbar sind, wenn sie einmal im Speicher liegen. Ansonsten könnte man den Speicher „defragmentieren“, also die Prozesse so verschieben, dass möglichst wenig kleine Löcher verbleiben (s.u.). Im Extremfall bleibt ein einziges großes Loch, wofür aber fast immer *alle* Prozesse verschoben werden müssten. Wenn man mit weniger Verschiebungen auskommen will, kann man kleine Löcher akzeptieren.

Wenn die Berechnung der physischen Adressen vom Loader vorgenommen wird, sind dazu Tabellen im Objektcode notwendig. Dort sind die Stellen angegeben, an denen die Basisadresse des tatsächlichen Speicherbereichs aufaddiert werden muss. Diese Adressvergabe ist üblicherweise völlig statisch.

Eine Verschiebung von Prozessen in solchen Systemen ist denkbar, aber sehr ineffizient. Diese Tabellen müssten mit im Speicher abgelegt werden, und bei jeder Verschiebung müsste ein Loader-ähnlicher Mechanismus in Aktion treten, der vergleichsweise zeitaufwendig sein kann.

Eine Prozessverschiebung wird daher fast ausschließlich in Systemen mit spezieller Hardware vorgenommen. Die einfachste Möglichkeit dazu ist ein Register (im Prozessor oder in Zusatz-Hardware), das **Basis-Adressregister**.

In ihm ist die Basisadresse des Hauptspeicherbereichs des aktuellen Prozesses abgelegt. Die Maschinenbefehle sprechen den Speicher über *logische Adressen* (relativ, bezogen auf 0) an. Die Hardware *addiert* vor dem Speicherzugriff den Inhalt des Basisregisters.



Das Basis-Register wird beim Context-Switch zusammen mit den anderen Registerinhalten im Prozess-Kontrollblock abgespeichert und beim Weiterlaufen des Prozesses restauriert.

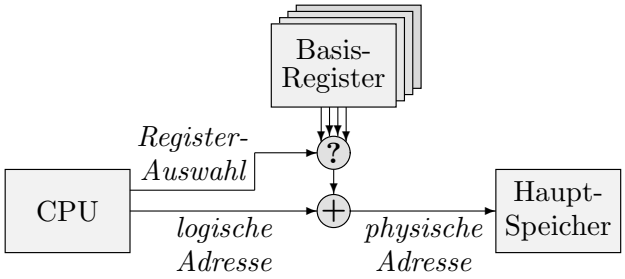
In solcher Hardware gibt es oft zusätzlich ein **Grenzregister** (Limit), das die *Größe* des zugreifbaren Bereichs angibt („*base-limit-addressing*“). Wenn ein Maschinenbefehl versucht, eine jenseits liegende Speicherzelle anzusprechen, wird dann eine Ausnahme ausgelöst.

Dadurch wird das Konzept des *Speicherschutzes* (*memory protection*) realisiert. Mehrere gleichzeitig im Speicher liegende Prozesse können so nicht auf die Daten der jeweils anderen zugreifen. Die Prozesse werden voneinander abgeschottet.

Diese Hardware bildet den logischen Adressraum $[0, l-1]$ ab auf den physischen Adressraum $[b, b+l-1]$ (b=Basis-Register, l=Limit-Register).

Es gibt viele Systeme mit **mehreren Basisregistern**, einem für jeden der logischen Speicherteile eines Prozesses (hauptsächlich also Code-Bereich, Daten-Bereich, Stack-Bereich).

Meist ist der Zusammenhang zwischen einem Maschinenbefehl und dem passenden Register implizit gegeben – ein „`push`“-Befehl spricht eben immer den Stack an und ist daher mit dem Stack-Basisregister gekoppelt, etc. Manchmal kann man auch explizit ein Basis-Register im Maschinenbefehl anwählen.



Es gibt einige Vorteile davon, die logische Aufteilung des Prozessspeichers auch physisch widerzuspiegeln:

- Es wird nicht so viel zusammenhängender Speicher benötigt wie bei einem großen Block. Große Prozesse sind also leichter in fragmentiertem Speicher unterzubringen.

- Kleinere Teile sind durch das System (z.B. beim Verschieben) leichter und schneller handhabbar.
- Wenn der Stack zu groß wird („*Stack Overflow*“, zu viele verschachtelte Unterprogramm-Aufrufe), überschreitet er die durch das Stack-Limitregister vorgegebene Grenze, und es wird eine Ausnahme ausgelöst. Die passende Routine kann dann *automatisch* einen *größeren* Stack-Bereich anlegen, den Stack kopieren und das Limit-Register entsprechend setzen. Das Programm merkt davon nichts und kann davon ausgehen, mit einem „unbegrenzt“ großen Stack zu arbeiten.
- Bei noch speziellerer Hardware kann das Code-Segment außerdem schreibgeschützt werden (bei einem Schreibversuch wird ebenfalls eine Ausnahme ausgelöst), um noch höhere Sicherheit zu erreichen.

11.3.6 Defragmentierung

Wenn eine Verschiebung der Prozesse im Speicher (auf welche Weise auch immer) möglich ist, kann der zerstückelte Speicher defragmentiert (oder kompaktifiziert) werden. Bei Basis-Adressregistern braucht beispielsweise lediglich der Speicherinhalt verschoben und das Register (bzw. dessen Spiegelung im PCB) auf den neuen Anfang gesetzt zu werden.

Die Defragmentierung kann stattfinden:

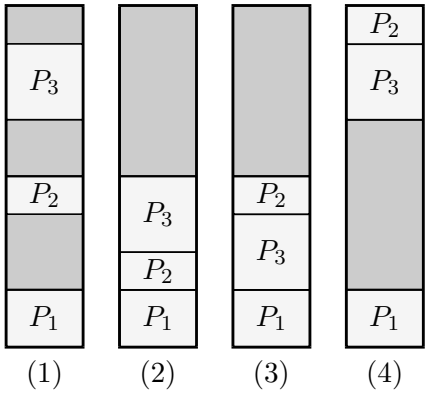
- immer, wenn ein Prozess beendet ist,
- wenn es für einen neuen Prozess nicht genügend zusammenhängenden Speicher gibt,
- in festgelegten (größeren) Zeitabständen,
- durch Eingriff des Administrators.

Das Herumkopieren vieler Prozesse ist natürlich zeitaufwendig. Auf manchen Systemen wird sogar der Umweg über den Sekundärspeicher gegangen (alle oder viele Prozesse werden ausgelagert und an neuen Positionen wieder eingelagert). Dann wird es nicht sehr oft durchgeführt, sondern nur im Notfall.

Bei großem Hauptspeicher und relativ häufiger Defragmentierung ist die Konzeption des Algorithmus, der die Art der Verschiebung bestimmt, sehr wichtig. Die Hauptschwierigkeit liegt im Auftreten der ganz unterschiedlichen Blockgrößen.

- Eine einfache Strategie verschiebt einfach alle Prozesse in der aktuellen Reihenfolge nach unten (bzw. nach oben) im Speicher, sodass ein einziges großes Loch am oberen (bzw. unteren) Ende entsteht. Davon werden allerdings *alle* Prozesse (außer dem untersten) berührt!
- Kompliziertere Strategien versuchen, eine neue Prozess-Reihenfolge zu bestimmen, in die die aktuelle Reihenfolge mit möglichst wenig Aufwand überführt werden kann. Eventuell ist es dabei sinnvoller, das Loch nicht an das Ende des Speichers, sondern in die *Mitte* zu

legen oder *mehrere* Löcher zu akzeptieren. Bei sehr vielen Prozessen ist es nicht möglich, *alle* Reihenfolgen zu überprüfen. Daher ist man mit einer nicht ganz optimalen, aber guten Lösung zufrieden.



(1) Hier ist die ursprüngliche Situation dargestellt.

(2) Der einfachste Algorithmus behält die Prozess-Reihenfolge bei und verschiebt hier P_2 und P_3 nach unten.

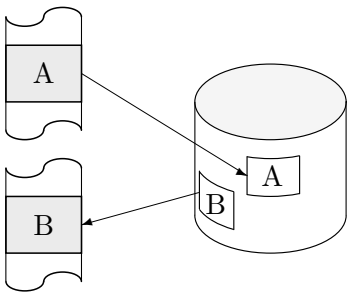
(3) Ein Algorithmus, der erkennt, dass P_3 gerade in das untere Loch passt, braucht nur P_3 zu verschieben.

(4) Einer, der erkennt, dass P_2 in das obere Loch passt, und der ein Loch in der Mitte akzeptiert, braucht nur das *kleinere* P_2 zu verschieben.

11.4 Swapping

Bei den bisher besprochenen Strategien verbleibt ein Prozess nach seinem Start im Hauptspeicher, bis er beendet ist. Die Anzahl von Prozessen, die gleichzeitig im System laufen, ist daher durch die Hauptspeichergröße beschränkt.

Es gibt zwei wesentliche Methoden, den Hauptspeicher durch Ausweichen auf Sekundärspeicher virtuell zu vergrößern: *Swapping* und *Paging*.



Der Begriff „Swapping“ wird sehr häufig falsch verwendet, nämlich für *beliebige* Auslagerungs-Mechanismen. Eigentlich steht er nur für das Auslagern eines *kompletten Prozesses* aus dem Hauptspeicher auf den Sekundärspeicher und wird heutzutage praktisch nicht mehr angewandt. Das Auslagern wird als „swap-out“, das Einlagern als „swap-in“ bezeichnet.

Die Anzahl der gleichzeitig möglichen Prozesse wird dadurch nicht mehr durch die Größe des Hauptspeichers, sondern durch die des für das Swapping zur Verfügung gestellten Sekundärspeichers begrenzt.

- Diese Technik wurde zuerst auf Systemen angewandt, bei denen nur ein Prozess gleichzeitig im Speicher existieren konnte. Bei einem Prozesswechsel wurde der laufende Prozess ausgelagert und ein wartender von der Platte zurückgeladen. Es fand also ein Prozess-*Austausch* statt, und daher der Name *swapping*.

- Gute Kandidaten für das Auslagern sind solche, die momentan durch Warten (z.B. auf I/O oder einen anderen Prozess) blockiert sind.

- Bei besonders einfachen solchen Systemen konnte der aktive Prozess den gesamten Hauptspeicher für seine Daten verwenden, ohne das System genauer darüber zu informieren. Dann musste also der gesamte Hauptspeicher ausgelagert werden, nicht nur der tatsächlich vom Prozess verwendete Teil! Wenn benutzter Speicher mit Systemaufrufen zum Reservieren und Freigeben verwaltet wird, kann das Auslagern effektiver gestaltet werden.

- Der Großteil des Memory Managements in Systemen mit vollständigem Swapping liegt in den Händen des Dispatchers. Wenn ein Prozess vom Scheduler aktiviert werden soll, der gerade ausgelagert ist, liest der Dispatcher ihn wieder zurück. Wenn nicht genügend Hauptspeicher dafür frei sein sollte, übernimmt er außerdem das Auslagern eines oder mehrerer anderer Prozesse. Erst danach findet das Restaurieren von Registerinhalten und die CPU-Übergabe statt.

- Die Zeit für einen Context-Switch, der mit einem Ein- und ggf. Auslagern verbunden ist, ist natürlich sehr groß. Entsprechend wird auch das Zeitquantum der Scheduler-Algorithmen vergleichsweise hoch angesetzt.

- Das Auslagern kann zwischenzeitlich durch I/O-Operationen verzögert werden. Wenn diese Operationen noch auf Speicherteile des Prozesses zugreifen, muss zunächst auf ihre Beendigung gewartet werden. Üblicherweise werden daher alle solche Bereiche in Systempuffer umkopiert.

- Um das Swapping möglichst effizient zu gestalten, werden die ausgelagerten Daten meist nicht in Form regulärer Dateien abgespeichert. Stattdessen wird ein spezieller Bereich auf der Platte zur Verfügung gestellt („*swap space*“), der direkt ohne den Umweg über ein Dateisystem verwaltet wird.

  Aus Geschwindigkeitsgründen wird meist versucht, die Daten eines Prozesses hintereinander in diesem Swap-Bereich abzulegen. Dann ensteht allerdings schnell wieder das Problem der Fragmentierung. Es ist allerdings praktisch unmöglich, hier eine Kompaktifizierung vorzunehmen, während derer das Swapping unmöglich wäre.

11.5 Paging

Zwei wesentliche Hindernisse haben die besprochenen Methoden bereits aus dem Wege geräumt:

- Durch **Basis-Register-Adressierung** sind zwar Prozesse im Speicher nicht mehr starr an eine Position gebunden. Der immer wieder notwendige Aufwand des *Kompaktifizierens* des Speichers ist jedoch beträchtlich. Besonders die unterschiedlichen Partitionsgrößen machen die optimale Speichernutzung schwierig.

  Als Bonus erhält man (per Limit-Register) die Abschottung der verschiedenen Speicherbereiche voneinander.

- Mit **Swapping** kann das System zwar mehr Prozesse verarbeiten, als sein rein physischer Hauptspeicher es (ohne Auslagerung) zulassen würde. Es ist so aber nicht möglich, dass ein Prozess (logisch) *mehr* Hauptspeicher erhält, als *physisch* überhaupt vorhanden ist.

  Wenn man eine große Aufgabe in mehrere Teilprozesse zerlegt (von denen immer nur ein Teil im Speicher zu sein braucht), kann es zu Deadlocks kommen, wenn ein unbedingt benötigter Teil nicht in den Speicher passt, aber der wartende Teil im Speicher verbleibt.

  Außerdem ist es nicht ökonomisch, ganze Prozesse ein- und auszulagern. Viele Prozesse verbringen die meiste Zeit in einem kleinen Teil ihres Codes oder greifen nur auf einen Teil ihres Datenbereichs zu. Dieses Verhalten vorherzusagen, ist allerdings nicht möglich.

Beim Paging werden daher der logische, der physische Adressraum im Hauptspeicher und der Auslagerungsbereich auf dem Sekundärspeicher in **Blöcke** einer global festen Größe aufgeteilt. Spezielle **Hardware** bildet (über „Seitentabellen“) logische Blöcke auf physische Blöcke ab (analog zu den Basis-Adressregistern). Das Verteilen und die Verwaltung der Seiten wird von **Software** übernommen, die per Exception von der Hardware aufgerufen wird.

Durch Paging erhält man folgende wichtige Vorteile:

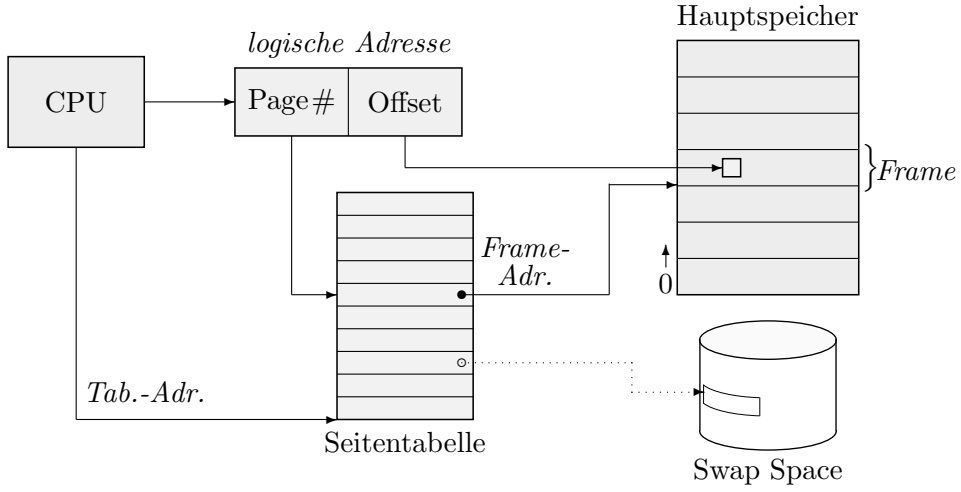
- Es braucht nicht immer der gesamte Adressbereich eines Prozesses im Hauptspeicher zu liegen, nur der aktuell tatsächlich verwendete. Mehr Prozesse finden gleichzeitig im Hauptspeicher Platz. Es findet weniger I/O beim Ein- und Auslagern statt.
- Prozessspeicher braucht nicht mehr zusammenhängend zu sein. Dadurch gibt es keinerlei *externe* Fragmentierung mehr. Bereiche eines Prozesses können problemlos wild im Haupt- und Sekundärspeicher verteilt sein. (Es gibt noch geringe *interne* Fragmentierung dadurch, dass die jeweils „letzte“ Seite eines Prozesses eventuell nicht vollständig genutzt wird.)
- Es braucht nicht mehr im Speicher verschoben zu werden. Wenn sich (nach einem Auslagern) die Adresse eines Blocks geändert hat, braucht nur ein Tabelleneintrag geändert zu werden.

11.5.1 Hardware für Paging

Unter **einfachem Paging** versteht man den grundlegenden Mechanismus, noch nicht kombiniert mit Auslagerung auf Sekundärspeicher. **Virtuelles Paging** stellt mit Auslagerungs-Mechanismen dann echten virtuellen Speicher zur Verfügung.

Die jetzt besprochene *Hardware* ermöglicht das *einfache* Paging. Das virtuelle Paging ist dagegen ein Software-Mechanismus des Betriebssytems.

Die Hardware ist etwas komplexer als bei der einfachen Basisregister-Adressierung. Mittlerweile findet sie innerhalb der meisten CPUs Platz. Früher war sie meist in Form einer *externen* Memory-Management-Unit realisiert, die zwischen CPU und Hauptspeicher lag. *Konzeptionell* wollen wir in diesem Abschnitt die MMU vom Rest der CPU trennen.



Die Hardware gibt (fast immer) die Größe der Blöcke fest vor. Meist handelt es sich um eine Zweierpotenz im Bereich von 512 bis 8192 (Bytes), was die Hardware-Umsetzung vereinfacht (s.u.). Bei der 80x86-Familie ist die Blockgröße 4 KByte.

Bei der Wahl der Blockgröße (beim Prozessor-Design) spielen ähnliche Faktoren wie bei der Blockgröße eines Dateisystems eine Rolle. Bei großen Blöcken werden die benötigten Tabellen kleiner, aber es wird mehr Verschnitt und I/O erzeugt. Bei kleineren Blöcken gibt es weniger Verschnitt, dafür sind die Tabellen groß und langsamer – beispielsweise können sie schwieriger im Cache gehalten werden. (Bei einer Größe von 1 Byte wäre der Speicher vollständig virtualisiert – und die Tabellen würden mehr Platz verschlingen als der eigentlich genutzte Speicher.)

Logische und physische Blöcke haben unterschiedliche Namen erhalten:

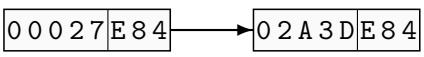
Pages sind **logische** Blöcke, die die CPU adressiert.

Frames (oder Page Frames, Kacheln) sind die **physischen** Blöcke im Hauptspeicher. Sie werden eben als *Rahmen* verstanden, die den tatsächlichen Inhalt der Seiten *aufnehmen* können.

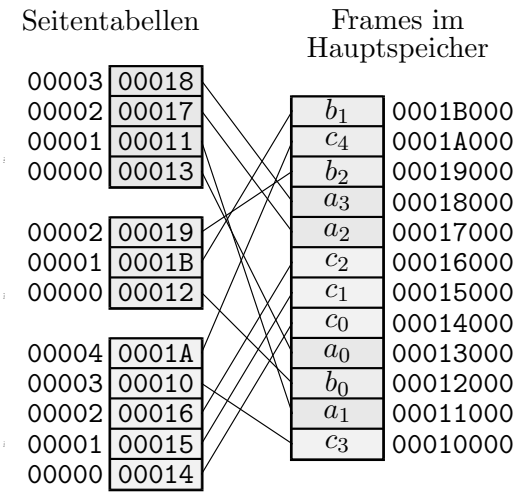
Dadurch, dass die Seitengröße (fast überall) eine **Zweierpotenz** ist, entspricht die Umsetzung von logischer in physische Adresse lediglich einem *Austausch der oberen Bits*: Die logische Seitennummer wird von der Framenummer überschrieben. Außerdem brauchen in der Seitentabelle nur die oberen Bits der Framenummer gespeichert zu werden. Die freiwerdenden Bits können für andere Zwecke verwendet werden (z.B. für Flags, ob die Seite im Speicher oder auf Platte liegt, etc.).

Beispiel: Beim 80x86 ist die Seitengröße 4096=2^{12} Bytes. Die untersten 12 Bits einer logischen Adresse stellen also den Offset dar, die oberen 20 die Seitennummer. In der Seitentabelle sind also 20-Bit-Zahlen abgespeichert.

Bei der Umsetzung der Adresse $(00027E84)_{16}$ (hexadezimal) erhalten wir die Seitennummer $(00027)_{16}$ und den Offset $(E84)_{16}$. Angenommen, an der Stelle $(27)_{16}$=39 in der Seitentabelle stehe $(02A3D)_{16}$. Dann erhält man die physische Adresse durch den Austausch der obersten 20 Bits, also als $(02A3DE84)_{16}$:



Die Seiten verschiedener Prozesse können natürlich bunt gemischt in den Hauptspeicher-Frames untergebracht werden. Rechts ist eine Situation mit drei Prozessen mit jeweils eigener Seitentabelle dargestellt. Die Seitengröße ist wieder mit 4096 Bytes angenommen (Frame-Nummern mit 20 Bits).



11.5.2 Die Seitentabelle

Jeder Speicherzugriff geht bei Paging durch die Umsetzung mit der Seitentabelle. Daher ist eine effiziente Implementation sehr wünschenswert.

- Im einfachsten Fall wird die Seitentabelle als ein Satz von **Prozessor-Registern** aufgebaut und ist entsprechend schnell. Leider ist dies überhaupt nur bei sehr kleinen Tabellen (bis einige hundert Einträge) machbar, bei heutigen Speichergrößen (und sinnvoll kleinen Seiten) jedenfalls nicht mehr.

  Wenn jeder Prozess eine eigene Seitentabelle hat, wird außerdem der Context-Switch stark verlangsamt, da nun auch die Seitentabelle gesichert werden muss.

- Daher liegt die Seitentabelle meist (wie im Bild oben) **im Hauptspeicher**, in einem Betriebssystem-Bereich, der nicht für Frames zur Verfügung steht. Üblicherweise besitzt jeder Prozess eine eigene Seitentabelle, wodurch die Adress-Umsetzung besonders einfach wird.

  Ein Prozessor-Register (**PTBR**, *Page Table Base Register*) hält die Anfangsadresse der gerade aktuellen Seitentabelle. Bei einem Context-Switch braucht nur der Wert dieses Registers gesichert zu werden. Ein weiteres Register (**PTLR**, *Page Table Length Register*) nimmt die Länge der Tabelle auf. So können Zugriffe außerhalb des gültigen Adressbereichs abgefangen werden.

  Es ergibt sich also ein Speicherzugriff nach folgendem Schema:

  memory**[**memory**[**PTBR**+**pagenumber**]+**offset**]**

  (Zugriffs*längen* sind hier unberücksichtigt).

- Ohne spezielle weitere Maßnahmen würde nun allerdings jeder einzelne Speicherzugriff *doppelt so langsam*, da jedesmal zunächst der Eintrag in der Seitentabelle gelesen werden muss! In den gängigen Systemen ist aber immer genügend **Cache** zwischen CPU und Hauptspeicher gelegt, sodass sich keine merkliche Verlangsamung ergibt.

- Bei mancher Hardware gibt es einen speziellen Cache nur für Einträge von Seitentabellen, meist innerhalb der CPU (bis typischerweise 32 Einträge).

  Da bei der Abarbeitung eines Programms sehr oft hintereinander auf dieselben wenigen Seiten (für Code und Daten) zugegriffen wird, ist dieser Cache sehr erfolgreich. Er wird **Translation Lookaside Buffer** (TLB) genannt; entsprechend sind die Begriffe **TLB Hit** und **TLB Miss** zu verstehen.

  Wenn im TLB Teile mehrerer Seitentabellen erlaubt sind (von verschiedenen Prozessen), reicht es nicht, als Schlüssel die logische Seitennummer zu verwenden (die Prozess-Identität muss mit eingehen). Alternativ ist nur eine Seitentabelle erlaubt, und alle TLB-Einträge müssen bei einem Context Switch für ungültig erklärt werden.

  Die **Hit Ratio** ist der (durchschnittliche) Prozentsatz erfolgreicher Zugriffe auf den TLB, also der Quotient $\alpha = \mathrm{Hit}/(\mathrm{Hit} + \mathrm{Miss})$. Ein echter Speicherzugriff möge eine Zeiteinheit benötigen, ein TLB-Zugriff nur $\varepsilon < 1$ Einheiten. Dann ergibt sich die effektive (durchschnittliche) Zugriffszeit t_{av} als:

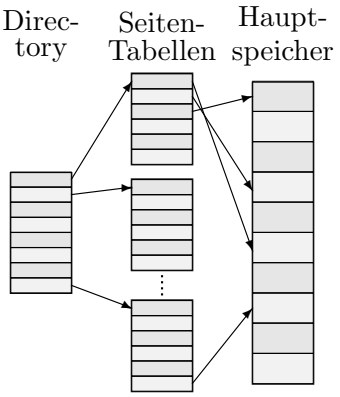
$$t_{\mathrm{av}} = \underbrace{\varepsilon \cdot \alpha}_{Hit} + \underbrace{(1+\varepsilon)\cdot(1-\alpha)}_{Miss} + 1 \;=\; 2 + \varepsilon - \alpha.$$

11.5.3 Multilevel-Paging

Um Speicher bei großen Seitentabellen einzusparen, implementieren manche Systeme einen Paging-Mechanismus in mehreren Stufen (auch die 80x86-Familie, genaueres später).

Im zweistufigen Fall wird die uns geläufige Seitentabelle in kleinere Tabellen (fester Größe) aufgeteilt. Es wird eine äußere Tabelle („**Directory**“, Page-Directory) eingeführt, deren Einträge Zeiger auf die Teil-Seitentabellen sind.

Es brauchen nicht von vornherein alle Seitentabellen wirklich angelegt zu werden. Die Einträge im Directory bleiben dann leer (0). Erst, wenn der Prozess tatsächlich auf den entsprechenden Teil des logischen Adressraums zugreift, müssen die Lücken gefüllt werden (und bleiben von da an gefüllt).



Die logische Adresse ist entsprechend mehrfach unterteilt nach dem Schema „Directory/Seite/Offset“. Systeme mit noch mehr Stufen sind analog zu verstehen.

Ähnliche Mehrstufen-Strukturen mit Lücken werden wir übrigens noch beim UNIX-Dateisystem kennenlernen.

11.5.4 Demand Paging

Bisher haben wir nur gesehen, wie der Mechanismus des einfachen Pagings die Speicherabbildung vornimmt und dadurch Fragmentierung eliminiert. Um aber echten virtuellen Speicher über die Hauptspeichergröße hinaus zu erhalten, müssen Swapping-Konzepte hinzugenommen werden (virtuelles Paging).

Es muss ein Mechanismus eingebaut werden, der Seiten aus dem Hauptspeicher auf Platte aus- und wieder einlagern kann. Eine Strategie muss formuliert werden, *wann welche* Seiten in den Hauptspeicher geladen werden, und wann andere Seiten aus dem Hauptspeicher entfernt werden.

Das übliche Verfahren heißt **Demand Paging**, was andeutet, dass Seiten genau dann geladen werden, wenn sie (durch einen CPU-Zugriff) *benötigt* werden.

11.5.4.1 Seiten-Flags

Zur Verwaltung werden zusätzliche **Flags** für jede Seite benötigt, die in den freien Bits in der Seitentabelle untergebracht werden.

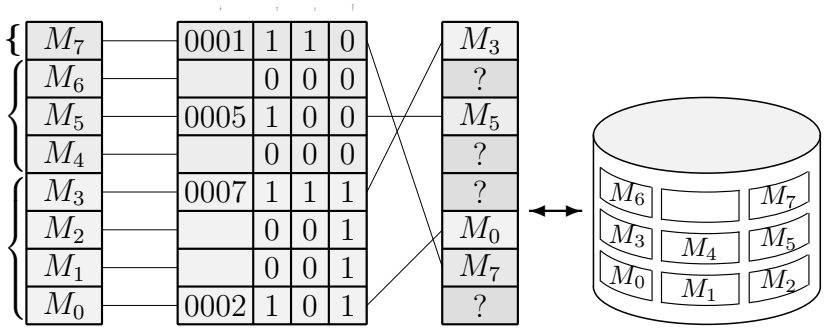
Die wichtigsten Flags sind:

valid (oder **present**): Die Seite steht im Hauptspeicher. Falls nicht (valid=0, „invalid“), kann sie z.B. auf der Platte liegen. Dann kann die Frame-Nummer dieses Eintrags für die Position auf der Platte verwendet werden.

dirty (oder **modified**): Die Seite steht im Hauptspeicher und wurde dort verändert (meist nicht möglich bei Code), analog zu dirty bei Cache-Einträgen. Wenn die Seite gegen eine andere ausgetauscht werden soll, muss ihr Inhalt also im Swap-Space gesichert werden.

code/data : Hier wird die Nutzungsart des Speichers angegeben, da Seiten mit Code oder Daten in bestimmten Zusammenhängen unterschiedlich behandelt werden.

Ggf. gibt es weitere Flags wie „read-only“, um Seiten als nur-lesbar zu kennzeichnen (typisch bei Maschinencode-Seiten).



11.5.4.2 Page Faults

Wenn ein **neuer Prozess** geladen wird, wird zunächst die vollständige Seitentabelle (mit dem gesamten Adressbereich für Code, Daten und Stack) angelegt. Die Einträge werden aber alle als nicht gültig gekennzeichnet.

Es wird danach nicht der gesamte Code eingelesen. Lediglich die ersten paar Seiten (die mit dem Einsprungpunkt in den Code) werden angelegt, der Code hineingeladen und die Einträge in der Seitentabelle geschrieben und gültig gemacht. Seiten zu Datensegmenten werden meist nicht erzeugt, meist aber eine Seite für den Beginn des Stacks. Danach wird der Prozess gestartet.

Es passiert schnell, dass die CPU auf eine (logische) Adresse zugreift, deren physische Entsprechung noch nicht im Hauptspeicher liegt – wenn die ersten Code-Seiten verlassen werden oder ein Daten-Zugriff stattfindet. Die CPU bemerkt dies durch das nicht gesetzte Valid-Flag in der Seitentabelle. Sie löst eine Exception aus. Diese Situation bezeichnet man als **Page Fault** (analog zu einem Cache Miss).

Durch die Exception tritt das Betriebssystem in Aktion. Seine Aufgabe ist es, die angeforderte Seite gültig zu machen.

Zunächst wird im PCB des aktuellen Prozesses nachgeschaut, ob der Speicherzugriff überhaupt **gültig** war. Ungültige Speicherzugriffe sind solche über das Ende des Adressraums hinaus oder auf Lücken im Adressraum. In solchen Fällen wird der Prozess mit einen „segmentation fault“ o.ä. abgebrochen.

Ansonsten muss ein **leerer Frame** gefunden werden (üblicherweise werden leere Frames in einer Liste geführt). Wenn es keine leeren Frames gibt, muss eine andere Seite ausgelagert werden (Strategien dazu weiter unten). Der neue Frame wird schließlich wie folgt gefüllt:

- Wenn es sich um einen **Code**-Teil handelt (der ja nie verändert wird), kann die Seite aus der Datei mit dem ausführbaren Programm nachgeladen werden.

- Wenn es sich um **initialisierte Daten** handelt, die *noch nicht überschrieben* wurden, kann die Seite ebenfalls aus der Datei nachgeladen werden.
- Wenn es sich um **nicht initialisierte Daten** (oder Stack) handelt, die *noch nicht angesprochen* wurden, kann einfach eine neue Seite angelegt werden. Eventuell wird sie mit Nullen gefüllt.
- Wenn es sich um **Daten** handelt, die bereits einmal im Speicher gestanden haben und *überschrieben* wurden, stehen sie jetzt nicht mehr im Speicher, weil sie ausgelagert wurden. Entsprechend müssen sie aus dem **Swap-Bereich** restauriert werden.

Wenn eine Seite von Platte einzulesen ist, stößt das Betriebssystem den passenden I/O-Vorgang an und *blockiert* den Prozess. Ein anderer Prozess übernimmt die CPU-Kontrolle. Wenn der I/O beendet ist, wird der Prozess in den *Wartezustand* versetzt und erhält bei einem der nächsten Prozesswechsel die CPU. Er läuft bei dem Befehl weiter, der die Exception (den Page Fault) auslöste.

11.5.4.3 Speicherschutz

Speicherschutz ist die Zugriffs-Beschränkung eines Prozesses auf dem ihm tatsächlich zugewiesenen Adressraum. Zusätzlich können Speicherteile als nur lesbar markiert werden (Maschinencode). Ein fehlerhafter Zugriff löst eine Exception aus, z.B. „segmentation fault“, „addressing violation“ o.ä.

Eine einfache Form von Speicherschutz hatten wir bereits in Form der Grenzregister (Limits) bei der Basisregister-Adressierung (Seite 291) kennengelernt. Damals war allerdings der Adressraum eines Prozesses zusammenhängend (und hatte zu jedem Zeitpunkt eine komplette physische Entsprechung).

Beim Paging braucht der Adressraum nicht unbedingt zusammenhängend zu sein; der Datenteil muss beispielsweise nicht direkt hinter dem Codeteil liegen. In der Seitentabelle können einfach einige Einträge immer leer bleiben. Es kann dann drei Arten von fehlerhaften Zugriffen geben:

- Wenn eine logische Adresse verwendet wird, die außerhalb der „Hülle“ des physisch realisierten Adressraums liegt, wird bei der Adressumsetzung hinter das Ende der Seitentabelle zugegriffen. Die Hardware erkennt das durch Vergleich mit dem Längenregister PTLR und löst eine Exception aus.
- Wenn auf einen Seitentabellen-Eintrag zugegriffen wird, der keine physische Entsprechung (im Speicher oder auf Platte) hat, kann die Paging-Routine des Betriebssystems eine Exception auslösen.
- Es könnte versucht werden, in eine nur-lesbare Seite zu schreiben. Ein Flag in der Seitentabelle codiert einen Nur-Lese-Status. Die Hardware kann solche Versuche abfangen und eine Exception auslösen.

11.5.4.4 Verhalten von Demand Paging

Demand Paging funktioniert erstaunlich gut, obwohl ein Page Fault eine ziemliche Wartezeit verursacht. Messungen zufolge löst nur etwa jede millionste Anweisung einen Page Fault aus.

Das liegt am „lokalen“ Verhalten typischer Programme (die in Hochsprachen wie `C` oder `PASCAL` geschrieben wurden). Ein Sprung in ein Unterprogramm verursacht oft einen Page Fault. Danach hält sich der Prozess aber einige Zeit in diesem Unterprogramm und damit auf der neuen Seite (oder auf einigen der nachfolgenden) auf.

Dieses Verhalten nennt man **Referenzlokalität**. Es betrifft nicht nur den *Code*, sondern auch die *lokalen Variablen*, die ja nah beieinander auf dem Stack liegen. Daher ist es nicht nur guter Programmierstil, sondern auch effizienter, alle Variablen, die nur lokal benutzt werden, auch nur lokal zu deklarieren!

Es liegt dabei am Betriebssystem, im Hinblick auf diese Lokalität eventuell etwas mehr als nur die direkt angesprochene Seite einzulesen. Wenn zu viele weitere Seiten geladen werden, werden eventuell zusätzliche Page Faults bei anderen Prozessen ausgelöst, deren Seiten dafür ausgelagert werden mussten.

11.6 Auslagerung

Bei allen Verfahren für virtuellen Speicher muss ein vernünftiges Verfahren gefunden werden, das festlegt, welche momentan residenten Speicherteile auf den Sekundärspeicher ausgelagert werden. Die Betrachtungen sind immer ähnlich; wir werden uns hier allerdings am Demand Paging orientieren.

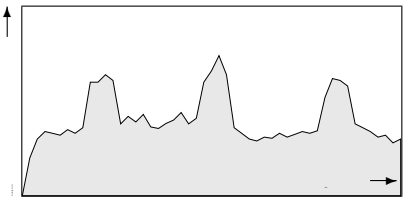
Wenn eine noch nicht residente Seite angesprochen wird, *muss* sie in einen freien Frame gelesen werden. Wenn es keinen freien Frame gibt, wäre es angesichts des Hintergrund-Charakters des Paging-Mechanismus keine gute Idee, den Prozess abzubrechen. Möglich wäre allerdings noch ein Auslagern des Prozesses und ein Suspendieren, bis wieder genügend Speicher frei ist.

- Üblicherweise wird aber nur eine andere, gerade (oder schon längere Zeit) nicht benutzte Seite auf den Sekundärspeicher ausgelagert (in Auslagerungs-Datei oder Swap Space).
- Wenn ein „Opfer“ gefunden ist, wird in der Seitentabelle das Present-Flag dieser Seite gelöscht, und in irgendeiner Form eingetragen (z.B. anstelle der physischen Frame-Adresse), wo die Seite auf dem Sekundärspeicher zu finden ist.
- Der Name „Swapping“ trifft hier also wieder zu, da eine neue Seite gegen eine alte *ausgetauscht* wird. Der Zeitbedarf gegenüber einem einfachen Page Fault wird verdoppelt.
- Nicht in jedem Fall muss aber das Opfer überhaupt abgespeichert werden. Wie wir oben schon gesehen haben, können *Code-Seiten* später genauso gut direkt aus der Objektcode-Datei restauriert werden. Sie werden daher meist nicht ausgelagert. Ebenso brauchen Daten-Seiten nur gesichert zu werden, wenn sie seit dem Einlagern *geändert* worden sind. Aus diesem Grunde ist das Dirty-Flag im Page-Table-Eintrag eingeführt worden, das bei jedem Schreibzugriff von der Hardware automatisch gesetzt wird. Ein Read-Only-Flag kann denselben Zweck erfüllen.

Auf einigen Systemen (bei den meisten UNIXen) werden von Zeit zu Zeit Seiten ausgelagert, obwohl noch kein Platzmangel herrscht. Es wird so versucht, immer eine kleine Anzahl von Seiten freizuhalten (z.B. 20 bis 40) – für eine Situation, in der schnell viele neue Seiten bereitstehen sollten.

11.6.1 Working Set

Man bezeichnet als „*Working Set*“ die Menge an Seiten, die ein Prozess innerhalb eines vorgegebenen kleinen Zeitraums tatsächlich anspricht. Wenn er sich lokal verhält (in einem oder in wenigen Unterprogrammen), ist das Working Set klein. Wenn er die Lokalität wechselt, ist das Working Set kurzfristig relativ groß (siehe Skizze rechts).



Gute Demand-Paging-Verfahren versuchen jeweils, das aktuelle Working Set zu erraten und zugehörige Seiten im Speicher zu belassen, wenn irgend möglich. Als einfachstes kann man feststellen, dass in einer kurzen Zeit mehrere Page Faults zur selben Seite ausgelöst wurden. Diese Seite sollte danach, wenn möglich, im Speicher gehalten werden.

Wenn ein Paging-Verfahren schlecht arbeitet, schätzt es das Prozess-Verhalten falsch ein. Es lagert Seiten aus, die kurz danach wieder benötigt werden, oder zu viele nicht benötigte Seiten ein. Dann kann es passieren, dass ein großer Prozentsatz der CPU-Zeit durch Swapping-I/O verbraucht wird, was man auch leicht an der hohen Plattenaktivität hören kann. Dieses Verhalten nennt man **Thrashing** (wie bei schlechten Scheduling-Verfahren).

11.6.2 Auslagerungs-Strategien

Das beste Verfahren ist das, das langfristig durchschnittlich die wenigsten Page Faults erzeugt. Am besten sollte immer die Seite verdrängt werden, die in Zukunft am längsten unbenutzt bleiben wird (nach Belady, 1966). Dieses Verhalten der Prozesse kann natürlich in der Praxis nur erraten werden. Entsprechende Annäherungen heißen **Seitenverdrängungs-Strategien** oder **Page Replacement Policies**.

Das Verhalten der unterschiedlichen Algorithmen kann getestet werden, indem man eine große Zahl typischer Speicherzugriffe simuliert, etwa mit einem Zufallsgenerator.

Man darf dem Zufallsgenerator nicht völlig freie Hand lassen. Die Referenzlokalität muss bei den erzeugten Daten berücksichtigt werden.

Das rechtsstehende Programm in einer Pseudo-Maschinensprache erzeugt bei einer Page-Größe von 100 Einheiten folgende Zugriffsfolge:

Seite 0,2,0,3,0,0,2,1,1,4,1,1,2.

Zugriffsfolgen in dieser Notation nennt man **Referenzstrings**.

```
 96 MOV R0,200
 97 CMP R0,300
 98 JNE 102
 99 MOV R1,201
100 ADD R1,R0
101 MOV 400,R1
102 DIV R0,#2
103 MOV 201,R0
```

Die Häufigkeit von Page Faults hängt natürlich von der Gesamt-Anzahl der Frames ab. Mit 5 Frames erzeugt die obige Folge 5 Page Faults, jeweils beim ersten Zugriff auf jede der 5 vorkommenden Seiten. Mit einem Frame wird bei jedem Zugriff ein Page Fault erzeugt – oft also zwei pro Maschinenbefehl!

Professionelle Test-Verfahren benötigen zunächst einen Algorithmus, der „glaubhafte“ Zufallsdaten dieser Art produziert. Der Einfachheit halber legen wir uns hier auf eine kurze, feste Folge von Zugriffen fest (direkt aufeinanderfolgende Zugriffe auf dieselbe Seite weggelassen):

0,2,1,5,6,1,2,5,0,3,0,5,2,1,5,4,1,5,2,1,5,2,0,5,3,0,5,4,1,5

Außerdem nehmen wir immer an, wir hätten drei Frames (für diese maximal sieben angesprochenen Seiten) zur Verfügung.

Wir benutzen jeweils eine kleine C++-Klasse, um die Algorithmen zu implementieren und das Verhalten beobachten zu können. Alle sind von der folgenden abstrakten Oberklasse abgeleitet:

```
class page_replacer
 {
   private:
      int filled;
      virtual int find_victim() = 0;
   protected:
      int swaps;
      const int frames;
      int *pagetable;
      void next_page(int p);
   public:
      page_replacer(int f)
       : swaps(0), frames(f), pagetable(new int[f]), filled(0) { }
      ~page_replacer() { delete[] pagetable; }
      virtual void test_string(const int *);
      virtual void access_entry(int) { };
      virtual void in_entry(int) { };
 };

void page_replacer::test_string(const int *s)
 {
   int p;
   while ((p=*s++)>=0) next_page(p);
   cout << swaps << " page swaps\n\n";
 }

void page_replacer::next_page(int p)
 {
   int victim=-1;
   for (int i=0;i<filled;++i)
      if (pagetable[i]==p) { victim=i; break; }
   if (victim<0)
    {
      if (filled<frames) in_entry(victim=filled++);
      else { ++swaps; victim=find_victim(); }
    }
   pagetable[victim]=p;
   access_entry(victim);
   for (int i=0;i<filled;++i) cout << setw(3) << pagetable[i];
   cout << endl;
 }
```

`pagetable` stellt die Seitentabelle dar, `filled` gibt ihren Füllstand an. `swaps` zählt die durchgeführten Swapping-Vorgänge.

Die Funktion `next_page` ist das allgemeine Herzstück der Page-Fault-Behandlung. Zunächst wird geschaut, ob die Seite noch in der Tabelle steht. Sonst wird bei noch nicht gefüllter Tabelle der erste unbenutzte Eintrag verwendet. Ist die Tabelle voll, wird die Funktion `find_victim` zur Bestimmung des Opfers aufgerufen. Sie muss in konkreten Unterklassen definiert werden. Die Funktion `test_string` wendet den Algorithmus auf eine ganze Kette von Seitenzugriffen an.

Es gibt zwei Funktionen, die für spezielle Algorithmen notwendig sind: `access_entry` wird bei jedem Seitenzugriff aufgerufen, `in_entry` bei einer Einlagerung in einen freien Frame. In Klassen, die sie nicht überschreiben, tun die Funktionen nichts.

Folgendes Hauptprogramm funktioniert mit jeder konkreten Klasse *xxxx*`_replacer`, die von der abstrakten Klasse `page_replacer` abgeleitet wird und gibt die Austauschvorgänge und die Anzahl der Swappings aus:

```
int main()
 {
   static const int reference[]=
    { 0,2,1,5,6,1,2,5,0,3,0,5,2,1,5,4,1,5,2,1,5,2,0,5,3,0,5,4,1,5,-1 };

   xxxx_replacer repl(3);
   repl.test_string(reference);
 }
```

11.6.2.1 Optimaler Algorithmus

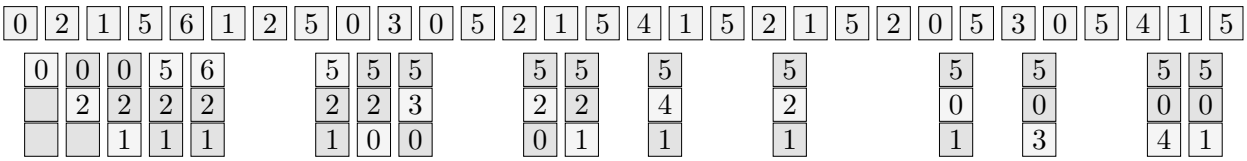
Das ist natürlich eine rein theoretische Betrachtung, da auch das beste Betriebssystem keine präkognitiven Fähigkeiten hat. Es wird jeweils die Seite ausgelagert, die am längsten nicht benutzt werden wird. Wir können hier mogeln und in die Zukunft schauen.

Der Pointer `future` steht zu diesem Zweck immer an der aktuellen Stelle der abzuarbeitenden Folge. Die Funktion `find_victim` benutzt ihn, um (sonst unerlaubterweise) weiter nach rechts zu schauen:

```
class optimal_replacer : public page_replacer
 {
   private:
      const int *future;
   public:
      optimal_replacer(int f) : page_replacer(f) { }
      virtual void test_string(const int *);
      virtual int find_victim();
 };

void optimal_replacer::test_string(const int *s)
 {
   future=s;
   while (*future>=0) next_page(*future++);
```

```
  cout << swaps << " page swaps\n\n";
 }

int optimal_replacer::find_victim()
 {
  int maxdist=-1,farthest;
  for (int i=0;i<frames;++i)
   {
    int p=pagetable[i],j;
    const int *f=future;
    for (j=0;*f>=0,*f++!=p;++j);
    if (j>maxdist) { maxdist=j; farthest=i; }
   }
  return farthest;
 }
```

Im folgenden Bild ist das Verhalten des optimalen Algorithmus, angewandt auf den Referenzstring aus dem Hauptprogramm von oben, dargestellt. Nach dem Einlagern der ersten drei Seiten werden noch 13 Swaps benötigt.



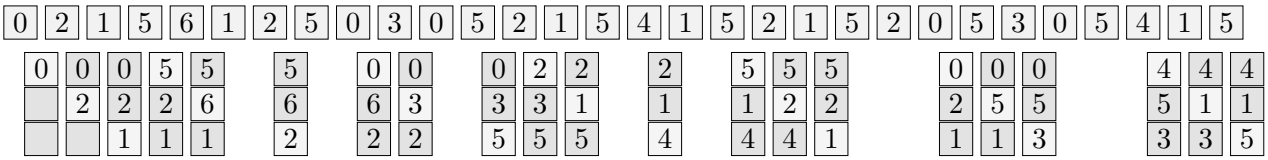
11.6.2.2 FIFO-Algorithmus

Alle Seiten werden in einer Warteschlange organisiert. Die älteste noch residente Seite wird ausgelagert, da angenommen wird, dass sie nicht mehr wirklich gebraucht wird.

Die FIFO-Performance ist ziemlich schlecht, da nicht berücksichtigt wird, dass einige wichtige Seiten fast immer benötigt werden („**hot spots**“). Sie werden aus- und kurz danach wieder eingelagert. Außerdem wächst die Page-Fault-Rate mit zunehmender Speichergröße anomal stark an (Belady 1969).

```
class fifo_replacer : public page_replacer
 {
  private:
     int next_swap;
  public:
     fifo_replacer(int f) : page_replacer(f), next_swap(0) { }
     virtual int find_victim();
 };

int fifo_replacer::find_victim()
 {
  int ret=next_swap;
  next_swap=(next_swap+1)%frames;
  return ret;
 }
```

Bei unserem Referenzstring benötigt FIFO 18 Swaps (optimal waren 13):



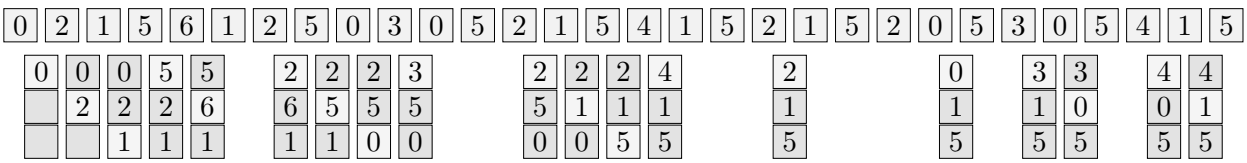
11.6.2.3 LFU-Algorithmus

LFU steht für *Least Frequently Used*. Es wird die Anzahl der Zugriffe auf jede Seite gezählt. Die Seite mit den wenigsten Zugriffen wird ausgelagert.

Der Algorithmus ist dann ungünstig, wenn zu Beginn eines Prozesses Seiten sehr stark benutzt werden, später aber nicht mehr. Sie bleiben „für immer“ unbenutzt im Speicher. Daher werden in der hier implementierten Variante von Zeit zu Zeit alle Zähler halbiert.

```
class lfu_replacer : public page_replacer
 {
   private:
      int *count,step,period;
   public:
      lfu_replacer(int f, int p)
       : page_replacer(f), count(new int[f]), step(p), period(p)
       { for (int i=0;i<frames;++i) count[i]=0; }
      ~lfu_replacer() { delete[] count; }
      virtual int find_victim();
      virtual void access_entry(int e);
 };

void lfu_replacer::access_entry(int e)
 {
   if (--step<=0)
    {
      step=period;
      for (int i=0;i<frames;++i) count[i]>>=1;
    }
   ++count[e];
 }

int lfu_replacer::find_victim()
 {
   int min=count[0],minindex=0;
   for (int i=1;i<frames;++i) if (count[i]<min) { min=count[i]; minindex=i; }
   return minindex;
 }
```

Der LFU-Algorithmus erzeugt 16 Swaps bei unserem Referenzstring (bei Periode 4):



11.6.2.4 LRU-Algorithmus

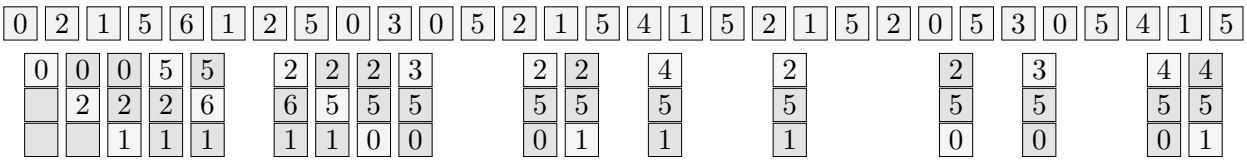
LRU steht für *Least Recently Used*. Die Seite, die am längsten nicht angesprochen wurde, wird ausgelagert.

Das LRU-Verfahren arbeitet (bei Referenzlokalität) fast so gut wie das optimale Verfahren. Leider ist es in dieser reinen Form schwer zu implementieren. Zu jeder Seite muss die letzte Zugriffszeit abgespeichert werden. Beim Auslagern müssen entweder alle Seitentabellen vollständig durchsucht werden, oder die Einträge müssen in einer doppelt verketteten Liste geführt werden.

Leider muss aber *bei jedem Speicherzugriff* das Datum in der Seitentabelle aktualisiert werden! Keine reale Hardware implementiert daher ein echtes LRU.

```
class lru_replacer : public page_replacer
 {
   private:
      int *date;
      int step;
   public:
      lru_replacer(int f) : page_replacer(f), date(new int[f]), step(0) { }
      ~lru_replacer() { delete[] date; }
      virtual int find_victim();
      virtual void access_entry(int e) { date[e]=step++; }
 };

int lru_replacer::find_victim()
 {
   int min=step,minindex;
   for (int i=0;i<frames;++i) if (date[i]<min) { min=date[i]; minindex=i; }
   return minindex;
 }
```

LRU ist mit 14 Swaps schon fast optimal:



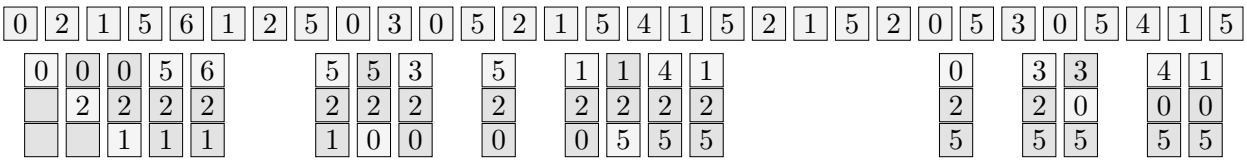
11.6.2.5 NRU-Algorithmus

Da LRU durch die Zähler für Hardware zu aufwendig ist, arbeitet NRU (*Not Recently Used*) ökonomischer mit einem Bit.

Bei jedem Zugriff auf eine Seite setzt die Hardware ein „Hit“-Flag im Seitentabellen-Eintrag. Von Zeit zu Zeit werden alle Flags gelöscht. Beim Auslagern sind Seiten mit gelöschtem Hit-Flag gute Kandidaten. (Reines NRU lagert die *erste gefundene Seite* mit gelöschtem Hit-Flag aus.)

```
class nru_replacer : public page_replacer
 {
   private:
      bool *hit;
      int step,period;
   public:
      nru_replacer(int f, int p)
       : page_replacer(f), hit(new bool[f]), step(0), period(p) { }
      ~nru_replacer() { delete[] hit; }
      virtual int find_victim();
      virtual void access_entry(int e);
 };

int nru_replacer::find_victim()
 {
   for (int i=0;i<frames;++i) if (!hit[i]) return i;
   return 0;
 }

void nru_replacer::access_entry(int e)
 {
   if (--step<=0)
    {
      step=period;
      for (int i=0;i<frames;++i) hit[i]=false;
    }

   hit[e]=true;
 }
```

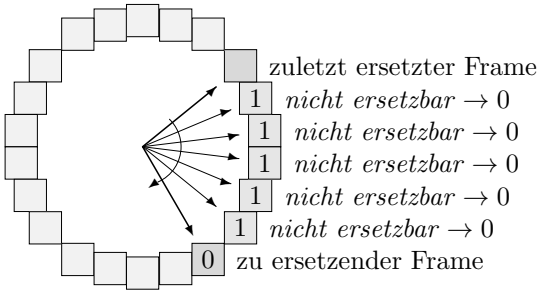
Bei unserem Referenzstring von oben benötigt NRU 15 Swaps (bei Periode 6):



11.6.2.6 Clock-Algorithmus

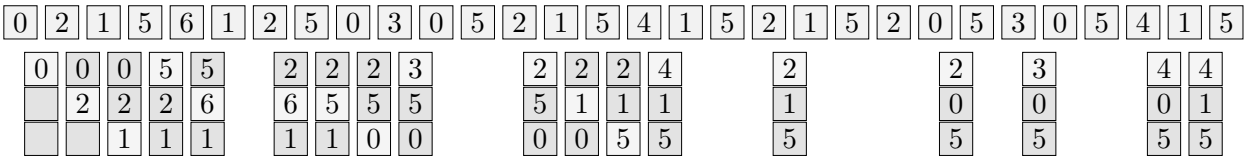
Dieser Algorithmus wird auch „*Second-Chance*-Algorithmus“ genannt. Er ist eine Kombination von FIFO und NRU und versucht, die Working-Sets der Prozesse zu approximieren.

Die Seiten liegen in einer zyklischen Liste, ein Zeiger wandert in einer Richtung darin. Bei einem Zugriff *nach dem Einlagern* wird das Hit-Flag einer Seite gesetzt. Wenn verdrängt werden muss, wandert der Zeiger in der Liste und löscht dabei alle gesetzten Hit-Flags. Wenn er eine Seite mit schon gelöschtem Hit-Flag findet, wird diese Seite ausgelagert.



```
class clock_replacer : public page_replacer
 {
   private:
      int *hit;
      int cycle;
   public:
      clock_replacer(int f) : page_replacer(f), hit(new int[f]), cycle(0) { }
      ~clock_replacer() { delete[] hit; }
      virtual int find_victim();
      virtual void access_entry(int e) { ++hit[e]; }
      virtual void in_entry(int e) { hit[e]=-1; }
 };

int clock_replacer::find_victim()
 {
   for (int i=0;i<frames;++i)
    {
      if (hit[cycle]==0) { hit[cycle]=-1; break; }
      hit[cycle]=0;
      if (++cycle>=frames) cycle=0;
    }
   int ret=cycle;
   if (++cycle>=frames) cycle=0;
   return ret;
 }
```

Dieser Clock-Algorithmus benötigt 15 Swaps bei unserem Referenzstring:

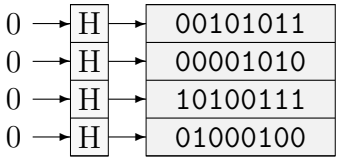


Es gibt eine Variante des Clock-Algorithmus mit zwei Zeigern mit konstantem Abstand. Der vordere prüft Hit-Flags, der hintere löscht sie. Es ergibt sich ein leicht verbessertes Page-Fault-Verhalten.

11.6.2.7 Referenzstring-Algorithmus

In jedem Eintrag wird eine Annäherung an den zurückliegenden Teil des Referenzstrings, bezogen auf die zugehörige Seite, abgespeichert. Er wird wieder mit Hilfe eines Hit-Flags (wie bei NRU) ermittelt.

Bei diesem speziellen Referenzstring handelt es sich um ein Bitmuster, in dem eine 1 eine Referenzierung der Seite (im vorgegebenen Zeitraum) anzeigt. Eine 0 bedeutet, dass die Seite nicht benutzt wurde. In regelmäßigen Abständen wird das Hit-Flag von links in das Bitmuster *eingeschoben* und dann gelöscht.



Im Verdrängungsfall werden die Bitmuster als ganze Zahlen aufgefasst. Die Seite mit der kleinsten Zahl (z.B. mit den meisten führenden Nullen) ist am längsten nicht benutzt worden und wird ausgelagert.

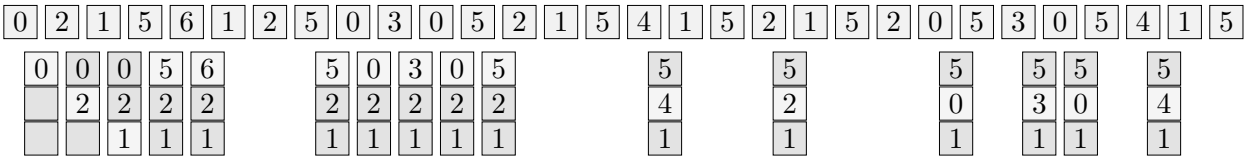
Auf diese Weise braucht zwar nur ein Hit-Flag bei jedem Zugriff gesetzt zu werden, beim Verdrängen muss aber immer noch die gesamte Seitentabelle nach der kleinsten ganzen Zahl durchsucht werden.

Unsere Beispiel-Klasse benutzt `unsigned long` (also meist 32-Bit-Zahlen) als Referenz-Strings.

```
class reference_replacer : public page_replacer
 {
   private:
      unsigned long *ref;
      bool *hit;
      int step,period;
      const unsigned long mask;
   public:
      reference_replacer(int f, int p);
      virtual int find_victim();
      virtual void access_entry(int e);
 };

reference_replacer::reference_replacer(int f, int p)
 : page_replacer(f), ref(new unsigned long[f]), hit(new bool[f]),
   step(p), period(p),
   mask(ULONG_MAX/2)
 {
   for (int i=0;i<frames;++i) { ref[i]=0; hit[i]=false; }
 }

void reference_replacer::access_entry(int e)
 {
   if (--step<=0)
    {
      step=period;
      for (int i=0;i<frames;++i)
       { ref[i]=(ref[i]>>1)|(hit[i]?mask:0); hit[i]=false; }
    }
   hit[e]=true;
 }

int reference_replacer::find_victim()
 {
   int min=ref[0],minindex=0;
   for (int i=1;i<frames;++i) if (ref[i]<min) { min=ref[i]; minindex=i; }
   return minindex;
 }
```

Dieser Algorithmus benötigt bei unserem Referenzstring (bei einer Periode von 4) 13 Swaps, also die optimale Anzahl (aber nicht mit derselben Verteilung wie beim „optimalen Algorithmus“).

Der Referenzstring-Algorithmus kann seine Stärken normalerweise allerdings erst bei wesentlich längeren Strings entfalten (was wir hier aus Platzgründen nicht vorführen können).



11.7 Segmentierung

Eine andere Möglichkeit für virtuellen Speicher ergibt sich aus dem Konzept der **Speichersegmentierung** (*memory segmentation*).

Die Abstraktion des Hauptspeichers über den logischen Adressraum (als lineare Wortfolge ab der Adresse 0) ist in gewisser Hinsicht zu starr und nicht an der logischen Struktur eines Programms orientiert.

Die einfachste Unterteilung hatten wir bereits kennengelernt:

- Code-Segment/Text-Segment (der Maschinencode)
- Daten-Segment (initialisierte und nicht-initialisierte Daten)
- Stack-Segment (Rücksprungadressen, Parameter, lokale Variablen)

(Die 8086-Architektur unterstützte mit ihren beschränkten Mitteln bereits genau diese Unterteilung – mit Hilfe der entsprechenden Segment-Register.)

Zusätzlich gibt es aber diverse Strukturen innerhalb des Maschinencode-Teils. Es gibt

- ein Hauptprogramm,
- Module aus mehreren Unterprogrammen,
- eventuell Unterprogramme in Unterprogrammen verschachtelt,
- „eigenen“ Code und Bibliotheks-Code,
- Hilfsstrukturen (z.B. die Verwaltung langer Sprünge),
- systemabhängig Diverses mehr.

Wenn starr mit einem linearen Adressraum gearbeitet wird (z.B. bei reinem Paging), kann leicht eine solche Struktur zertrennt werden, und nicht zusammengehörige Teile liegen nebeneinander in einer Seite. Diese Strukturierung entspricht nicht dem Ablauf- und Aufruf-Verhalten beim tatsächlichen Prozess. Man bezahlt dies mit Effizienz-Einbußen.

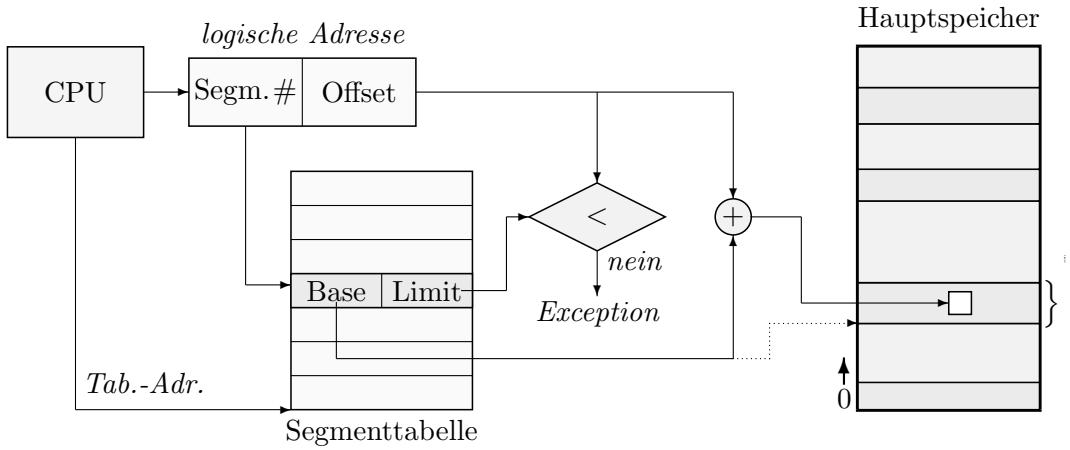
Bei der Segmentierung wird die logische Struktur eines Programms auf die Darstellung im Hauptspeicher übertragen. Die logischen Einzelteile (Segmente) können geschickter im Speicher verteilt, adressiert und verwaltet werden.

Die logischen Einheiten ergeben sich normalerweise direkt aus dem Programm-Quelltext. Es ist Aufgabe von Compiler und Linker, die entsprechende Unterteilung vorzunehmen. Beispielsweise könnte in `C` jedes „Modul“ (ein separat übersetzter Quelltext) oder in `C++` jede Klasse ein eigenes Segment erzeugen. Eventuell stellen Assembler und Compiler auch Direktiven bereit, selbst eine Unterteilung (in benannte Segmente) vorzunehmen.

11.7.1 Segmentierungs-Hardware

Die Segmente eines Prozesses werden meist einfach ab 0 durchnummeriert (aus technischen Gründen – s.u. – kann die Numerierung aber Lücken enthalten). Die logische Adresse (aus der Prozess-Sicht) ist dann zweigeteilt – in die Segment-Nummer und den Offset innerhalb des Segments.

Der physische Hauptspeicher ist natürlich immer noch eine lineare Wortfolge. Analog zum Paging muss spezielle Hardware nun die entsprechende Umsetzung vornehmen. Die Grundlage für die Abbildung bildet wieder eine Tabelle, die „**Segment-Tabelle**“.



Die Segmenttabelle kann völlig analog zur Seitentabelle beim Paging implementiert werden. Entweder stellt man also wieder einen Registersatz zur Verfügung oder bringt sie im Hauptspeicher unter (dabei helfen wiederum Caches oder TLBs). Die Register mit Anfangsadresse und Länge der Tabelle heißen hier **STBR** (*Segment Table Base Register*) und **STLR** (*Segment Table Length Register*).

11.7.2 Shared Memory

Die Tatsache, dass in einem Segment zusammengehörige Daten untergebracht sind, bietet einige Vorteile. Sie lassen sich beispielsweise gemeinsam schreibschützen oder als ausführbar kennzeichnen. Beim Paging müssten dafür viele kleine Seiten einzeln markiert werden, etc.

Vor allem lässt sich die gemeinsame Benutzung von Segmenten (von mehreren Prozessen aus) wesentlich einfacher verwalten, als das mit Paging möglich wäre.

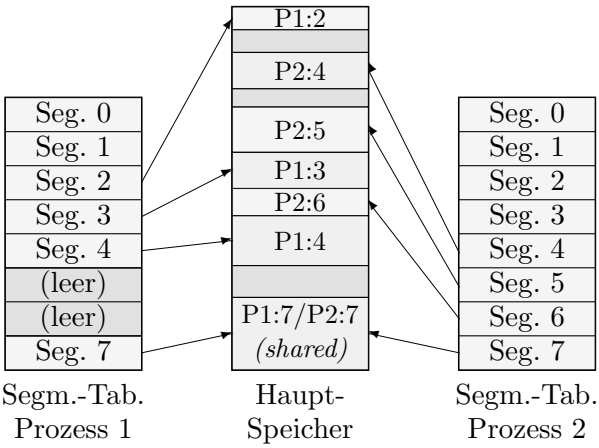
- Ein mehrfach gestartetes Programm (z.B. ein Editor) braucht nur in einer Kopie im Speicher zu liegen, sodass (besonders bei vielen Benutzern) viel Speicher gespart werden kann. Das ist nur mit Code-Segmenten (und ggf. mit konstanten Daten) möglich, insbesondere einen eigenen Stack braucht natürlich jeder Prozess!
- Analog kann auch ein und dieselbe Kopie einer Bibliothek von mehreren Prozessen aus benutzt werden (*Shared Libraries*).

Theoretisch ist das auch mit Paging denkbar, aber schwieriger. Code- und Datenteile halten sich dort nicht an Seitengrenzen, sodass eine Seite beides gemischt enthalten kann.

Es gibt noch kleine technische Probleme von der Systemseite aus zu lösen.

Beispielsweise kann der gemeinsam benutzte Code Sprünge enthalten (nicht PC-relative, aber noch innerhalb des Segments). Die zugehörigen logischen Adressen enthalten dann die *Segment-Nummer* dieses Segments.

Wenn ein Segment von mehreren Prozessen aus genutzt wird, muss es aber in allen Prozessen *dieselbe Segmentnummer* tragen! Gegebenenfalls muss das System Segment-Nummern im Code nachträglich ändern – z.B. so weit nach oben schieben, bis die Nummer in allen angeschlossenen Prozessen frei ist.



11.7.3 Dynamischer Speicher

Die Segmentierung bietet einen weiteren Vorteil: Dynamisch wachsende Strukturen können leichter unterstützt werden.

Code- und Datensegment haben i.Allg. während eines Programmlaufs konstante Größe. Dagegen wächst und schrumpft der Laufzeit-Stack kontinuierlich, bei jedem Unterprogramm-Aufruf und -Rücksprung. Ebenso (wenn auch nicht so häufig) ändert der Heap seine Größe, also der Speicherbereich, in dem Funktionen wie `malloc` dynamischen Speicher reservieren.

Wenn der Stack „überläuft“ – also zu viele verschachtelte Unterprogramme (oder mit großen lokalen Datenmengen) aufgerufen werden, wird der Adressbereich des Stack-Segments überschritten und eine Exception („*Segmentation Fault*“) ausgelöst. Das System kann nun durch Vergrößerung (und ggf. Neuplazierung) des Stack-Segments reagieren, ohne dass die Anwendung überhaupt von dem Vorfall etwas mitbekommt.

Analog kann `malloc` bei Speichermangel im Heap das System bitten, das zugehörige Segment zu vergrößern. Gelegentlich könnte auch nachgeschaut werden, ob Heap oder Stack so geschrumpft sind, dass sich eine Segment-Verkleinerung lohnt.

11.7.4 Virtueller Speicher durch Segmente

Auch mit Segmentierung lässt sich echter virtueller Speicher schaffen. Es müssen analog zum Demand Paging Auslagerungs-Strategien auf die Festplatte hinzugenommen werden.

Dabei sind vergleichsweise kleine Segmente wünschenswert, da natürlich nur komplette Segmente ausgelagert werden können. Zu klein können Segmente aber nicht gewählt werden, da logisch zusammengehörige Daten getrennt würden. Das Stack-Segment ist beispielsweise praktisch unteilbar.

Überhaupt treten bei reiner Segmentierung leider dieselben Probleme auf, die wir bereits bei der Unterteilung in Partitionen kennengelernt haben. Insbesondere kann es schnell zu **externer Fragmentierung** kommen.

Es kann aber leicht ein Kompaktifizierungs-Algorithmus angewandt werden. Es müssen zwar die Segmente im Speicher herumkopiert werden, durch die Segmentierungs-Hardware ist aber keinerlei Anpassung von Adressen im Code notwendig.

11.8 Segmentiertes Paging

Man kann Paging und Segmentierung durchaus sinnvoll kombinieren und ein Verfahren erhalten, das die Vorteile beider vereint („*page-segmented systems*“). Die MULTICS-Architektur arbeitete bereits so, und die 80x86-Familie seit dem 80386 ebenfalls.

Sinnvollerweise ist die erste Stufe der Unterteilung die in Segmente, die zweite die Unterteilung von Segmenten in Seiten.

- Die im Allgemeinen zu großen Segmente bei reiner Segmentierung werden in Seiten unterteilt, was Auslagerungs-Probleme und externe Fragmentierung vollständig behebt. (Die interne Fragmentierung beläuft sich auf durchschnittlich eine halbe Seite *pro Segment*.)
- Die Zusammenfassung von Seiten zu Segmenten bietet dagegen eine bessere Verwaltung (Schreibschutz, gemeinsamer Speicher) und die Möglichkeit dynamischer Erweiterung.

Eine logische Adresse ist nun dreigeteilt (analog zum mehrstufigen Paging, Seite 298), nach folgendem Schema:

Segment-Nummer	Seiten-Nummer	Offset

Jeder Prozess besitzt eine eigene Segment-Tabelle, deren Einträge auf Seiten-Tabellen verweisen, die Frame-Nummern im Hauptspeicher enthalten. Die Seiten-Tabellen haben unterschiedliche Längen, so wie die Segmentgrößen es verlangen.

Die Segment-Tabellen können relativ groß werden. Einige Systeme wenden daher zusätzlich einen Paging-Algorithmus *auf die Segment-Tabellen* an, so dass eine weitere Zwischenstufe entsteht!

11.9 Konkrete virtuelle Speichersysteme

11.9.1 Speichermodelle beim 80386

Für den 80386 wurde ein ziemlich komplexes **segmentiertes** Speichermodell mit **zweistufigem Paging** und **Translation-Lookaside-Buffer** eingeführt. Es hat viele subtile Eigenarten, die wir hier nicht alle behandeln können. Als Beispiel für segmentiertes Paging ist es aber gut geeignet.

Der 8086-Prozessor arbeitete mit dem segmentierten Speichermodell, das wir bereits in Kapitel 2 kennengelernt haben.

- Die maximale Segmentgröße betrug 64 KByte, und die Segment-Positionen wurden durch vier Segment-Register definiert. Speicherzugriffe waren immer segmentiert. Für einen beliebigen Zugriff mussten Segment-Adresse und Offset angegeben werden.

- Der Hauptgrund für diese Art der Segmentierung war allerdings, dass man nur 16 Bits für die Adressdarstellung spendieren wollte, aber mehr als 64 KByte Speicher ansprechen können wollte.
- Code, der nicht in ein 64 KByte-Segment passte, musste kompliziert Verbindung zwischen den Segmenten herstellen. Datenblöcke, die größer als 64 KByte waren, konnten nur sehr kompliziert behandelt werden, da mitten im Block der Wert des Segmentregisters geändert werden musste.

Der Einsatz von Segmenten auf dem 8086 brachte eigentlich nur Nachteile. Die Deskriptor-Lösung des 80286 war eher eine Hilfskonstruktion, auf die wir hier nicht genau eingehen wollen. Mit dem 80386 stellte Intel dann (für den Protected Mode) einen kombinierten Segmentierungs-/Paging-Mechanismus zur Verfügung. Aus Kompatibilitätsgründen ist er allerdings an den 80286 angelehnt und deshalb nicht besonders elegant – aber effizient – realisiert (und außerdem muss ja noch Hardware für die 8086-Segmentierung im Real-Mode zur Verfügung stehen).

Je nach Organisation der benötigten Segment- und Seitentabellen kann man fast jedes Speichermodell realisieren, also

- flach (ein linearer Adressraum)
- Segmentierung (wie bei den Vorgängerprozessoren oder frei)
- Paging (durch Ignorieren der Segmentierung)
- segmentiertes Paging (die übliche Wahl, z.B. unter L*i*nux und Windows)
- letztere jeweils mit oder ohne virtuellen Speicher (dies ist ja ohnehin Aufgabe der System-Software)

Der 80386 hat zusätzliche Register zur Steuerung von Segmentierung, Paging und Multitasking:

`CR0:`	Control Register 0, Prozessor-Optionen (z.B. Paging ja oder nein)
`CR1:`	Control Register 1, reserviert
`CR2:`	Control Register 2, Adresse der Page-Fault-Behandlungsroutine
`CR3:`	Control Register 3, Adresse des Page-Directories
`GDTR:`	Adresse der globalen Deskriptortabelle
`LDTR:`	Adresse der lokalen Deskriptortabelle
`IDTR:`	Adresse der Interrupt-Deskriptortabelle
`TR:`	Adresse des Task-Status-Segments

Die Adressen sind echte physische Hauptspeicheradressen im linearen Adressraum. Die meisten Begriffe werden in den folgenden Abschnitten erklärt. Vorab nur Folgendes:

Ein **Task-Status-Segment** (TSS) ist eine Datenstruktur ähnlich einem Prozess-Kontrollblock, die bei der hardwaremäßigen Unterstützung für Multitasking verwendet wird. Dieses Segment enthält Abbilder der Daten-Register, des Programmzählers, etc., aber auch der MMU-Daten, also Adressen von Seitentabellen, etc. Weitere Teile sind vom Betriebssystem frei verwendbar.

11.9.1.1 Segmentierung

- Der Speicher im Protected Mode ist *immer* segmentiert, eventuell gibt es aber Segmente, die den gesamten Adressbereich von 4 GByte abdecken.
- Segmente dürfen sich im Adressraum überlappen.

- Code- und Datensegmente werden hardwaremäßig unterschiedlich behandelt, sodass es immer zumindest ein Code- und ein Datensegment geben muss.
- Segment-Größen können von 1 Byte bis 1 MByte in Schritten von einem Byte, von 1 MByte bis 4 GByte in Schritten von 4096 Bytes gewählt werden. Es kann kein Code in Daten-Segmenten ausgeführt werden (wird nur bei Sprüngen überprüft). Segmente können lese- und schreibgeschützt werden (wird bei jedem Zugriff geprüft).
- Eine logische Adresse ist 48 Bit lang. Die obersten 16 Bit wählen das Segment aus und werden **Selektor** genannt, die unteren 32 Bit stellen den Offset im Segment dar. Da ein Segment ja maximal 4 GByte groß sein kann, sind die 32 Bit für den Offset auch erforderlich.

  Die Selektoren stammen beim 80386 immer aus speziellen Registern und sind oft implizit durch den Maschinenbefehl festgelegt, während der Offset direkt im Befehl angegeben ist.
- Die gängigen CPU-Befehle arbeiten mit 32-Bit-Adressen. Sie beziehen sich – je nach Zusammenhang – immer auf das „aktuelle“ Code-, Daten-, Stack- oder Extra-Segment. Diese Segmente werden durch die CPU-Register `CS`, `DS`, `SS`, bzw. `ES` festgelegt.
- Es gibt zwei zusätzliche Segment-Register `FS` und `GS`, die in Datenzugriffen analog zu `ES` verwendet werden können. Code, der auf verschiedene Datensegmente zugreift, kann so beschleunigt werden. Sie haben keine festgelegte Bedeutung wie `ES` bei einigen String-Befehlen.
- Die Segment-Register enthalten im Protected Mode keine 8086-Segment-Nummer (also Segment-Adresse, geteilt durch 16), sondern einen 16-Bit-Selektor – einen Index in die Segmenttabelle (genaueres weiter unten).

Mit dieser Kenntnis kann man beispielsweise folgende einfachen Speichermodelle „bauen“:

- Es wird ein Code- und ein Datensegment definiert, die beide jeweils den gesamten Adressbereich von 4 GByte umfassen. `ES`, `SS` werden auf den Wert von `DS` gesetzt. Die Registerwerte werden nie mehr verändert. So hat man einen 4 GByte großen, flachen, linearen Adressraum (ohne Speicherschutz und Paging) realisiert.
- Die Segmentverteilung des letzten Falls werden für das Betriebssystem übernommen. Je ein weiteres, kleineres Daten- und Code-Segment wird für (alle) Anwenderprogramme zur Verfügung gestellt. Die Daten des Betriebssystems sind so geschützt. Bei Betriebssystem-Aufrufen muss das Code-Segment des Systems explizit angegeben werden (langsamer als einfache Sprünge).

11.9.1.2 Segment-Deskriptoren

Die Segment-Größen, -Positionen und -Flags werden mit Hilfe von **Deskriptoren** festgelegt. Die Segmenttabelle hat als Einträge solche Deskriptoren und wird daher hier auch meist **Deskriptortabelle** genannt.

Ein Deskriptor ist eine Datenstruktur der Länge vier 16-Bit-Worte, die im Hauptspeicher liegt. Deskriptoren für Code und Daten sehen leicht unterschiedlich aus (der seltsame Aufbau ist u.a. durch die 80286-Kompatibilität bedingt):

Code-Deskriptor

	Bit 15															Bit 0
W3	BASE 31-24								G	D	0	AV	LIM. 19-16			
W2	P	DPL		1		C	R	A	BASE 23-16							
W1	BASE 15–0															
W0	LIMIT 15–0															

Daten-Deskriptor

	Bit 15															Bit 0
W3	BASE 31-24								G	B	0	AV	LIM. 19-16			
W2	P	DPL		1		0	E	W	A	BASE 23-16						
W1	BASE 15–0															
W0	LIMIT 15–0															

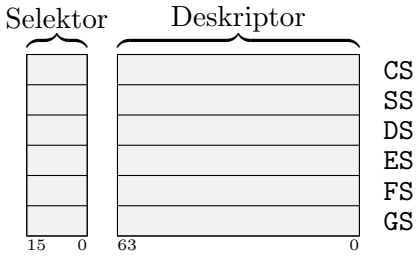
Die Felder haben folgende Bedeutung:

BASE	Basisadresse (32 Bit ⇒ 4 GByte)	LIMIT	Grenzadresse (20 Bit ⇒ bis 1 MByte bei 1-Byte-Schritten, 4 GByte bei 4-KByte-Schritten)
G	Granularität (siehe LIMIT, 0=byte-weise, 1=4-KByte-weise)	B	Big, Segment länger als 64 KByte
D	1=80386-Code, 0=80286-Code	DPL	*Descriptor Privilege Level* (0 bis 3), Privileg-Stufe, „Ring“
P	Present, Segment komplett physisch im Hauptspeicher	C	Conforming
R	Readable	W	Writeable
E	Expansions-Richtung (1 = expand-down)	A	Accessed, Zugriff erfolgt (Hinweis für die Auslagerungs-Strategie)
D	Default Operand Size (1=32 Bit)	AV	Available, frei verfügbar (System)

- Wenn `B` für ein Stack-Segment gesetzt ist, wird `ESP` (32 Bit) als Stack-Pointer verwendet, sonst `SP` (16 Bit).
- Wenn `E` gesetzt ist, wird ein Segment bei einer notwendigen Vergrößerung nicht zu *höheren* Adressen erweitert (*expand-up*), sondern zu niedrigeren (*expand-down*). Da Stacks meist zu niedrigeren Adressen hin wachsen, sind üblicherweise genau Stack-Segmente expand-down.
- Der `DPL` legt die Privilegstufe des Codes in einem Code-Segment fest. Der 80386 unterstützt nicht nur einen User- und einen Supervisor-Mode (*dual mode instructions*), sondern vier Modi, die sogenannten „**Ringe**“. Ring 0 ist der privilegierteste und gedacht für den Betriebssystem-Kern und sehr hardwarenahen Code, Ring 3 ist der am wenigsten privilegierte und soll Anwenderprogramme aufnehmen. Dazwischen sind Abstufungen (je nach tatsächlichem Bedarf der Privilegien) möglich.

  Die meisten Betriebssysteme (auch Windows und L*i*nux) benutzen aber nur Ring 0 (Kernel) und Ring 3 (Rest des Systems und Anwenderprogramme). Das Paging-System hat übrigens wieder nur zwei Modi (über das U/S-Bit).

Die Segment-Register `CS`, etc. sind in Wirklichkeit nicht 16 Bit, sondern 80 Bit groß. Es kann nur der 16-Bit-Selektor-Teil, der den Index in die Deskriptortabelle darstellt, direkt beschrieben und gelesen werden. Intern hat jedes Register *zusätzliche 64 Bit*, in dem eine Kopie des kompletten zugehörigen Deskriptors gespeichert wird.



Es muss also nicht bei jedem Speicherzugriff aus der Deskriptortabelle gelesen werden. Nur bei jeder Änderung eines Segment-Registers wird der entsprechende Deskriptor-Teil aus dem Speicher geholt.

Das Verhalten entspricht also einem TLB (Translation Lookaside Buffer), den wir beim Paging (Seite 297) besprochen haben – nur dass er direkt durch den Programmcode gesteuert wird und kein assoziativer Speicher zu sein braucht.

11.9.1.3 Deskriptortabellen

Deskriptortabellen können bis zu 8192 Einträge lang sein (da man mit 16-Bit-Selektoren 64 KByte ansprechen kann und ein Deskriptor 64 Bits lang ist).

- Die globale Deskriptor-Tabelle (GDT) enthält Deskriptoren für alle Prozesse. Das CPU-Register `GDTR` (*Global Descriptor Table Register*) enthält ihre Adresse. Es wird normalerweise nur einmal beim Systemstart mit dem Befehl `lgdt` gesetzt. Hierüber wird meist Speicher angesprochen, der systemweit benutzt wird (z.B. Betriebssystem-Schnittstellen).
- Die lokale Deskriptor-Tabelle (LDT) jedes Prozesses enthält Prozess-spezifische Deskriptoren. Hierauf zeigt das Register `LDTR` (*Local Descriptor Table Register*), das bei einem Task-Wechsel mit `lldt` geändert werden kann.

11.9.1.4 Paging

Während eine Art von Segmentierung beim 80386 zwingend notwendig ist, ist Paging optional, wird aber von praktisch allen Betriebssystemen für virtuellen Speicher eingesetzt. Es kann durch Setzen des PG-Flags im CPU-Register `CR0` bei der Systeminitialisierung eingeschaltet werden.

Das Paging ist *logisch hinter* die Segmentierung geschaltet, was aber keine zusätzliche Verlangsamung bedeutet. In der Hardware der MMU gehen Einträge aus Segment-Tabellen und aus Seitentabellen parallel in die endgültige physische Adresse ein.

Es kann je nach Betriebssystem unterschiedliche Beziehungen zwischen Segmenten und Seiten geben:

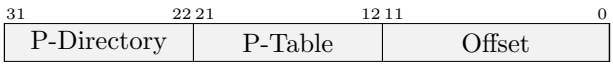
- Wenn es je Prozess nur wenige große Segmente gibt, wird üblicherweise jedes Segment aus mehreren bis sehr vielen Pages bestehen. Jedes Segment beginnt an einer Seitengrenze. Die letzte Seite eines Segments wird eventuell nicht ganz ausgenutzt.
- Wenn es sehr viele Segmente je Prozess gibt (z.B. eines für jedes Unterprogramm), sind die meisten Segmente kleiner als eine Page. Um nicht sehr viel Speicher zu vergeuden, liegen dann mehrere Segmente in einer Page, oder es wird innerhalb des Prozesses gar nicht mehr auf Page-Grenzen geachtet. Die Segmentierung dient dann zur logischen Verwaltung, das Paging nur für virtuellen Speicher.

Die Seitentabellen werden üblicherweise in Segmenten mit der Privilegstufe `DPL=0` abgelegt, damit nur die Systemsoftware der Stufe 0 darauf zugreifen kann.

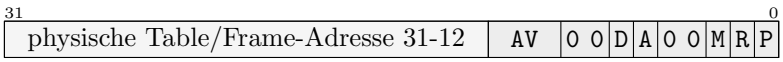
Das Paging des 80386 ist zweistufig, d.h. es gibt Page-Directories, deren Einträge Zeiger auf Seitentabellen sind.

Jedes Page-Directory kann bis zu 1024 Einträge groß sein, und jede Page-Tabelle kann bis zu 1024 Einträge haben. Da jede Seite 4 KByte groß ist, kann so über ein Directory der gesamte 4 GByte-Adressraum angesprochen werden.

Entsprechend sind in der logischen 32-Bit-Adresse (die aus der Segmentierungsstufe herauskommt) der Page-Directory-Teil und der Page-Tabellen-Teil jeweils 10 Bit lang, und der Offset innerhalb der Page wird durch 12 Bits beschrieben:



Die Einträge im Page-Directory (PDE, Page Directory Entries) und im Page-Table (PTE, Page Table Entries) sind einander im Aufbau ähnlich:



Die Umsetzung von logischen in physische Adressen ist wieder denkbar einfach. Bei einem PTE werden die unteren 12 Bits durch den Offset ersetzt. Bei einem PDE werden die Bits 11–2 durch die Page-Nummer ersetzt (und die Bits 1–0 auf 0 gesetzt).

Oben bedeuten:

`D`	Dirty	`A`	Accessed
`P`	Present	`R`	Readable/Writeable
`AV`	Available (frei verfügbar für das System)	`M`	Modus, `U/S` (User/Supervisor)

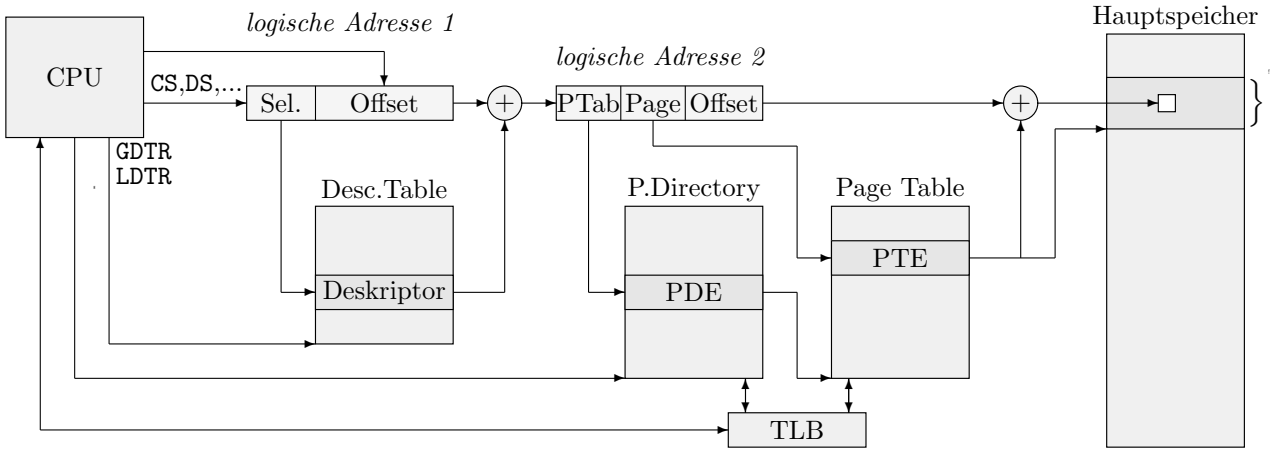
Im Fall `P=0` (also wenn die Seite ausgelagert ist), können alle 32 Bits des Eintrags frei vom System verwendet werden. Sie geben dann in irgendeiner Weise an, wo die Seite auf dem Sekundärspeicher zu finden ist.

Es gibt nun drei Sätze von Eigenschaften (wie „Writeable“), die eine Seite hat, und die in folgender Reihenfolge geprüft werden:

- Eigenschaften des Segments
- Eigenschaften aus dem Page-Directory-Entry
- Eigenschaften aus dem Page-Table-Entry

Die 32 zuletzt benutzten Pagetable-Einträge werden im Prozessor in einem TLB gespeichert. Durchschnittlich 98 % der Hauptspeicher-Zugriffe kommen daher ohne einen zusätzlichen Lese-Zugriff aus der Tabelle aus.

Im folgenden Bild sind noch einmal alle Elemente der Adressumsetzung zusammen dargestellt.

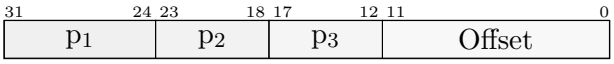


11.9.2 Paging bei SPARC-Prozessoren

Auch die SPARC-Familie, die in den SUN Workstations eingesetzt wird, arbeitet mit einer mehrstufigen Adressumsetzung. Die MMU ist wiederum im Prozessor integriert.

Es wird mit logischen Adressen der Länge 32 Bit gearbeitet (bis 4 GByte), während der Hauptspeicher mit 36 Adressleitungen angesprochen wird (bis 64 GByte). Die Seitengröße beträgt 4 KByte ($=2^{12}$ Bytes). Ein Frame wird daher durch 36:12=24 Bits beschrieben.

Die logische Adresse ist wie folgt aufgeteilt:



Page-Directories im obersten Level sind 1 KByte groß (8-Bit-Indizes, also maximal 256 Einträge à 4 Byte), die im zweiten Level und die Page-Tables im dritten Level nur 256 Bytes (6-Bit-Indizes).

Alle Tabellen müssen an einer durch 256 teilbaren Adresse beginnen, sodass ihre Lage durch 36-8=28 Bits festgelegt ist. Die Deskriptoren sind 32 Bits groß, und ihre übrigbleibenden 4 Bits werden für Steuerzwecke verwendet. Die zwei-Bit-Zahl „ET“ (entry type) legt beispielsweise die Art des Eintrags fest:

0 (Nulldeskriptor, d.h. unbenutzter Tabellenteil),
1 (Seiten-Deskriptor),
2 (Seitentabellen-Deskriptor).

Auch die SPARC-MMUs haben einen integrierten TLB, dessen Größe von Prozessor zu Prozessor aber variiert. Seine Einträge müssen explizit gelöscht werden! Der „Flush“-Befehl leert

- den ganzen TLB,
- nur Einträge, die einem speziellen Prozess gehören (ein solcher Verweis wird in jedem TLB-Eintrag mitgeführt),
- oder exakt bestimmte einzelne Einträge.

12 Einige UNIX-Spezialitäten

12.1 UNIX-Geräte-Dateien

12.1.1 Special Files

Anwenderprogramme können Geräte wie Drucker, Terminal, etc. wie normale Dateien ansprechen. Zu allen Geräten gibt es (üblicherweise im Verzeichnis `/dev`) eine Spiegelung als Datei („**special file**“), beispielsweise:

`/dev/console`	Systemkonsole
`/dev/hda9`	Festplatten-Partition (HD=HardDisk)
`/dev/null`	Daten-Nirvana
`/dev/ram0`	RAM-Disk
`/dev/tty`	aktuelles Terminal

Die Gerätedateien können an beliebiger Stelle im Dateibaum liegen, vorausgesetzt, das Filesystem dort unterstützt das Anlegen von Special Files (also beispielsweise nicht auf DOS-Partitionen). In praktisch allen Systemen liegen sie aber gesammelt im Verzeichnis `/dev`.

Die *eigentlichen* Treiber-Routinen für das Ansprechen von Geräten auf der Rechnerseite liegen im Betriebssystem-*Kernel*. Die Gerätedateien sind nur Verweise auf Kernel-Bestandteile und enthalten *Parameter* für sie.

Wenn man ein ganz neues Gerät betreiben möchte, bedeutet das, dass man (abgesehen vom Anlegen der Special Files) den Kernel (teilweise) neu compilieren und linken muss.

- Eine Gerätedatei ist entweder *blockorientiert* (‘`b`’, für Platten) oder *zeichenorientiert* (‘`c`’, für Drucker, Terminals, etc.) – ein entsprechender Buchstabe erscheint ganz links in der Auflistung bei „`ls -l`“:

  ```
  crw--w--w-   1 root     tty        4,   0 Nov  8 13:29 /dev/console
  brw-rw----   1 root     disk       3,   9 Mar  5  1998 /dev/hda9
  crw-rw-rw-   1 root     root       1,   3 Apr 18  1998 /dev/null
  brw-rw----   1 root     disk       1,   0 Mar  5  1998 /dev/ram0
  crw--w----   1 root     root       4,   1 Nov  8 13:29 /dev/tty
  ```

  Wenn ein Gerät sowohl block- wie auch zeichenweise angesprochen werden kann, gibt es zwei Special Files, eines von jeder Art.

- Auf die Gerätedateien sind die normalen Datei-Operationen wie `open`, `close`, `read`, `write` anwendbar, die üblicherweise auch die erwarteten Bedeutungen haben. Die genaue Umsetzung ist aber Aufgabe der eigentlichen Treiber.

  Bei einem `open`-Aufruf auf ein Special File geschieht im Kernel fast all das, was auch beim Öffnen einer normalen Datei geschieht. Die interne Verwaltungsstruktur (I-Node) wird angelegt, die Anzahl der zugreifenden Benutzer wird heraufgezählt, die Datei wird in die globale Tabelle offener Dateien eingetragen, und der Benutzer erhält einen File-Deskriptor zurück. *Zusätzlich* wird aber die gerätespezifische Initialisierungsroutine aufgerufen.

- Die Zugriffsberechtigung vorausgesetzt, kann jeder Benutzer direkt auf diese Dateien schreiben oder aus ihnen lesen:

```
cp lang.ps /dev/tty      # auf das aktuelle tty schreiben
od -x /dev/hda9          # Festplatte "roh" lesen, nur Super-User
```

Beispiel: Folgendes Programmstück liest die ersten Bytes einer Festplatten-Partition (*L*inux-Name) also vermutlich Teile ihres „Superblocks“ (bleibt dem Superuser vorbehalten):

```
int main()
 {
   int i,j,fd;
   if ((fd=open("/dev/hda",O_RDONLY))< 0)
      puts("huch, erste Festplatte weg!");
   else
    {
      for (i=0;i<10;++i)
       {
         unsigned char buffer[64];
         printf("%04x: ",i<<6);
         read(fd,buffer,64);
         for (j=0;j<64;++j)
          {
            char c=buffer[j];
            putchar(isprint(c)?c:'.');
          }
         putchar('\n');
       }
    }
 }
```

Eine mögliche Ausgabe (gekürzt) ist folgende:

```
0000: .3.....|..P.P.....................<.t..<.u......u......L.......
0040: .t..<.t.....<.t.V.......^.........|...W.._s.3...Ou...........}.=
0080: U.u.....|..Invalid partition table.Error loading operating syste
00c0: m.Missing operating system...0!.V3.VVRP.SQ...V..PR..B.V$..ZX.d.r
0100: .@u.B......^..tUngültige Partitionstabelle.Fehler beim Laden des
0140:  Betriebssystems.Betriebssystem fehlt.........................
```

12.1.1.1 `ioctl`

Das Analogon zur File-Kontrollfunktion `fcntl` für Gerätedateien ist der Systemaufruf `ioctl` (definiert in `sys/ioctl.h`). Mit ihm kann man (per File-Deskriptor) Steuerkommandos an ein Gerät schicken. Die Kommandos sind allerdings absolut geräteabhängig:

	UNIX/DOS
int ioctl**(int** fd, **int** request, ...**);** schickt das Kommando Nummer `request` über den File-Deskriptor `fd` an ein Gerät. Dabei steht ... für kommandoabhängige Argumente.	

Die möglichen Werte für `request` und Konstanten für die weiteren Argumente sind in eigenen Header-Dateien definiert, eine für jedes Gerät. Mit den L*i*nux-Namen und -Kommandos sind z.B. folgende Aufrufe sinnvoll:

```
fd=open("dev/lp");        // Drucker, LinePrinter
ioctl(fd,LPRESET);        // Drucker-Reset
ioctl(fd,LPABORT,1);      // kein Retry, nach Fehlern abbrechen
```

Eine relativ vollständige Liste erhält man unter L*i*nux mit dem Aufruf „`man ioctl_list`“.

Beispiel: Folgendes C-Programm spielt unter L*i*nux das erste Stück einer Musik-CD ab. Auf einen Tastendruck hin beginnt das Programm, das Stück *auszublenden* (ca. 7 Sekunden). Zuletzt wird die CD angehalten.

```
#include <sys/ioctl.h>
...
#include <linux/cdrom.h>

void sigalrm_handler(int sig) { }

int main()
 {
   int cdrom,timer_dec;

   struct cdrom_ti ti={ 1,0,2,0 };
   struct cdrom_volctrl volctrl={ 255,255,0,0};

   struct sigaction sigact={ sigalrm_handler, 0, SA_RESTART,0 };
   struct itimerval iv={ 0,30000,0,30000},ov;

   if ((cdrom=open("/dev/cdrom",O_RDONLY,0))<0) { perror("oops"); exit(0); }

   ioctl(cdrom,CDROMVOLCTRL,&volctrl);
   if (ioctl(cdrom,CDROMPLAYTRKIND,&ti)) { printf("Track-Fehler\n"); exit(0); }

   getchar();

   sigaction(SIGALRM,&sigact,&sigact);
   timer_dec=255;
   setitimer(ITIMER_REAL,&iv,&ov);
   while (timer_dec>0)
    {
      volctrl.channel0=volctrl.channel1=--timer_dec;
      ioctl(cdrom,CDROMVOLCTRL,&volctrl);
      pause();
    }
   iv.it_value.tv_sec=iv.it_value.tv_usec=0;
   setitimer(ITIMER_REAL,&iv,&ov);

   ioctl(cdrom,CDROMSTOP);
   close(cdrom);
 }
```

Vorsicht: Oft haben nur `root` und die Gruppe `disk` überhaupt Leserechte für das CD-ROM. Wenn andere Benutzer das Programm benutzen können sollen, muss einer dieser beiden das SUID- oder SGID-Bit für das Programm setzen!

In der Struktur `struct cdrom_ti` sind Anfangs- und End-Track und -Index angegeben. `struct cdrom_volctrl` enthält die Lautstärke-Werte der vier möglichen Kanäle (Nummer 3 und 4 sind dabei zukunftsorientiert), jeweils im Bereich von 0 bis 255.

Beim Ausblenden arbeiten wir mit der Timer-Funktion `setitimer` und lassen uns alle 30 ms ein `SIGALRM`-Signal schicken (dazu die Einträge 0 Sekunden, 30000 Mikrosekunden in `struct itimerval`). Unser Signal-Handler tut dabei gar nichts. Damit aber nicht automatisch das Default-Verhalten (Abbruch) eingestellt wird, arbeiten wir mit `sigaction` (statt mit `signal`).

12.1.1.2 Device Numbers

Jede Gerätedatei ist durch zwei Kennziffern charakterisiert (sie sind statt Blocknummern in ihrer „I-Node“ abgelegt, s.u.):

major device number: die Geräteart (Platte/Controller, Terminal, etc.)

minor device number: je nach Geräteart unterschiedlich (Nummer der Platte bzw. Partition, Nummer der Schnittstelle bei Terminals, Flags für Streamer)

Eine Faustregel ist folgendes: Die Major Number wählt unter den grundlegend unterschiedlichen Treibern im Kernel einen aus; die Minor Number ist eher als Parameter an diesen Treiber anzusehen. Wie die Zahlen vergeben werden, variiert von System zu System. (Intern werden die Treiber meist in ganz anders geführt, z.B. in einer Liste oder Tabelle, und durch ID-Nummern identifiziert.)

Das Kommando `file` zum Erkennen des Dateityps gibt die Numbers mit aus, beispielsweise:

```
/dev/console: character special (4/0)
/dev/hda9: block special (3/9)
/dev/null: character special (1/3)
/dev/ram0: block special (1/0)
/dev/tty: character special (5/0)
```

Ein *Block* Special File und ein *Character* Special File können durchaus dieselbe Major und Minor Number haben, ohne sich auf dasselbe Gerät zu beziehen: bei L*i*nux z.B. `/dev/ram1` und `/dev/mem` (beide 1,1).

Intern wird meist mit zwei getrennten *Tabellen* gearbeitet, in die die Adressen der zuständigen Routinen eingetragen werden. Die Major Number ist ein Index in die Tabelle für Block- bzw. in die für Character-Devices. (Intern können die Treiber-Funktionen auch den Parameter Minor Number über eine Tabelle auswerten.)

12.1.1.3 `mknod`

Ein Special File wird mit dem Kommando `mknod` (bzw. dem gleichnamigen Systemaufruf) angelegt:

```
mknod filename b major minor    # block special file
mknod filename c major minor    # character special file (buffered)
mknod filename u major minor    # character special file (unbuffered)
```

Mit „`-m` *mode*“ können die Zugriffsrechte angegeben werden, Standard ist `0666&~umask`.

Der passende Treiber im Kernel muss natürlich existieren! Um ein ganz neues Gerät in das System zu integrieren, benötigt man einen neuen Kernel oder zumindest ein neues Kernel-Modul.

Wenn man eine neue Platte/Partition integrieren möchte, benötigt man nicht nur eine neue Gerätedatei. Man muss mit `mkfs` ein Filesystem auf ihr anlegen und es schließlich mit `mount` in den Dateibaum einhängen (dazu später).

12.1.2 Typische Geräte

In der folgenden Auflistung sind die wichtigsten Geräte und Gerätedateien mit ihren L*i*nux-Namen aufgeführt. Die meisten Geräte existieren unter praktisch allen UNIX-Systemen, die Namensvergabe weicht eventuell etwas ab. So heißen die Treiber für SCSI-Platten `/dev/*sd*` (L*i*nux, SunOS), `/dev/hdisk*` (AIX), `/dev/?dsk*` (Solaris, IRIX, HP-UX, SCO), etc.

Die speziellen Angaben wie Major und Minor Numbers beziehen sich alle auf L*i*nux (die Major Numbers unter L*i*nux sind in `include/linux/major.h` definiert).

Genauere Angaben, insbesondere über die zugehörigen `ioctl`-Kommandos, findet man (hoffentlich) in den zugehörigen Man-Pages.

special file	ty	maj	min	Bedeutung

`/dev/ram*`	b	1	0^+	**RAM-Disks**
				Man muss auf einer RAM-Disk erst mit `mkfs` ein Filesystem erzeugen und sie dann mit `mount` in den Dateibaum einhängen, z.B.:
				`mkfs -t ext2 /dev/ram2 4096  # ca. 4 MByte`
				`mkdir /ram ; mount /dev/ram2 /ram`
				(genauer siehe Kapitel 13)

`/dev/initrd`	b	1	250	**Init-RAM-Disk** temporäre RAM-Disk beim Boot-Vorgang
`/dev/mem`	c	1	1	**Physischer Hauptspeicher** im Rohformat (hauptsächlich für Entwicklungszwecke gedacht)
`/dev/kmem`	c	1	2	**Virtueller Arbeitsspeicher** im Rohformat (hauptsächlich für Entwicklungszwecke gedacht)
`/dev/null`	c	1	3	**Nirvana-Device** geschriebene Daten werden weggeworfen, Lesen liefert End-of-File
`/dev/port`	c	1	4	**IO-Ports** im Rohformat (hauptsächlich für Entwicklungszwecke gedacht)

special file	ty	maj	min	Bedeutung
`/dev/zero`	c	1	5	**Zero-Device** Lesen liefert Nullbytes, geschriebene Daten werden weggeworfen
`/dev/full`	c	1	7	**Always Full Device** Schreiben erzeugt den Fehler `errno=ENOSPC` (*no space left on device*), Lesen liefert Nullbytes, Seek klappt immer, mit welchen Positionen auch immer.
`/dev/random`	c	1	8	**Zufallszahlen-Generator** für Bytes (nur L*i*nux) in die Zufallszahlen geht u.a. „Rauschen“ von diversen Gerätetreibern ein (um unangreifbar gegen Entschlüsselungsversuche zu sein) – blockiert, wenn gerade wenig Rauschen anliegt
`/dev/urandom`	c	1	9	**Zufallszahlen-Generator** für Bytes (nur L*i*nux) liefert immer Zufallszahlen (bei Rauschmangel ggf. von geringerer Qualität als `random`
`/dev/fd*`	b	2	0^+	**Diskettenlaufwerke** (FloppyDisk) die Minor-Number enthält codiert Laufwerksnummer, Controller-Nummer, Typ (3.5”, 5.25”, DD, HD). Der Kernel erkennt das Diskettenformat normalerweise automatisch, sodass man meist mit `fd0` und `fd1` auskommt. Es gibt weitere Dateien, die über die Minor Number direkt ein ganz spezielles Format ansteuern. Andere Namen: `/dev/rfd?` (AIX), `/dev/rdsk/*` (IRIX, HP-UX), `/dev/diskette` (Solaris)
`/dev/pty*`	c	2	0^+	**virtuelle Terminals** (control pseudo terminal) für Netzwerk-Zugriff, X-Window, etc.

`/dev/hd*`	b	3
		22
			0^+
			64^+
				Festplatten bzw. -Partitionen (IDE, nicht SCSI)
				Die Namen der ganzen Platten (roh, unpartitioniert) sind `hda` (erster Master), `hdb` (erster Slave), `hdc` (zweiter Master), `hdd` (zweiter Slave). Die Namen der Partitionen entstehen durch Anhängen der Partitionsnummern.
				Major Numbers: 3 (erster Controller), 22 (zweiter). Minor Numbers: 0,1,... (Partitionen auf dem Master), 64,65,... (auf dem Slave).
				Beispiel für das Anlegen solcher Special Files:
				`mknod -m 660 /dev/hdb b 3 64`
				`mknod -m 660 /dev/hdb1 b 3 65`
				`chown root.disk /dev/hdb*`

`/dev/ttyp*`	c	3	0^+	**virtuelle Terminals** (slave pseudo terminal) für Netzwerk-Zugriff, X-Window, etc.

special file	ty	maj	min	Bedeutung
/dev/ttys*	c	3	48^+	**serielle Schnittstellen** (als TTYs)
/dev/console	c	4	0	**Systemkonsole** (äquivalent mit /dev/tty0)
/dev/tty*	c	4	0^+	**Terminals** (bzw. Modems, serielle Schnittstellen) siehe Kapitel 12.2
/dev/tty	c	5	0	**das aktuelle Terminal**, (*kein Link* auf ein /dev/tty*!)
/dev/cua*	c	5	64^+	**serielle Schnittstellen** (serial unit)
/dev/lp*	c	6	0^+	**Drucker** (LinePrinter) an einer parallelen Schnittstelle
/dev/vcs*	c	7	0^+	**Bildschirmspeicher** der virtuellen Konsolen (nur L*i*nux) die Minor Numbers entsprechen denen bei /dev/tty*
/dev/vcsa*	c	7	128^+	**Bildschirmspeicher** der virtuellen Konsolen (nur L*i*nux) analog zu /dev/vcs, mit zusätzlichen Attributen
/dev/sd*	b	8		**Festplatten, SCSI-Disks** bzw. -Partitionen Die Minor Number ist 16·*drive*+*partition* (wobei *partition* so zu verstehen ist: 0=ganze Platte, 1–4 DOS primary, 5–8 DOS extended). Die Namen werden analog zu /dev/hd vergeben.
/dev/st*	c	9	0^+	**SCSI-Tapes (Streamer)** Alternativ-Name (Link) /dev/rmt*
/dev/st*	c	9	128^+	**SCSI-Tapes (Streamer)** ohne Rewind Die Minor Numbers 128,129,... sind No-Rewind-Versionen der Minor Numbers 0,1,... (alternativ /dev/nrmt*)
/dev/sr*	b	11	0^+	**SCSI CD-ROMs** andere Namen /dev/cd* (AIX, SCO), /dev/dsk/c* (Solaris, HP-UX)
/dev/mixer*	c	14	0	**Mixer** (Soundkarte)
/dev/sequencer	c	14	1	**Sequencer** (Soundkarte)
/dev/midi*	c	14	2	**MIDI-Schnittstelle** (z.B. Soundkarte)
/dev/dsp*	c	14	3	**Digital Signal Processor** (Soundkarte)
/dev/audio*	c	14	4	**Soundkarte** (FM-Synthese)
/dev/mouse	-	-	-	**Maus** meist ein Link (bei serieller Maus z.B. auf /dev/cua0)
/dev/tape	-	-	-	**Streamer** meist ein Link auf ein /dev/st?

12.2 Terminals unter UNIX

Unter Terminals wollen wir nicht nur physische Sichtgeräte verstehen (die über eine serielle Schnittstelle an den Rechner angeschlossen sind), sondern auch Modem-Verbindungen und virtuelle Geräte wie z.B. Fenster-Programme (xterm, cmdtool), die sich wie solche Geräte ver-

halten. Hier beschäftigen wir uns mit dem Einrichten und nachträglichen Einstellen der Treiber aller Geräte, die sich wie Terminals ansprechen lassen.

12.2.1 Geräte-Dateien

Die zu den TTYs gehörigen Special Files haben meist Namen wie `/dev/tty*`, siehe Kapitel 12.1. Welches aktuell benutzt wird, erfährt man durch das Kommando `tty`.

`/dev/tty`	**das aktuelle Terminal** (das Controlling Terminal der Session des aktuellen Prozesses)
`/dev/tty1`...	**virtuelle Terminals** In L*i*nux gibt es maximal 63 virtuelle Konsolen. Die ersten 6 werden beim Booten über `getty` geöffnet (s.u.) und bleiben permanent erhalten. Man kann mit `ALT-F1` bis `ALT-F6` zwischen ihnen hin- und herschalten. Mit `ALT-F7` kommt man zum X-Window-Bildschirm (falls es einen gibt), und von dort aus mit `CTRL-ALT-1` etc. zurück zu den virtuellen Terminals. Falls man eines dieser Terminals durcheinandergebracht haben sollte (Umschaltung auf Grafik-Zeichen o.ä.), kann man durch Ausgabe von `<ESC>C` einen Terminal-Reset auslösen, beispielsweise mit „`echo <CTRL-V><ESC>C`“.
`/dev/tty0`	**das aktuelle virtuelle Terminal**

Wenn ein neues Terminal angeschlossen werden soll (oder allgemeiner, wenn eine serielle Schnittstelle konfiguriert werden soll), ist folgendes zu tun:

- das Gerät (Terminal/Modem) physisch anschließen
- ein passendes Special File finden oder mit `mknod` neu anlegen
- bei Terminals einen zum Protokoll passenden `terminfo`-Eintrag finden oder anlegen (s.u.)
- die Terminal-Konfigurations-Datei (z.B. `/etc/gettytab` oder ggf. `/etc/inittab`, s.u.) ergänzen – dazu werden die Daten des Terminals wie Übertragungsrate etc. benötigt
- dem `init`-Prozess mitteilen, dass er die Terminal-Konfigurierung auf den neuesten Stand bringen soll (mit „`telinit q`“ oder in L*i*nux einfach „`init q`“)

12.2.2 `getty`-Aufrufe

Der `init`-Prozess startet beim Eintritt in Run Level 1 (oder manchmal 2) `getty`-Prozesse für alle angeschlossenen Terminals (definiert in `/etc/inittab`). Eine solche `getty`-Aufrufzeile sieht wie z.B. wie folgt aus:

```
getty tty2 9600n vt100
```

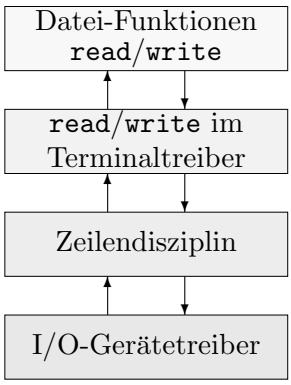
- `tty2` steht für den „Kanal“ und bezieht sich auf die Gerätedatei `/dev/tty2`.
- `9600` gibt die Übertragungsrate in Baud an (`n`=no parity, also kein Kontrollbit).
- `vt100` legt das Terminal-„Protokoll“ fest (s.u.). Wenn diese letzte Angabe fehlt, wird die Information aus einer Datei wie `/etc/ttytype` oder `/etc/gettydefs` geholt. Dort ist eingetragen, welches Protokoll standardmäßig auf einem Kanal verwendet werden soll.

Unter L*i*nux gibt es ein vereinfachtes `getty` namens `mingetty`, das nur für virtuelle Konsolen auf dem System-Bildschirm gedacht ist. Es entfallen Baud- und Protokoll-Angabe.

12.2.3 Zeilendisziplin

Terminal-Treiber werden vom System ein wenig anders gehandhabt als Treiber für andere Geräte. Das liegt daran, dass sie die direkte Schnittstelle zum menschlichen Benutzer darstellen. Während Daten von Platten, Bändern, etc. direkt übernommen werden können, ist es sinnvoll, dem Menschen noch Korrekturmöglichkeiten bei Eingaben und zusätzliche Steuermöglichkeiten einzuräumen.

Die Instanz zwischen dem Treiber des eigentlichen Geräts und der System-Schnittstelle heißt **Zeilendisziplin**.



- Sie zerlegt den Zeichenstrom der Eingabe in Zeilen und gibt nur komplette Zeilen statt einzelner Zeichen weiter.
- Sie behandelt Löschzeichen (Backspace, Delete, ...), bevor sie die Zeile weitergibt (ggf. auch Cursor-Steuerungszeichen).
- Sie erzeugt ein Echo aller Zeichen auf dem Bildschirm.
- Sie wandelt Tabulatorzeichen in die passende Anzahl von Spaces um.
- Sie sendet vorgegebene Signale beim Empfang spezieller Steuercodes wie `CTRL-C`, `CTRL-Z`, etc.
- Sie erlaubt aber das Umgehen der Spezialbehandlung, sodass Tastendrücke so weitergegeben werden, wie sie vom Benutzer erzeugt werden.

Wenn alle üblichen Effekte der Zeilendisziplin aktiviert sind, befindet sich das Terminal im „**kanonischen Modus**“. Der Modus, bei dem fast alle Effekt abgeschaltet sind, heißt „**Rohdatenmodus**“. Mit `ioctl`-Aufrufen lässt sich die Zeilendisziplin ziemlich genau einstellen (siehe 12.2.4).

Früher war die Zeilendisziplin ausgelagert in die Shell, in Editoren, etc. Um ihre Funktionalität allgemein zur Verfügung zu stellen, wurde sie schließlich in den Komplex der Terminaltreiber verschoben. Wenn Shells heutzutage auf bestimmte Tastendrücke anders reagieren wollen als gewohnt (z.B. Dateinamen-Expansion bei `TAB`, History), dann können sie mit `ioctl` die Zeilendisziplin verändern oder fast vollständig umgehen.

12.2.4 Terminal-Einstellungen

12.2.4.1 Allgemeine Funktionen

Es gibt einige einfach zu verwendende Bibliotheksfunktionen zu TTYs, deklariert in `unistd.h`.

Auf TTYs wird normalerweise über File-Deskriptoren lesend oder schreibend zugegriffen, z.B. über 0 und 1 des Session-Leader-Prozesses. Es gibt eine nützliche Funktion, um zu erfragen, ob ein File-Deskriptor sich auf ein TTY bezieht, oder ob es sich um ein echtes (Disk-)File handelt:

UNIX/DOS
int isatty**(int** fd**);** liefert 1, wenn sich `fd` auf ein TTY bezieht, sonst 0

Programme können hiervon abhängig machen, ob sie ihre Ausgabe seitenweise tätigen (TTY) oder nicht (File), wie es `man` und `more` tun.

Mit der Funktion `ttyname` erfährt man den Pfadnamen der zu einem TTY zugehörigen Gerätedatei in `/dev`:

UNIX
char ***ttyname(int** fd**);** wenn sich `fd` auf ein TTY bezieht, wird als (statischer) String der Pfadname von dessen Gerätedatei zurückgegeben, ansonsten der Null-Pointer

Ein Programm mit der Zeile „`puts(ttyname(0));`“ ist damit äquivalent zum Shell-Kommando `tty`.

Die Funktion `getpass` ermöglicht das Einlesen eines Passworts, ohne dass man explizit die Steuersequenzen für das Aus- und Einschalten des Bildschirm-Echos bemühen müsste:

UNIX/DOS
char ***getpass(const char** ***prompt);** gibt `prompt` aus, liest verdeckt ein Passwort ein und gibt einen statischen String darauf zurück. Es wird `/dev/tty` verwendet, kein umgeleiteter Eingabekanal!

Für die TTYs gibt es außerdem folgende nützliche `ioctl`-Kommandos (siehe auch Seite 322):

UNIX
int ioctl**(int** fd**,** TIOCGWINSZ**, struct** winsize ***w);** get winsize, fragt die aktuelle Fenstergröße ab und speichert sie in einer `winsize`-Struktur (s.u.)
int ioctl**(int** fd**,** TIOCSWINSZ**, const struct** winsize ***w);** set winsize, setzt die Fenstergröße entsprechend `*w` neu, dem Session-Leader wird das Signal `SIGWINCH` (window change) gesendet

Die verwendete Struktur sieht wie folgt aus:

```
struct winsize
 {
   unsigned short ws_row, ws_col;        // Größe in Zeichen
   unsigned short ws_xpixel, ws_upixel;  // Größe in Pixeln
 };
```

Beispiel: Folgendes Programm zentriert den Text „`xterm`“ und reagiert dabei auf Größenänderungen des Fensters. Der Kürze halber wird die `xterm`-Steuersequenz `clear-cup` zur Cursor-Positionierung direkt verwendet, weswegen das Programm so nur in einem `xterm` läuft. Wie man Protokoll-unabhängige Programme schreibt, wird in Abschnitt 12.2.5 behandelt.

```
#include <sys/ioctl.h>
#include <signal.h>
#include <stdio.h>
```

```
static void sigwinch_handler(int s)
 {
   struct winsize ws;
   char buf[32];
   ioctl(0,TIOCGWINSZ,&ws);
   sprintf(buf,"\033[H\033[2J\033[%d;%dH""LINUX",
      ws.ws_row/2,(ws.ws_col-5)/2);
   write(1,buf,strlen(Buffer));
   signal(SIGWINCH,sigwinch_handler);
 }

int main()
 {
   sigwinch_handler(SIGWINCH);
   for (;;) sleep(1000);
 }
```

12.2.4.2 Einstellungen von `C` aus

Die Terminal-Treiber haben eine große Anzahl von Einstellungsmöglichkeiten, um sie an die physischen Gegebenheiten anzupassen: Übertragungsgeschwindigkeit auf der Leitung, Parity (Kontroll-Bits), Stop-Bits, Übertragungsprotokoll, Echo-Verhalten, Signalerzeugung und natürlich Benutzerwünsche.

Mit den angegebenen Funktionen wird das Verhalten des Treibers *für die aktuelle Sitzung* geändert, nicht das grundlegende Verhalten.

Die wesentlichste *numerische* Einstellung ist die Übertragungsrate. Es gibt eine Familie von Funktionen `cf-set/get-i/o-speed` (z.B. `cfgetospeed`), mit denen man die Baudrate für Ein- und Ausgabe abfragen und steuern kann.

Die anderen Einstellungen sind fast alle *boolescher* Natur. Sie sind in der Struktur `termio` bzw. `termios` codiert (in `termios.h` definiert, in L*i*nux ausgelagert z.B. nach `asm/termbits.h`).

```
#define NCCS 19                     // number of control characters (POSIX: 11)
typedef unsigned int  tcflag_t;     // terminal control type
typedef unsigned char cc_t;         // control character type

struct termios
 {
   tcflag_t c_iflag;                // input mode flags
   tcflag_t c_oflag;                // output mode flags
   tcflag_t c_cflag;                // control mode flags
   tcflag_t c_lflag;                // local mode flags
   cc_t c_line;                     // line discipline
   cc_t c_cc[NCCS];                 // control characters
 };
```

Die einzelnen Einträge werden weiter unten besprochen. Das Lesen bzw. Schreiben der Struktur erfolgt mit den folgenden `ioctl`-Aufrufen bzw. äquivalenten POSIX-Funktionen:

UNIX
int ioctl**(int** fd**, TCGETA, struct termio** ***termio_p**); liest die Attribute des mit dem File-Deskriptor `fd` verbundenen TTY in die Struktur, auf die `termio_p` zeigt
int ioctl**(int** fd**, TCSETAF, const struct termio** ***termio_p**); setzt für das mit `fd` verbundene TTY die in `*termio_p` festgelegten Attribute
int tcgetattr**(int** fd**, struct** termios ***termios_p**); analog zu `ioctl(,TCGETA)`
int tcsetattr**(int** fd**, int** opt_act**, struct** termios ***termios_p**); wie `ioctl(,TCSETA)`, zusätzlich legt `opt_act` fest, *wann* das Umstellen geschehen soll (`TCSANOW`=sofort)

Ein Rückgabewert 0 bedeutet wieder okay. `tcsetattr` liefert allerdings 0, wenn *zumindest ein* Wert gesetzt werden konnte. Am besten prüft man den Erfolg mit einem zusätzlichen `tcgetattr` nach.

Die `flag`-Felder enthalten etwa 100 boolesche Variablen (und einige längere Einträge), die man mit Hilfe von Bitmasken aus `termios.h` abfragen und verändern kann. Alle Flags findet man im Manual-Eintrag zu `termios`. Im Folgenden werden nur einige Flags beispielhaft herausgegriffen.

c_iflag	ICRNL	Carriage-Return wird in der Eingabe in LineFeed umgewandelt
	IMAXBEL	ein voller Eingabepuffer löst einen Piepton aus (kein POSIX)
	ISTRIP	des achte Bit der Eingabezeichen wird auf 0 gesetzt (ASCII)
	IXON	mit den STOP- und START-Zeichen (s.u.) kann die Eingabe zeitweise angehalten werden
c_oflag	OCRNL	Carriage-Return wird in der Ausgabe in LineFeed umgewandelt
	OXTABS	Tabulatoren werden durch passend viele Leerzeichen ersetzt
c_cflag	CREAD	es können Zeichen gelesen werden
	CSTOPB	es sollen *zwei* Stopbits (statt einem) gesendet werden
	HUPCL	hang-up-close, wenn der letzte Prozess die Gerätedatei schließt, wird die Verbindung abgebrochen
	PARODD	odd parity (Flag nicht gesetzt ⇒ even parity)
c_lflag	ECHO	Tastendrücke sollen auf dem Bildschirm ausgegeben werden
	ICANON	kanonischer Modus, zeilenorientiert
	XCASE	Großbuchstaben werden in Kleinbuchstaben umgewandelt

Beispiel: Folgendes Programm schaltet das aktuelle Terminal in den Rohdatenmodus, wozu wir nur das Flag `ICANON` löschen müssen. Außerdem unterdrücken wir das automatische Echo durch Löschen des `ECHO`-Flags.

```
int tty;
struct termios oldtty;

void sigcatch(int sig)
 {
   tcsetattr(tty,TCSANOW,&oldtty);
   exit(0);
 }
```

```
int main()
 {
   struct termios newtty;

   if ((tty=open("/dev/tty",O_RDWR,0))<0) { perror("open"); exit(1); }

   if (tcgetattr(tty,&oldtty)) { perror("tcgetattr"); exit(1); }
   newtty=oldtty;
   newtty.c_lflag&=~(ICANON|ECHO);
   newtty.c_cc[VMIN]=16;

   signal(SIGINT,sigcatch);
   if (tcsetattr(tty,TCSANOW,&newtty)) { perror("tcsetattr"); exit(1); }

   for (;;)
    {
      unsigned char buffer[32];
      int j;
      int n=read(0,buffer,16);
      for (j=0;j<n;++j) printf(" %02x ",buffer[j]);
      putchar('\n');
      for (j=0;j<n;++j)
       {
         int c=buffer[j];
         if (c==0x1b) printf("ESC ");
         else printf("  %c ",isprint(buffer[j])?buffer[j]:' ');
       }
      putchar('\n');
    }
 }
```

Wir geben immer 16 gelesene „rohe“ Tastendrücke als Hex-Codes und als ASCII-Zeichen aus (falls sie einem druckbaren Zeichen entsprechen). Cursor-, Funktionstasten etc. bekommen wir als Escape-Kombinationen zurück, beispielsweise:

```
 72  65  74  75  72  6e  0a  74  61  62  09  75  70  1b  5b  41
  r   e   t   u   r   n       t   a   b       u   p ESC   [   A
 64  6f  77  6e  1b  5b  42  6c  65  66  74  1b  5b  44  72  69
  d   o   w   n ESC   [   B   l   e   f   t ESC   [   D   r   i
 67  68  74  1b  5b  43  64  65  6c  65  74  65  1b  5b  33  7e
  g   h   t ESC   [   C   d   e   l   e   t   e ESC   [   3   ~
```

Das Programm kann mit `CTRL-C` abgebrochen werden und restauriert dann die alten Terminal-Einstellungen.

12.2.4.3 Spezielle Eingabezeichen

Terminals werden, abgesehen von Steuersequenzen, im Normalbetrieb durch spezielle Eingabezeichen (*control characters*) beeinflusst: `CTRL-C` sendet das `SIGINT`-Signal an den Leader der verbundenen Session, `CTRL-Q` hält die Ausgabe an, `CTRL-S` lässt sie weiterlaufen, etc.

Welche Tastencodes diesen Zeichen entsprechen sollen, ist im Array `c_cc` in **`struct`** `termios` festgelegt. Es gibt in `termios.h` symbolische Konstanten wie `VINTR`=0, die Indizes in das Array sind. Beispielsweise ist normalerweise `c_cc[VINTR]`=3 (entspricht `CTRL-C`).

Es gibt meist die unten aufgeführten spezielle Eingabezeichen. Die mit * versehenen sind nicht in POSIX definiert. Die zugehörigen symbolischen Konstanten haben ein führendes `V`.

Name	Bedeutung	typischer Wert
DISCARD*	discard output	CTRL-O
EOF	end-of-file	CTRL-D
EOL	end-of-line	
EOL2*	alternate eol	SHIFT-RETURN
ERASE	backspace	CTRL-H
INTR	interrupt (send SIGINT)	CTRL-C
KILL	erase line	CTRL-U
LNEXT*	literal text (s.u.)	CTRL-V
QUIT	quit (send SIGQUIT)	CTRL-\
REPRINT*	reprint all	CTRL-R
START	resume output	CTRL-S
STOP	stop output	CTRL-Q
SUSP*	suspend (send SIGSTOP)	CTRL-Z
WERASE*	word backspace	CTRL-W

`LNEXT` bewirkt, dass die nächste Tastenkombination ohne Interpretation übernommen wird. Beispielsweise kann man in der Shell durch „`var=<CTRL-V><CTRL-H>`“ ein Backspace-Zeichen in die Variable `var` schreiben. Das Kommando „`echo aaa${var}bbb`“ gibt dann „`aabbb`“ aus.

Beispiel: Folgendes Programm gibt die aktuelle Belegung der wichtigsten Control Characters für die (Standard-Eingabe) aus. Mit der Umlenkung „<“ kann also ein beliebiger lesbarer Kanal abgefragt werden.

```
#include <iostream.h>
#include <iomanip.h>
#include <termios.h>

int main()
 {
   termios tios;

   static struct { char *name; int index; } flags[]=
    { "EOF", VEOF, "ERASE", VERASE, "INTR", VINTR, "KILL", VKILL,
      "LNEXT", VLNEXT, "QUIT", VQUIT, "STOP", VSTOP, "START", VSTART,
      "SUSP", VSUSP, "WERASE", VWERASE
    };

   tcgetattr(0,&tios);

   for (int i=0;i<sizeof(flags)/sizeof(flags[0]);++i)
    {
      cout << setw(7) << flags[i].name << ' ';
```

```
      int c=tios.c_cc[flags[i].index];
      if (c==0) cout << "---";
      else if (c<32) cout << "CTRL-" << (char)(c+'@');
      else if (c<127) cout << (char)c;
      else cout << "ASCII " << c;
      cout << '\n';
    }
  }
```

12.2.4.4 Einstellungen von der Shell aus

Das Kommando `stty` dient zum Anpassen einzelner Terminal-Einstellungen von der Shell aus. Die Aufrufe sollten am besten beim Einloggen vorgenommen werden, z.B. in `.profile` o.ä.

Die Bezeichnungen der Flags sind als Options-Namen übernommen worden, beispielsweise „`-parodd`“.

Beispiel: Ohne Parameter gibt `stty` die wichtigsten aktuellen Einstellungen aus, mit „`-a`“ *alle*, z.B.:

```
speed 9600 baud; rows 14; columns 80; line = 0;
intr = ^C; quit = ^\; eof = ^D; kill=^U; lnext=^V;
...
-parodd -ignbrk -imaxbel
...
```

Bei dieser Einstellung löst also ein `CTRL-C` das Signal `SIGINT` aus. Das können wir leicht auf `CTRL-I` ändern:

```
stty intr <CTRL-V><CTRL-I>    # LNEXT, es erscheint: stty intr ^i
```

Mit „±`echo`“ kann man das Echo-Verhalten ändern, d.h. die Wiedergabe gedrückter Tasten als Bildschirmzeichen unterbinden (und wieder einschalten). Das ist für die Eingabe von Passwörtern sehr sinnvoll:

```
echo -n "Passwort: "
stty -echo
read password
stty echo
```

`stty` liest Einstellungen immer von dem Terminal, das an die Standard-Eingabe gekoppelt ist. Mit einer Eingabe-Umlenkung kann man daher Informationen über andere Terminals als das aktuelle einholen:

```
stty -a < /dev/console
```

So kann man allerdings keine fremden Terminal-Einstellungen *ändern*.

Der Name eines `termio`-Flags als Option schaltet dieses Flag ein, mit vorangestelltem ‘`-`’ ab. Die Namen aller Flags sind im Manual-Eintrag zu `termios` aufgeführt. Die folgende Tabelle führt nur einige Beispiele auf.

n	Übertragungsrate auf `n` Baud setzen
`speed`	Übertragungsrate ausgeben
`parodd`	wie `PARDODD`, Odd Parity, `-parodd` steht für Even Parity
`echo`	wie `ECHO`, Echo einschalten (Tastendrücke erscheinen auf dem Bildschirm)
`sane`	„vernünftige“ Normaleinstellungen

12.2.5 Terminal-Protokolle

Die eigentlichen Terminal-Treiber antworten auf druckbare Zeichen mit seiner Ausgabe, und sie reagieren auf die oben angegebenen Kontrollzeichen. Um sie zu komplexen Aktionen zu bewegen, z.B. den Bildschirminhalt komplett zu verändern (löschen, scrollen) oder verschiedene Modi zu verwenden (fett, Farbe), setzt man Steuerzeichen bzw. Steuersequenzen ein (aus mehreren Zeichen, meist mit `ESC`=ASCII 27 eingeleitet). Die Definition dieser Steuerzeichen ist von Treiber zu Treiber (eventuell völlig) unterschiedlich.

Beispiel: Das American National Standards Institute (ANSI) hat einen Satz von Steuersequenzen definiert, die von vielen Terminal-Treibern (als Untermenge) übernommen wurden. Unter MS-DOS gibt es z.B. einen Treiber `ANSI.SYS`, den man in `CONFIG.SYS` mit `DEVICE=ANSI.SYS` einbinden und danach vernünftige Cursor-Steuerung auf der Text-Konsole vornehmen kann.

Alle Sequenzen beginnen mit `CTRL` oder mit `ESC` und einer eckigen Klammer ‘`[`’. Nützliche Sequenzen sind z.B.

`<ESC>[H`	Home (Cursor nach links oben)
`<ESC>[J`	Bildschirm ab Cursor löschen
`<ESC>[`*f*`m`	Modus setzen, z.B. *f*=1 fett, *f*=0 normal

Folgendes `C++`-Programm löscht – auf ANSI-kompatiblen Terminals! – den Bildschirm und schreibt in die erste Zeile „**Fett**schrift“. Das `ESC` ist oktal angegeben ($(33)_8 = (27)_{10}$).

```
int main()
 {
   cout << "\033[H\033[J\033[1mFett\033[0mschrift\n";
 }
```

Eine Sammlung aller relevanten Terminal-Eigenschaften und Definitionen der Steuersequenzen heißt **Terminal-Protokoll**. Der Name eines solchen Protokolls heißt **Protokoll-Typ**. Typische Typen sind `ansi` (s.o.), `tek` (Tektronix), `vt100` (DEC Video Terminal), `xterm`, `dumb`, `unknown`, `amiga`, `ibmpc`, `sun`.

Die meisten Protokolle sind in großen Teilen kompatibel mit `vt100` (einem der ersten Standards). Alle verfügbaren Protokolle kann man mit dem Kommando `toe` (*table of entries*) auflisten lassen. Das jeweils aktive Protokoll findet man in der Umgebungsvariablen `$TERM` wieder.

12.2.5.1 Protokoll-Definitionen

Früher (von BSD her kommend) gab es ein einziges großes Textfile `/etc/termcap` (*terminal capabilities*, Terminal-Fähigkeiten), in dem *alle* verfügbaren Protokolle aufgeführt waren. Leider

wächst das File im Laufe der Zeit stark, und der Zugriff auf die Einträge dauert immer länger. Außerdem ist es umständlich, nicht mehr benötigte Protokolle daraus zu löschen.

Aus Kompatibilitätsgründen mit alten Programmen existiert das File heute noch. Man bevorzugt aber eine andere Form von Datenbank: Im Verzeichnis `/usr/lib/terminfo` (der „*terminal capability data base*“) sind die Protokolle für alle dem System bekannten Terminals in Form von Definitionsdateien abgelegt (z.B. `/usr/lib/terminfo/x/xterm`). Zum Übergang erstellt das Kommando `captoinfo` aus einem `termcap`-Eintrag eine `terminfo`-Datei.

In den Dateien ist beschrieben, welche Fähigkeiten überhaupt zur Verfügung stehen, über welche Steuersequenzen diese angesprochen werden, wie das Terminal zu initialisieren ist, etc. Die Dateien sind aus Performancegründen *binär* und nicht direkt lesbar.

Aus diesen Dateien holen sich Programme wie `vi` und die `C`-Bibliothek `curses` die nötigen Informationen zur Bildschirmsteuerung.

Normalerweise wird ein neues Terminal mit Hilfe eines *Textfiles* beschrieben, das dann mit `tic` (*terminal info compiler*) in die *Binärform* überführt wird. Das Kommando `infocmp` dient eigentlich zum Vergleich der Fähigkeiten zweier Terminals, kann aber auch zur Rekonstruktion der Textform verwendet werden.

„`infocmp xterm`“ (oder „`infocmp -I`“, wenn `$TERM`=`xterm`) liefert abgekürzt folgendes:

```
# Reconstructed via infocmp from file: /usr/lib/terminfo/x/xterm
xterm|vs100|xterms|xterm terminal emulator (X Window System),
am, bce, km, mir, msgr, xenl,
colors#8, cols#80, it#8, lines#24, pairs#64,
...
bel=^G, blink=\E[5m, bold=\E[1m, cr=^M,
clear=\E[H\E[2J, home=\E[H,
...
```

Die Dateien stellen eine Liste Einträge dar, die durch Kommas (in `/etc/termcap` durch Doppelpunkte) getrennt sind. Der erste liefert Alternativnamen und eine Kurzbeschreibung. Danach gibt es drei Arten von Einträgen:

- einzelne Worte, Flags (`bool`, beschreiben Terminal-Fähigkeiten)
- Einträge mit `#`, Konstanten-Definitionen (z.B. als Initialisierungs-Angaben)
- Einträge mit `=`, String-Definitionen für Steuersequenzen
  (darin steht „`\E`“ für ESC und ‘`^`’ für CTRL, Parameter werden mit ‘`?`’ und ‘`%`’ angegeben)

Mit „`man termcap`“ erhält man ausführliche Informationen über alle erlaubten Einträge. Einige interessante sind in der Tabelle weiter unten angegeben.

Das Kommando „`infocmp xterm ansi`“ listet zeilenweise die Unterschiede in den Einträgen der beiden Terminal-Definitionen auf, z.B.:

```
blink: NULL, '\E[5m'.
```

Die Zeile sagt uns, dass man mit ANSI-Steuersequenzen blinken kann, mit `xterm` nicht.

		dumb	ibmpc	vt100	ansi	amiga	xterm	linux
bel	bell/beep	^G	^G	^G	^G	^G	^G	^G
blink	blink			\E[5m$<2>	\E[5m	\E[7;2m		\E[5m
bold	bold on		^M^^	\E[1m$<2>	\E[1m	\E[1m	\E[1m	\E[1m
clear	clear scr.		^L^K	\E[H\E[J$<50>	\E[H\E[J	\E[H\E[J	\E[H\E[2J	\E[H\E[J
cr	crg return	^M		^M	^M	^M	^M	^M
cub1	csr back		^]	^H	\E[D	\E[D	^H	^H
cub	csr $n\times$back			\E[%p1%dD	\E[%p1%dD	\E[%p1%dD	\E[%p1%dD	
cud1	csr down	^J	^J	^J	\E[B	\E[B	^J	^J
cuf1	csr fwd		^\	\E[C$<2>	\E[C	\E[C	\E[C	\E[C
cuu1	csr up		^^	\E[A$<2>	\E[A	\E[A	\E[A	\E[A
dch1	delete char				\E[P	\E[P	\E[P	\E[P
dl1	delete line				\E[M	\E[M	\E[M	\E[M
ed	erase disp.			\E[J$<50>	\E[J	\E[J	\E[J	\E[J
home	csr home		^K	\E[H	\E[H	\E[H	\E[H	\E[H
ht	horiz. tab			^I	\E[I	^I	^I	^I
rev	rev. video			\E[7m$<2>	\E[7m	\E[7m	\E[7m	\E[7m

In Anwendungen sollte man im Normalfall nicht direkt Steuersequenzen verwenden, da man so entweder auf ein Terminal festgelegt ist oder selbst die Datenbank auswerten muss. Es gibt Schnittstellen wie `tput` und die `curses`-Bibliothek, die einem fast alle Operationen in anderer Form zur Verfügung stellen (s.u.).

Beachte: Einige Terminal-Programme wie `xterm` besitzen *zusätzliche Steuersequenzen*, die nicht zum Protokoll gezählt werden. In `xterm` beginnen sie mit `<ESC>]`. Beispielsweise kann man mit einer Sequenz wie der folgenden jederzeit einen beliebigen X-Font (hier „`vga`“) einstellen:

```
echo "<CTRL-V><ESC>]50;vga<CTRL-V><CTRL-G>"
```

Mit folgender Sequenz kann man einen beliebigen Text *text* in die Titelzeile des `xterm-Fensters` setzen:

```
echo "<CTRL-V><ESC>]2;text<CTRL-V><CTRL-G>"
```

Bei den SUN-Entsprechungen `cmdtool` und `shelltool` lautet diese Sequenz:

```
echo "<CTRL-V><ESC>]1text<CTRL-V><ESC>"
```

12.2.5.2 Textformatierung in der Shell

Das Kommando `tput` dient verschiedenen Zwecken:

- zur Initialisierung eines Terminals
- zur Abfrage von Terminal-Fähigkeiten, die sich auf `terminfo` beziehen
- zur Verwendung von `terminfo`-Steuerungen in der Shell

Mit der Option „`-T`*type*“ kann der zu verwendende Terminal-Typ angegeben werden, sonst wird `$TERM` verwendet.

Ansonsten wird als Argument eine Terminal-Fähigkeit angegeben. `tput` gibt auf die Standard-Ausgabe die passende Steuersequenz aus:

```
tput home          # Cursor in linke obere Ecke
tput clear         # Fenster ab Cursor löschen (hier also komplett)
tput bold          # Fettschrift
tput rev           # revers
echo "I'm home!"
tput sgr0          # Normalschrift, alle Attribute abschalten
```

In jedem `tput`-Aufruf kann normalerweise nur eine Terminal-Fähigkeit verwendet werden. Mit „`-S`“ und liest `tput` zeilenweise Anweisungen von der Standard-Eingabe. Also kann man wie folgt mehrere Aktionen auslösen:

```
tput -S <<!
home
clear
!
```

Man kann sich die Steuersequenz, die `tput` ausgibt, natürlich auch gut in einer Variablen merken und später in `echo` verwenden. Das ist schneller (weil nicht jedesmal in `terminfo` nachgeschaut werden muss) und übersichtlicher.

Folgende Zeilen geben oben links „**fett** nicht fett **wieder fett**“ aus und hinterlassen das Terminal im Normalzustand.

```
clrhome=‘tput home; tput clear‘
bold=‘tput bold‘
normal=‘tput sgr0‘
echo "$clrhome${bold}fett$normal nicht fett$bold wieder fett$normal"
```

Mit den Cursor-Sequenzen `cuu`, `cud`, `cub`, `cuf` (Cursor up, down, backward, forward) und der absoluten Cursor-Positionierung „`cup` *row col*“ kann man schon Bildschirm-Masken etc. realisieren. Zeilen und Spalten werden dabei wie üblich von links oben an gezählt.

Beispiel: Das folgende Shell-Skript fragt nach Benutzernamen und Passwort. Die Eingabefelder werden in der Mitte des Bildschirms angezeigt.

```
clrhome=‘tput home ; tput clear‘
bold=‘tput bold‘
normal=‘tput sgr0‘
top=‘tput lines‘; top=‘expr "( $top - 4 ) / 2"‘; top=‘expr $top‘
left=‘tput cols‘; left=‘expr "( $left - 20 ) / 2"‘; left=‘expr $left‘

while true do
   echo $clrhome
   tput cup $top $left
      echo Name:
   tput cup ‘expr $top + 1‘ $left
      echo Passwort:
   tput cup $top ‘expr $left + 10‘
      read name
   tput cup ‘expr $top + 1‘ ‘expr $left + 10‘;
      stty -echo; read passwort; stty echo
   if test $name = axel -a $passwort = linux ; then break; fi
```

```
    tput cup `expr $top + 3` $left
       echo -n "${bold}sorry...$normal "
    sleep 5
done
```

12.2.5.3 Textformatierung in C

Für die normale *Verwendung* des Terminals inklusive der Sequenzen zur Cursor-Steuerung gibt es eine Standard-Bibliothek namens `curses`.

- In `C` muss man `curses.h` inkludieren und die Bibliothek mit „`-lcurses`“ zum Programm dazulinken. Die ursprünglichen Versionen (aus BSD und SVR4) sind nicht frei als Quelltext vertreibbar. Unter L*i*nux wird daher eine kompatible neue Version `ncurses` (für new) verwendet. Sie muss mit „`-lncurses`“ gelinkt werden.

- Man sollte `curses`-Funktionen nicht mit normalen `stdio`-Funktionen o.ä. mischen! Die Cursor-Buchführung von `curses` wird sonst durcheinandergebracht. Es gibt spezielle Versionen wie `printw` und `scanw`.

- Vor der Verwendung der ersten echten `curses`-Funktion *muss* die Funktion `initscr`, ganz am Schluss `endwin` aufgerufen werden. Nach jeder Änderung (nach jeder Ausgabe) sollte `refresh` aufgerufen werden, damit die Änderungen wirksam werden:

curses
`WINDOW *initscr(void);` Initialisierung, gibt Zeiger auf den Gesamtbereich zurück
`int endwin(void);` Beendigung des `curses`-Modus, Zurücksetzen der TTY-Einstellungen auf die alten Werte
`int refresh(void);` Übertragung der internen Änderungen in `stdscr` auf den physischen Bildschirm (geschieht automatisch vor einer *Eingabe*)

  Alle `curses`-Funktionen liefern, wenn nicht anders angegeben, im Fehlerfall `ERR`, sonst einen anderen Wert (meist `OK`).

- Man kann mehrere Fensterbereiche innerhalb eines Terminals verwalten. Diese Bereiche dürfen sich aber nicht überlappen. Wir wollen uns hier nicht ausführlich damit beschäftigen.

curses
`WINDOW *newwin(int nlines, int ncols, int begin_y, int begin_x);` legt ein neues Fenster mit `nlines` Zeilen und `ncols` Spalten an, mit der linken oberen Ecke in (`begin_y`,`begin_x`). Es wird ein Handle zurückgegeben, mit dem das Fenster später angesprochen werden kann.
`int delwin(WINDOW *win);` löst ein Fenster auf

Die meisten im Folgenden angegebenen Funktionen gibt es in einer Version mit vorangestelltem 'w' und beziehen sich dann nicht auf das ganze Terminal, sondern nur auf ein Fenster (erster Parameter `WINDOW*`). `stdscr` kann dabei immer für das ganze Terminal verwendet werden.

- Es werden automatisch folgende drei externe Variablen verwaltet:

```
extern int LINES;    // Anzahl Zeilen
extern int COLS;     // Anzahl Spalten
extern int TABSIZE;  // Tabulator-Breite
```

  Wie wir in 12.2.4 gesehen haben, wird bei Größenänderung ein `SIGWINCH`-Signal geschickt. Eventuell werden `LINES` und `COLS` nicht automatisch angepasst, sodass man das in einer Handler-Routine selbst übernehmen muss.

- Die nützlichsten einfachen Ausgabe-Funktionen sind folgende:

curses
int clear**(void);** Clear-Home
int move**(int** y**, int** x**);** Cursor an die Position (y, x) bewegen (0 ≤y<LINES, 0 ≤x<COLS)
int addch**(const** chtype c**);** ein Zeichen ausgeben
int addstr**(const char *s);** einen **char**-String ausgeben
int printw**(const char *format, ...);** formatierte Ausgabe à la `printf`
int mvaddch**(int** y**, int** x**, const** chtype c**);** kombiniertes `move`/`addch`
int mvaddstr**(int** y**, int** x**, const char *s);** kombiniertes `move`/`addstr`
int mvprintw**(int** y**, int** x**, const char *format, ...);** kombiniertes `move`/`printw`

  Die Ausgaben finden jeweils an der Cursor-Position statt, und der Cursor wird der Ausgabe entsprechend weiterbewegt.

  Der Typ `chtype` kann ein anderer als **`char`** sein. Meistens wird ein 32-Bit-Typ (wie **`long`**) verwendet, von dem die untersten 16 Bit für ein Zeichen des `C++`-Typs **`wchar_t`** (Unicode) verwendet werden. Die oberen 16 Bit stehen für Attributinformationen (Fettschrift, Farbe) zur Verfügung.

  Man kann auch einfache Zeichen-Grafiken für Umrahmungen etc. erstellen. Je nach Terminal werden dann echte Grafik-Zeichen (`ACS`, alternate character set) oder ASCII-Ersatz-Zeichen benutzt. Alle entsprechenden Konstanten (sie beginnen mit `ACS_`) kann man in `curses.h` nachschauen oder mit „`man curs_addwch`“ erfragen, beispielsweise:

`ACS_HLINE`	horizontal line	`ACS_ULCORNER`	upper left corner
`ACS_VLINE`	vertical line	`ACS_LARROW`	left arrow

Bemerkung: Die meisten üblichen X-Window-Fonts enthalten *nicht* die notwendigen Grafikzeichen. `xterm` stellt aber einige interne Fonts zur Verfügung, die geeignet sind (über `CTRL`-mittlere Maustaste).

- Zeichenattribute werden als Bitmasken-Konstanten des Typs `NCURSES_ATTR_T` definiert:

`A_NORMAL`	keine besonderen Attribute
`A_STANDOUT`	hervorgehoben (meist fett oder invers)
`A_UNDERLINE`	unterstrichen
`A_REVERSE`	invers
`A_BLINK`	blinkend
`A_DIM`	mit halber Intensität
`A_BOLD`	fett
`A_COLOR`	in Farbe, der Farbcode muss dazugeodert werden

  Bei Terminals mit eingeschränkten Darstellungsmöglichkeiten werden allerdings typischerweise `BOLD`, `STANDOUT` und `UNDERLINE` durch `REVERSE` ersetzt und `BLINK`, `DIM` und `COLOR` ignoriert.

  Einige Funktionen beeinflussen die Attribute für die nachfolgenden Ausgeben:

	curses
`int attrset(NCURSES_ATTR_T a);`	Attribut `a` einschalten
`int attron(NCURSES_ATTR_T a);`	Attribut `a` hinzuschalten
`int attroff(NCURSES_ATTR_T a);`	Attribut `a` wegschalten

- Es gibt z.B. folgende Eingabefunktionen:

curses
`int getch(void);` ein Zeichen einlesen
`int getstr(char *buf);` String einlesen (Vorsicht bei Pufferüberlauf!)
`int wgetnstr(WINDOW *scr, char *buf, int n);` String einlesen, Maximallänge `n`
`int scanw(const char *format, ...)` formatierte Eingabe wie `scanf`, ab Cursor-Position
`int mvscanw(int y, int x, const char *format, ...)` kombiniertes `move`/`scanw`

  Spezielle Tasten wie Cursor- und Funktionstasten liefern einen speziellen Code, der symbolischen Konstanten aus `curses.h` entspricht. Maus-Ereignisse werden über den Tastencode `KEY_MOUSE` gemeldet und können danach genauer ausgewertet werden (siehe z.B. „`man curs_mouse`“).

`KEY_DOWN`	`KEY_RIGHT`	`KEY_F1`
`KEY_UP`	`KEY_HOME`	`KEY_F2`
`KEY_LEFT`	`KEY_BACKSPACE`	`KEY_DC`

Die `stdio`-Funktionen arbeiten mit einem Puffer und normalerweise zeilenweise, d.h. das Programm erhält nicht bei jedem Tastendruck sofort eine Rückmeldung. Bei `curses` kann man dieses Verhalten ein- und ausschalten, Auch das Echo-Verhalten kann variiert werden.

	curses
`int cbreak(void);`	einzelne Zeichen werden gelesen (default)
`int nocbreak(void);`	es wird zeilenweise in einen Puffer gelesen
`int echo(void);`	Echo einschalten
`int noecho(void);`	Echo ausschalten

- Außerdem gibt es einige Funktionen zum Löschen/Kopieren von Zeilen und rechteckigen Bildschirmausschnitten innerhalb des Terminal-Bereichs:

curses
`int deleteln(void);` Cursorzeile löschen (Zeilen darunter scrollen nach oben)
`int insertln(void);` Zeile an Cursorposition einfügen (Zeilen darunter scrollen nach unten)
`int clrtoeol(void);` Löschen ab Cursor-Position bis zum Zeilenende
`int clrtobot(void);` Löschen ab Cursor-Position bis zum Bildschirmende
`int copywin(const WINDOW *src, WINDOW *dest, int stop, int sleft, int dtop, int dleft, int dbottom, int dright, int overlay);` kopiert einen Rechteckbereich – als Quell- und Zielfenster kann man z.B. einfach `stdscr` angeben.

Beispiel: So füllt man das gesamte Terminal mit dem Schriftzug „`Linux`“ (mit fettem ‘`i`’):

```
#include <curses.h>

int main()
 {
   int i;
   initscr();
   mvaddch(0,0,'L'); addch(A_BOLD|'i'); addstr("nux");
   for (i=5;i<COLS;i+=5) copywin(stdscr,stdscr,0,0,0,i,0,i+4,0);
   for (i=1;i<LINES;++i) copywin(stdscr,stdscr,0,0,i,0,i,COLS-1,0);
   refresh();
   sleep(2);
   endwin();
 }
```

Beispiel: Das folgende Programm imitiert das Shell-Skript aus dem letzten Abschnitt. Es wird mit einem Fenster gearbeitet, das in der Mitte des Bildschirms plaziert wird. Die Koordinaten der weiteren Operationen sind dann relativ zur oberen linken Ecke des Fensters und vereinfachen sich daher erheblich. Außerdem lässt sich so die Funktion `box` verwenden, die (wenn auf dem jeweiligen Terminal möglich) das Fenster umrahmt.

```
int main()
 {
   WINDOW *win;
   initscr();
   win=newwin(6,24,(LINES-6)/2,(COLS-24)/2);

   for (;;)
    {
      char name[12],passwd[12];
      wclear(win); wattrset(win,A_BOLD);
      box(win,ACS_VLINE,ACS_HLINE); wattrset(win,A_NORMAL);
      mvwaddstr(win,1,1,"Name:");
      mvwaddstr(win,2,1,"Passwort:");
      wmove(win,1,11); wgetnstr(win,name,10);
      wmove(win,2,11); noecho(); wgetnstr(win,passwd,10); echo();
      if (strcmp(name,"axel")==0&&strcmp(passwd,"linux")==0) break;
      wattron(win,A_BOLD); mvwaddstr(win,4,2,"-- ACCESS DENIED --");
      wrefresh(win);
      sleep(5);
    }

   delwin(win);
   clear(); addstr("Hallo Axel!"); refresh();
   // ...
   endwin();
 }
```

12.3 UNIX-Systemverwaltung

12.3.1 Benutzerverwaltung

12.3.1.1 /etc/passwd

Benutzer werden in UNIX dadurch definiert, dass sie zusammen mit einigen Angaben als eine Zeile in die Datei `/etc/passwd` eingetragen werden. Diese Datei wird als die Benutzer-Datenbank (*user database*) bezeichnet. Sie ist für alle Benutzer lesbar, aber für niemanden außer für den Systemverwalter beschreibbar.

Ein Eintrag entspricht der **struct** `passwd`, die in `pwd.h` definiert ist. In einigen UNIXen (z.B. SVR4) gibt es zusätzliche Einträge.

```
struct passwd
 {
   char *pw_name;              // Username         POSIX
   char *pw_passwd;            // Paßword          SVR4, kein POSIX
   uid_t pw_uid;               // User ID          POSIX
   gid_t pw_gid;               // Group ID         POSIX
   char *pw_gecos;             // Real name        SVR4, kein POSIX
   char *pw_dir;               // Home directory   POSIX
   char *pw_shell;             // Shell program    POSIX
 };
```

`pw_gecos` kann irgendwelche Kommentare enthalten – üblicherweise steht hier aber der Benutzername, ggf. mit Adresse (das Kommando `finger` gibt diese Informationen aus). `pw_dir` ist das Heimatverzeichnis des Benutzers im Dateisystem (siehe unten), `pw_shell` ist seine bevorzugte Shell.

In Textform werden die Einträge (wie in UNIX üblich) durch ':' getrennt, beispielsweise:

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/bin/bash
...
axel:x:500:100:ich:/home/axel:/bin/bash
```

Früher war dort schon das Passwort angegeben – nicht im Klartext, sondern mit Hilfe der Funktion `crypt` (s.u.) in eine immer 13-stellige Kombination verschlüsselt. Seit SVR3 findet man an der entsprechenden Stelle nur ein „`x`“, und die eigentlichen Passwort-Daten sind nach `/etc/shadow` (normalerweise nur vom Administrator lesbar) ausgelagert (mit dem Kommando `pwconv` kann man automatisch Passwörter aus `/etc/passwd` nach `/etc/shadow` verschieben):

```
root:M1oN16HN/0i9g:10333:0:10000::::
bin:*:8902:0:10000::::
...
axel:FnVFQUsjone4U:10333:0:10000::::
```

Ein Eintrag entspricht der C-Struktur **`struct`** `spwd` (Shadow Password), die in `shadow.h` definiert ist (nicht in POSIX vorgeschrieben):

```
struct spwd
 {
   char *sp_namp;              // Username
   char *sp_pwdp;              // verschlüsseltes Passwort
   time_t sp_lstchg;           // letzte PW-Änderung
   time_t sp_min;              // minimale Zeit (Tage) zwischen PW-Änderungen
   time_t sp_max;              // maximale Zeit zwischen PW-Änderungen
   time_t sp_warn;             // Zeit, bis PW erlischt (Warnungen!)
   time_t sp_inact;            // Zeit nach PW-Erlöschen, bis Account erlischt
   time_t sp_expire;           // Zeit ab dem 1.1.1970, bis Account erlischt
   unsigned long sp_flag;      // bislang unbenutzt
 };
```

Ist *nichts* als Passwort eingetragen, hat der Benutzer noch kein Passwort und kann sich (Vorsicht!) allein mit seinem Namen einloggen. Steht dort ein „`*`“ (oder „`NONE`“), so soll es gar kein Passwort geben, was für bestimmte externe Programme sinnvoll sein kann (Server-Abfragen). Diese Benutzer haben dann aber auch keinerlei spezielle Zugriffsrechte.

Der Administrator kann einen Benutzer dadurch zeitweise sperren, dass er „Sperrzeichen“ in das Passwort einträgt, nämlich Zeichen, die keine Ausgabezeichen von `crypt` sein können (also nicht in „`a-zA-Z0-9./`“, z.B. Space und Semikolon). So braucht nicht gleich der ganze Eintrag in `/etc/passwd` gelöscht zu werden und kann später wiederhergestellt werden.

Wenn im Feld `lstchg` 0 eingetragen ist, *muss* der Benutzer beim nächsten Einloggen als erstes ein neues Passwort eingeben.

Anstatt die Dateien `passwd` und `shadow` selbst zu editieren, kann der Administrator auch das Kommando `passmgmt` (*password management*) oder das Kommando `passwd` mit speziellen Optionen verwenden (näheres dazu per `man`).

- `passwd` ist üblicherweise nicht lesegeschützt, damit die *anderen Angaben* frei zur Verfügung stehen (sonst müssten Programme wie `chsh`, `finger`, etc. alle `root`-Rechte haben, was zu gefährlich wäre).
- Wenn die Passwörter verschlüsselt, aber allgemein lesbar (wie früher in `passwd`) liegen, können Hacker mit Hilfe eines größeren Vorrats an verschlüsselten Wörtern versuchen, den zugrundeliegenden Algorithmus zurückzuentwickeln.
- Auf diese Weise kam die doppelte Sicherung durch Verschlüsselung und eine privilegierte Datei zustande.

12.3.1.2 `crypt`

Die Transformationsfunktion für Passwörter heißt `crypt` (nicht zu verwechseln mit dem Kommando `crypt` zur Textverschlüsselung auf anderer Basis). Sie ist in `unistd.h` deklariert:

UNIX
`char *crypt(const char *key, const char *salt);` Verschlüsselt den String `key` unter Berücksichtigung zusätzlicher Angaben in `salt` (ein String der Länge 2 aus „`a-zA-Z0-9./`“ (s.u.). Es werden nur die untersten 7 Bit der ersten 8 Zeichen von `key` beachtet.

Das Verfahren, das `crypt` verwendet, basiert auf dem *Data Encryption Standard* (DES), der zur Verschlüsselung ganzer Texte gedacht ist. Die Texte werden dazu in Blöcke der Größe acht Zeichen zerlegt, und jeder Block wird mit dem Schlüssel verrechnet. `crypt` wendet dieses Verfahren einfach auf einen Block von 8 Null-Bytes an.

- `crypt` ist im praktischen Einsatz erstaunlich sicher, obwohl oft (siehe L*i*nux) der Quellcode der Funktion öffentlich zugänglich ist. Die vorgenommene Transformation ist nicht umkehrbar, die entsprechende Abbildung nicht injektiv. Der `login`-Prozess verschlüsselt mit `crypt` das eingegebene Passwort und überprüft, ob das Ergebnis mit dem Eintrag in `passwd` bzw. `shadow` übereinstimmt.
- Wenn man `crypt` überlisten will, ist das einzig gangbare Verfahren, Passwörter mit einem Wörterbuch zu „raten“, mit `crypt` zu verschlüsseln und das Ergebnis mit dem Passwort-Dateien zu vergleichen („*dictionary attack*“).

  Um einen solchen Angriff zu erschweren, wird das DES-Verfahren 25 Mal hintereinander angewandt. Außerdem ist die Verschlüsselung absichtlich nicht ganz DES-kompatibel, um den Einsatz von Hardware für diesen Standard zu verhindern.

  Die `crypt`-Funktion wurde von Robert Morris und Ken Thompson für die PDP-11-Version von UNIX entwickelt. Auf diesem Rechner dauerte eine einzige Verschlüsselung eine Sekunde, sodass damals ein Dictionary Attack unmöglich war. (Das Wörterbuch des damaligen UNIX-Spell-Checkers hätte drei Jahre zur Verschlüsselung gebraucht).

- Der Parameter `salt` beschreibt (mit zwei Zeichen aus einem Vorrat von 64) eine Zahl zwischen 0 und 4095, die das Ergebnis der Transformation leicht verändert.

  Jedesmal, wenn ein Passwort kreiert oder verändert wird, wird ein „zufälliges“ Salt (meist über die Systemzeit) berechnet und für `crypt` verwendet. Die beiden Zeichen, die Salt entsprechen, werden als die ersten beiden der 13 Zeichen in der Passwort-Datei abgelegt. `login` liest diese beiden Zeichen bei der Passwort-Überprüfung aus und verwendet sie ebenfalls bei `crypt`.

  Auf diese Weise ist beispielsweise nach außen nicht mehr erkennbar, dass ein Benutzer auf mehreren Rechnern dasselbe Passwort besitzt (oder mehrere Benutzer dasselbe). Außerdem bindet es die Passwortsuche an die existierenden Einträge und erschwert Hackern noch ein wenig mehr das Leben.

- Heutzutage wäre es ohne weiteres möglich, ein komplettes Wörterbuch mit allen 4096 Salt-Möglichkeiten zu verschlüsseln und auf CD o.ä. abzuspeichern. Das ist mit ein Grund für das zusätzliche Einrichten einer `shadow`-Datei.

  Wenn man als Administrator den Quelltext von `crypt` zugänglich hat, kann man zur Sicherheit Veränderungen vornehmen, beispielsweise mehr als 25 Iterationen, andere Interpretation der Salts, etc. Dann bekommt man dann allerdings in Netzwerken (NIS) Probleme, die verschlüsselte Passwörter austauschen.

12.3.1.3 `/etc/group`

Die Gruppen werden in der „Group Database“ `/etc/group` festgelegt, beispielsweise wie folgt:

```
root:x:0:root
bin:x:1:root,bin,daemon
...
users:x:100:
```

Ein Eintrag entspricht einer **`struct`** `group`, definiert in `grp.h`:

```
struct group
 {
   char *gr_name;              // Gruppenname          POSIX
   char *gr_passwd;            // Password              SVR4, kein POSIX
   gid_t gr_gid;               // Gruppen-ID           POSIX
   char **gr_mem;              // Liste der Mitglieder  POSIX
 };
```

Die Liste der Mitglieder darf auch leer sein. Die Gruppenzugehörigkeit ergibt sich ja allein aus dem Eintrag in `/etc/passwd`. Dieser Eintrag dient nur der Übersicht.

Wenn der Benutzer temporär seine Gruppenzugehörigkeit ändern möchte, kann er das mit dem Kommando „`newgrp` *groupname*“ tun. Aus diesem Grund gibt es auch ein Passwort je Gruppe, das der Benutzer in solchen Fällen angeben muss. Wiederum ist es heutzutage nicht mehr in `/etc/group` angegeben, sondern nach `/etc/gshadow` ausgelagert:

```
root:*:root:root
bin:*:root:root
...
users:*:root:root
```

Ein Eintrag entspricht einer **struct** sgrp (auch in shadow.h):

```
struct sgrp
 {
   char *sg_name;              // Gruppenname
   char *sg_passwd;            // Gruppen-Passwort
   char **sg_adm;              // Liste der Administratoren
   char **sg_mem;              // Liste der Mitglieder
 };
```

In SVR4 gibt es „Zusatz-GIDs“ (*supplementary GIDs*), sodass ein Benutzer mehreren Gruppen gleichzeitig angehören kann. In /etc/group erscheint er dann einfach in mehreren Zeilen. Man benötigt dann aber zusätzliche Systemfunktionen wie getgroups, um alle GIDs geliefert zu bekommen.

12.3.1.4 Passwort-Hilfsfunktionen

Es gibt Bibliotheks-Funktionen, mit denen man über eine UID oder einen Login-Namen an eine passwd-Struktur bzw. an eine group-Struktur kommen kann:

UNIX
struct passwd *getpwnam(**const char** *name); liefert einen Zeiger auf eine statisch angelegte passwd-Struktur, die mit Informationen über den Benutzer mit dem Login-Namen name gefüllt ist
struct passwd *getpwuid(uid_t uid); analog, aber der Benutzer wird über seine UID angegeben
struct group *getgrnam(**const char** *name); liefert einen Zeiger auf eine statisch angelegte group-Struktur, die mit Informationen über die Gruppe mit dem Namen name gefüllt ist
struct group *getgrgid(gid_t gid); analog, aber die Gruppe wird über ihre GID angegeben

Beispiel: Folgendes Programm liefert Informationen über den Benutzer und seine Gruppe:

```
// include: iostream unistd pwd grp

int main()
 {
   struct passwd *P=getpwuid(getuid());
   cout << "Name:           " << P->pw_name << '\n';
   cout << "Kommentar:      " << P->pw_gecos << '\n';
   cout << "Home-Directory: " << P->pw_dir << '\n';
   cout << "Std.-Shell:     " << P->pw_shell << '\n';

   struct group *G=getgrgid(getgid());
```

```
  cout << "Gruppe:          " << G->gr_name << " [";
  for (char **N=G->gr_mem;*N!=0;++N) cout << ' ' << *N;
  cout << " ]\n";
 }
```

Wenn man `root` ist, seine Gruppe aber mit „`newgrp bin`“ ändert, erhält man folgende Ausgabe:

```
Name:           root
Kommentar:      root
Home-Directory: /root
Std.-Shell:     /bin/bash
Gruppe:         bin [ root bin daemon ]
```

Nach „`newgrp`“ erscheint dann in der letzten Zeile das erwartete „`root [root]`“.

In SVR4 kann man sukzessive alle Einträge (Entries) von `/etc/passwd` auslesen – mit den zusätzlichen Funktionen `setpwent` (Start), `getpwent` (einen Eintrag lesen) und `endpwent` (Ende). Analog gibt es `setgrent`, `getgrent` und `endgrent` für Gruppen.

Beispiel: So kann man *alle* Benutzer mit ihren Gruppen ausgeben:

```
// include iostream.h pwd grp

int main()
 {
  struct passwd *P;
  setpwent();
  while ((P=getpwent())!=0)
   {
    struct group *G=getgrgid(P->pw_gid);
    cout << P->pw_name << " (" << G->gr_name << ")\n";
   }
  endpwent();
 }
```

12.3.1.5 `utmp` und `wtmp`

Diese Dateien heißen genauer meist entweder `/var/run/utmp` bzw. `/var/log/wtmp` oder liegen beide in `/etc`. Hier wird abgespeichert, welche Benutzer gerade im System sind (`utmp`) bzw. von wann bis wann in der letzten Zeit eingeloggt waren (`wtmp`).

Beides sind Binärdateien und nicht direkt anzeigbar; Kommandos wie `who`, `finger`, `users` und `last` (s.u.) werten sie aus. Die Dateien sind üblicherweise von jedermann lesbar.

Die Daten in `utmp` werden eingetragen von `login`, `ftp` und Terminal-Emulatoren (`telnet`). Eventuell tragen einige solche Programme auch keine Informationen ein, sodass `utmp` unvollständig sein kann.

`utmp` wird bei jedem Systemstart neu angelegt, während `wtmp` alle Information aufbewahrt und ständig wächst. Wenn es zu groß wird, kann der Administrator es (einfach mit „`>/etc/wtmp`“) auf die Länge 0 zurücksetzen.

Ein Eintrag entspricht (binär!) einer **struct** utmp, die in utmp.h definiert, aber ziemlich systemabhängig ist und öfter auch noch geändert wird. Das Verwenden älterer Utilities zum Lesen/Schreiben in diese Dateien ist daher problematisch. Die L*i*nux-Version ist momentan folgende:

```
struct utmp
 {
   short   ut_type;       // login-type (Boot-Time, Login-Proc, User-Proc, ...)
   pid_t   ut_pid;        // pid des Login-Prozesses
   char    ut_line[12];   // TTY-Name
   char    ut_id[4];      // ID in der inittab
   char    ut_user[8];    // username, not null-term
   char    ut_host[16];   // hostname for remote login
   int     ut_exit;       // exit status
   struct  timeval ut_tv; // Entrags-Zeitpunkt
   long    ut_session;    // session ID
   long    ut_addr;       // Remote-IP-Adresse
 };
```

Zum Lesen der Einträge kann man aber komfortabel das Kommando „`last` *username*“ verwenden. Normalerweise liest es aus `wtmp`, mit der Option „`-f /var/run/utmp`“ (o.ä.) kann man es aber umlenken.

Das Ausgabeformat lässt sich mit Optionen variieren. Beispielsweise könnte es so aussehen:

```
root    ttyp2      :0.0           Sun Nov  8 14:20   still logged in
root    ttyp2      :0.0           Sun Nov  8 13:50 - 13:52  (00:01)  ...
```

Ohne Optionen listet `last` *alle* Zeiten auf, in denen der Benutzer *username* jemals eingeloggt war, in umgekehrter zeitlicher Reihenfolge. Durch die Option „`-`*n*“ kann man sich aber auch nur die letzten *n* Einträge ansehen. Um die Ausgabe einzuschränken, kann man auch ein TTY angeben („`last root console`“) oder einen Remote-Hostname („`last -h wmpi03 axel`“).

Es gibt einen Pseudo-Benutzer `reboot`, der sich bei jedem Systemstart einzuloggen scheint. Mit

```
last reboot | grep reboot | wc -l
```

erfährt man also, wie oft man das System bereits gebootet hat.

12.3.1.6 lastlog

In der Datei `/var/log/lastlog` (oder `/usr/adm/lastlog` o.ä.) wird für jeden Benutzer die Zeit abgelegt, zu der er sich zuletzt eingeloggt hat.

Es handelt sich um eine Binärdatei, die Platz für alle existierenden Benutzer auf dem System hat. Das `finger`-Kommando wertet diese Datei aus:

```
Login: axel                             Name: Axel Rogat
Last login Sat Feb 13 16:26 (CEST) on tty1           ...
```

Auf einigen Systemen bekommt man diese Zeit beim nächsten Einloggen angezeigt:

```
login: axel
Password:
Last login: Sat Feb 13 16:26:13 on tty1.
No mail.
```

12.3.2 syslog

BSD-UNIX und Abkömmlinge (auch L*i*nux) besitzen ein System zur Verwaltung von Log-Meldungen namens `syslog` (*system logger*). Ursprünglich diente es der Aufzeichnung des Wegs, den E-Mails (bei `sendmail`) intern gehen. Seine Möglichkeiten wurden aber seither stark erweitert. Ein Dämon namens `syslogd` kümmert sich im Hintergrund um die Verwaltung.

Auf diese Weise wird die Handhabung von Nachrichten der Subsysteme wie Mail („`return to sender: cannot send message within 5 days`“), vom Kernel („`FAILED SU`“, „`system halted by root`“) und User-Programmen vereinheitlicht und vereinfacht.

Jedes Programm kann eine Log-Nachricht erzeugen:

UNIX
`void openlog(char *ident, int option, int facility);` öffnet eine Verbindung des Prozesses zu `syslogd`, braucht nur aufgerufen zu werden, wenn `ident` gesetzt werden soll: ein Identifikations-String (Programmname. o.ä.), wird an den Anfang jeder Meldung gesetzt. `facility` unterscheidet zwischen verschiedenen Klassen von Meldungen/Programmen (ohne `openlog`-Aufruf ist die Klasse `USER`).
`void syslog(int priority, char *format, ...);` sendet eine Log-Message. `format` und `...` sind wie bei `printf` zu verstehen. Zusätzlich ist '`%m`' in `format` erlaubt und steht für die aktuelle Fehlermeldung (wie bei der `error`-Funktion) (Priorität = Wichtigkeit der Meldung).
`void closelog(void)` schließt die Verbindung zu `syslogd`.

Folgende Konstanten sind dazu z.B. in `syslog.h` definiert:

`facility`	
`LOG_KERN`	Kernel-Meldungen
`LOG_USER`	Meldungen normaler Benutzer-Programme
`LOG_MAIL`	Meldungen vom Mailsystem
`LOG_LPR`	Drucker-Meldungen
`LOG_CRON`	Meldungen von `cron` und `at`
`LOG_SYSLOGD`	interne Meldungen

`priority`	
`LOG_EMERG`	emergency, Notfall, das System steht (fast)
`LOG_ALERT`	Alarm, Reaktion des Administrators dringend erforderlich
`LOG_CRIT`	kritischer Systemzustand
`LOG_ERR`	Fehlermeldung
`LOG_WARNING`	Warnmeldung
`LOG_NOTICE`	wichtiger Hinweis
`LOG_INFO`	allgemeine Information
`LOG_DEBUG`	Debugging-Information

Verschiedene Klassen von Nachrichten können an verschiedenen Orten abgelegt werden. Das wird in der Konfigurationsdatei `/etc/syslog.conf` festgelegt, beispielsweise:

```
kern.warn;*.err          /dev/tty10
mail.*                   -/var/log/mail
news.*                   -/var/log/news
```

In der ersten Spalte werden Facility und Priorität angegeben, in der zweiten eine Log-Datei (oder natürlich ein Gerät per Gerätedatei).

12.3.3 `init` und `/etc/inittab`

Wie wir schon öfter gesehen haben, wird beim Systemstart nach dem eigentlichen Booten zunächst der Prozess 0 erzeugt. Er ist meist „idle“ und verbraucht im Fall, dass keine anderen Prozesse lauffähig sind, die überflüssige Systemzeit.

Bevor er sich der Untätigkeit hingibt, erzeugt er aber noch den ersten echten Prozess PID 1, den „init“-Prozess. Dieser wird der Vorgänger aller anderen Prozesse. Die Aktionen von `init` hatten wir in Abschnitt 6.3 bereits kurz betrachtet.

Der **Run Level** (Ausführungsgrad) von UNIX legt fest, in welcher Art das System gerade betrieben wird. Je nach Run Level sind nur bestimmte Prozesse erlaubt, bzw. andere Prozesse zwingend erforderlich. In der Datei `/etc/inittab` ist festgelegt, welche Prozesse in welchem Run Level laufen sollen. Dafür ist `init` verantwortlich.

- Der Run Level ‘`S`’ (oder ‘`s`’) liegt im eingeschränkten Single-User-Modus vor. `init` initialisiert *nicht* alle angeschlossenen Terminals, sondern ruft `/bin/sh` auf der Konsole `/dev/console` auf.
- Der Run Level 0 ist entweder der Zustand bei einem System-Halt oder steht für einen eingeschränkten Single-User-Modus, bei dem nicht alle Terminals initialisiert werden, sondern nur `/dev/console`.
- Der Run Level 1 steht üblicherweise schon für einen Multi-User-Modus, aber ohne Netzwerkaktivitäten.
- Die Run Level 2 bis 6 stehen für den normalen Multi-User-Modus. Dazu sind diverse Hintergrund-Aktivitäten vonnöten (Login-Dämon, diverse Netzwerk-Dämonen, etc.) Die genauen Unterschiede innerhalb dieser Gruppe sind systemabhängig. Beispielsweise könnte ab Level 3 eine grafische Benutzeroberfläche (wie X-Window) gestartet werden.
- ‘`I`’ ist kein echter Run Level, sondern steht für den Zustand des Systems während der Initialisierung.

Das Kommando `runlevel` gibt den Run Level vor der letzten Änderung und den aktuellen aus, z.B. „`I 2`“.

Mit „`init` *level*“ kann der Super-User den Run Level verändern. Das Kommando schickt dann dem `init`-Prozess ein entsprechendes Signal. Er muss dann ggf. Prozesse killen (die in niedrigeren Run Levels nicht mehr laufen sollen, zunächst `INTR`, nach 20 Sekunden `KILL`) oder neue Prozesse anlegen (die in höheren Run Levels laufen müssen) – beides laut `/etc/inittab`.

Die Einträge in `/etc/inittab` haben das Format

```
id:runlevels:action:process
```

`id` ist eindeutiges Label. `runlevels` sind die Run Levels, für die die angegebene Aktion ausgeführt werden soll. `process` beschreibt den zugehörigen Prozess.

Man kann statt Run Levels auch die Buchstaben `abc` angeben. Die entsprechenden Aktionen werden dann nur beim Aufruf des `init`-Kommandos ausgeführt, etwa „`int ab`“.

Eine Aktion `initdefault` definiert den Run Level, der nach dem Booten betreten werden soll. Ansonsten sind für `action` sind z.B. folgende Angaben gültig:

`once`	der Prozess wird einmal gestartet
`wait`	der Prozess wird einmal gestartet, und init wartet auf sein Ende
`respawn`	der Prozess wird nach seinem Tod neu gestartet (vor allem `getty`)
`boot`	der Prozess wird genau einmal beim Booten gestartet
`bootwait`	wie `boot`, aber `init` wartet auf das Ende des Prozesses
`sysinit`	wie `boot`, der Prozess wird aber vor allen `boot`-Aktionen gestartet
`ctrlaltdel`	nur L*i*nux, bei Drücken von `CTRL-ALT-DEL` auf der Systemkonsole

Es folgt ein Ausschnitt aus einer typischen SuSE-L*i*nux-`/etc/inittab`:

```
id:2:initdefault:

si:I:wait:/sbin/init.d/boot

l0:0:wait:/sbin/init.d/rc 0

ls:S:wait:/sbin/init.d/rc S
~~:S:respawn:sbin/sulogin

ca::ctrlaltdel:/sbin/shutdown -r -t 4 now

1:123:respawn:/sbin/mingetty --noclear tty1
2:123:respawn:/sbin/mingetty tty2
...
```

Die eigentlichen Aktivitäten beim Run-Level-Wechsel werden bei der SuSE-Distribution auf `rc` verlagert. Man sieht, dass die TTYs gekillt werden, wenn das System in den Single-User-Modus gefahren wird.

12.3.4 Weitere Dateien in `/etc`

Im Verzeichnis `/etc` haben sich im Lauf der Entwicklung diverse Konfigurationsdateien für das System angesammelt, von denen wir ja schon einige kennen. Nicht alle unten aufgeführten müssen in jedem System vorhanden sein, und es kann diverse systemspezifische geben:

`/etc/at.allow`	falls diese Datei existiert, dürfen nur die darin eingetragenen Benutzer `at`-Kommandos absetzen.
`/etc/at.deny`	wenn `at.allow` nicht existiert, aber diese Datei, dürfen die hier eingetragenen Benutzer `at` *nicht* benutzen.

/etc/crontab	Definitionen für cron, einen Dämon, der Kommandos zu sich regelmäßig wiederholenden Zeitpunkten automatisch ausführt.
/etc/group	Gruppen-Definitionsdatei, siehe 12.3.1.
/etc/gshadow	Gruppen-Schattendatei, siehe 12.3.1.
/etc/inittab	Definitionsdatei für init, siehe 12.3.3.

/etc/magic	**Dateityp-Definitionen** für das Kommando file. Anhand der ersten Bytes versucht file die geläufigen Programmcode-, Quellcode-, Text-, Grafik-, Sound-Formate etc. zu erkennen und gibt eine entsprechende Meldung aus, z.B.:
	`missfont.log: ASCII text`
	`pinfo: Bourne shell script text`
	`forever.cpp: C program text`
	`a.out: ELF 32-bit LSB executable, Intel 80386, ...`
	`op10.tex: LaTeX document text`
	`whatsthis: data`
	Das Format der Definitionen ist auf der Man-Page von file beschrieben. Der Abschnitt zu LaTeX-Dokumenten beginnt wie folgt:
	`0 string \documentstyle LaTeX document text`
	`0 string \documentclass LaTeX document text`

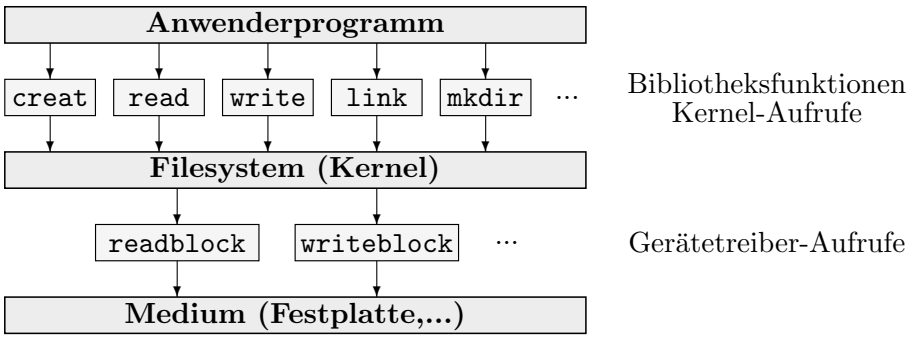
/etc/motd	**Message of the Day**, Tagesmeldung. Nach einem erfolgreichen Login wird diese Datei angezeigt. Sie kann vom Administrator verwendet werden, um aktuelle Meldungen und Hinweise zu verbreiten, ohne an alle Benutzer Mails schreiben zu müssen.
/etc/nologin	Das Vorhandensein dieser Datei **sperrt** alle Benutzer außer root **aus**, gedacht hauptsächlich für eine Zeit der Systemwartung. Normalbenutzer erhalten beim Login-Versuch den Inhalt dieser Datei angezeigt.
/etc/passwd	Passwort-Datei, siehe 12.3.1.
/etc/printcap	**Printer Capabilities**; diese Datei ist das Analogon zu /etc/termcap für Drucker.
/etc/profile	**Login-Skript** für alle Benutzer und alle Shells, wird vor den möglicherweise vorhandenen persönlichen Skripten ausgeführt.
/etc/protocols	Beschreibung der zur Verfügung stehenden TCP/IP-**Protokolle**.
/etc/rc.d	**System-Initialisierungsdatei**, wird von init beim Wechseln des Run Levels konsultiert. Je nach init-Version steht hier auch ein Directory oder eine Datei pro Run Level (rc4.d).
/etc/securetty	Wenn diese Datei vorhanden ist, sind Superuser-Logins nur von den darin eingetragenen „sicheren“ Terminals möglich.
/etc/shadow	Passwort-Schattendatei, siehe 12.3.1.
/etc/shells	Liste aller verfügbaren Shells. (chsh überprüft hiermit, ob das angegebene Kommando als Shell zulässig ist.
/etc/utmp	Logdatei für aktuell im System befindliche Benutzer, siehe 12.3.1.
/etc/wtmp	unbeschränkte Logdatei für System-Logins, siehe 12.3.1.

13 Dateisysteme

Als Dateisystem (Filesystem) bezeichnet man normalerweise eine Kernel-Schnittstelle zwischen Benutzer-Prozessen (mit Datei-Operationen) und den Treibern von peripheren Geräten jeglicher Art (mit hardwarenahen Operationen).

Im engeren Sinn ist der Begriff Dateisystem eingeschränkt auf block-orientierte Geräte wie Platten. Die Datei-Schnittstellen lassen sich aber sinnvoll auf seriell betriebene Geräte (Terminals, Drucker, etc.) übertragen. Ein Dateisystem kann daher eine einheitliche Schnittstelle zu *allen* Geräten (und noch anderen Diensten) im System herstellen.

Beim Zugriff auf ein Medium über ein Filesystem durchläuft eine Anfrage also (mindestens) zwei Schichten (eventuelle Pufferungen sind hier nicht berücksichtigt):



13.1 Block-Format

In sehr frühen Systemen wurden Dateien **zusammenhängend** auf das Medium geschrieben. Der Benutzer musste dazu eine Schätzung für die maximale Länge angeben. Das System suchte nach einem genügend großen Freiraum und reservierte ihn.

Die kontinuierliche Abspeicherung bietet große Geschwindigkeitsvorteile, da die Adressrechnung bei Random Access sehr einfach ist. Andererseits hatte man enorme Nachteile:

- Wenn die Datei zu groß wurde, musste eine neue, größere angelegt und die alte dorthinein kopiert werden.
- Wenn die Datei kleiner war als angenommen, wurde Platz vergeudet.

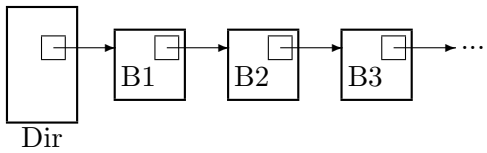
Aus diesen Überlegungen entwickelte sich eine dynamische Belegung des Mediums. Die Daten wurden dazu in **Blöcken** organisiert, z.B. 256 Byte bis 4 KByte.

- Die Datei- und Dateibaum-Struktur muss auf die Blockstruktur auf dem Medium abgebildet werden. Auch diese Abbildung und die entsprechende Verwaltung und Wartung übernimmt das Dateisystem.
- Die Blockgröße ist hardwaremäßig festgelegt (durch den Controller oder die Platte). Wenn sie für das jeweilige Dateisystem ungünstig ist (s.u.), belegen die Systeme keine einzelnen Blöcke, sondern „**Cluster**“ (Blockgruppen, Allocation Units), immer aus einer festen Anzahl hintereinanderliegender Blöcke bestehend.

Beispielsweise arbeiten MS-DOS und Windows auf großen Platten mit sehr großen Clustern, da ihre Dateisysteme zu wenig Bits für die Speicherung von Blocknummern vorgesehen haben. Eine Datei der Länge 1 Byte kann durchaus 32 KByte Platz belegen.

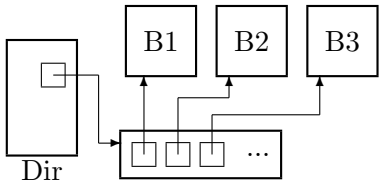
Die verschiedenen Systeme unterscheiden sich u.a. darin, wie sie abspeichern, welche Blöcke zu einer Datei gehören, und in welcher Reihenfolge:

Die Blöcke können in einer Liste miteinander **verkettet** werden. Das verlangsamt den Zugriff bei Random Access erheblich. Außerdem ist der Rest der Datei verloren, wenn ein Block nach einem Medienfehler nicht mehr lesbar ist.



Es kann eine **Tabelle** mit den Blocknummern geführt werden. Das ist die üblichere Lösung, die – in ganz unterschiedlicher Ausprägung – unter UNIX und MS-DOS verwendet wird.

Es gibt gemischte Systeme (AMIGA), die dann zwar mehr Platz für beide Strukturen benötigen, aber im Fall von Hardware-Fehlern Dateien leichter retten können.



Viele Systeme benutzen angebrochene Blöcke nicht anderweitig, d.h. eine Datei der Größe 1 Byte belegt einen ganzen Block (z.B. 4 KByte). Es gibt Dateisysteme, die ökonomischer vorgehen.

13.2 Pufferung

Im Vergleich zu Hauptspeicherzugriffen sind Plattenzugriffe um Größenordnungen langsamer. Normale Operationen weisen aber das Merkmal „Lokalität“ auf, d.h. es werden zusammenhängende Datenteile angesprochen, z.B. große Dateien sequentiell gelesen, Directories ausgelesen, Dateien aus demselben Directory gelesen, etc.

Das System versucht daher, die Verwaltungsstrukturen (Blocktabellen, Directories, etc., s.u.) zu puffern und so lange wie möglich im Hauptspeicher zu belassen. Die Mechanismen sind denen von Hauptspeicher-Caches sehr ähnlich.

Der entsprechende Filesystem-Teil wird in UNIX *buffer cache* genannt. In Windows NT ist der *Virtual Block Manager* dafür zuständig. Das MS-DOS-Filesystem hat direkt keine solche Vorrichtung; stattdessen kann man ein Utility namens `SMARTDRV` verwenden.

- Üblicherweise werden nicht einzelne Blöcke, sondern ganze Sektoren von der Platte gelesen und zwischengepuffert. Beim Schreiben werden (logisch) aufeinanderfolgende Blöcke so auf das Medium verteilt, dass sie nachher gut wieder hintereinander eingelesen werden können.

- Außerdem wird mit dem Schreiben gewartet, bis z.B. eine Kopfbewegung notwendig ist, d.h. es werden Blöcke gesammelt, die für denselben Sektor bestimmt sind (*Write-Behind*).

- Wenn gerade keine Plattenaktivität stattfindet, kann das Filesystem auch versuchen, zu „ahnen“, welche Blöcke als nächstes angefordert werden könnten, und diese im voraus einlesen (*Read-Ahead*).

Die wichtigsten Parameter für einen Buffer Cache sind seine Größe und die Strategie des Blockwechselns (meistens wird LRU verwendet).

Platten-Blöcke, deren Änderungen noch nicht physisch verwirklicht sind, heißen „*dirty blocks*“. Bei einem Stromausfall oder einem ungünstigen Absturz kann es passieren, dass Dateien inkonsistent werden, weil noch nicht alle Daten auf der Platte gelandet sind – besonders tragisch bei Verwaltungsdaten, die die Struktur der Dateien festlegen.

Das UNIX-Kommando `sync` schreibt alle Filesystem-Puffer auf Platte (*Synchronisieren*). Wenn das System heruntergefahren wird, sollte man ein `sync` ausführen und kurz warten, bis die Platten sich beruhigt haben. (`shutdown` und `reboot` erledigen das meist selbständig).

13.3 Verschiedene Dateisysteme

Es können mehrere Filesysteme gleichzeitig benutzt werden, allerdings normalerweise immer nur eines auf einer Festplatten-Partition. Die Systeme unterscheiden sich beispielsweise

- in den zulässigen Dateinamen
- in der Größe der Blöcke/Cluster
- in der internen Verwaltung: Zugehörigkeit Datei ↔ Datenblöcke, Directories, etc.
- in Vorhandensein/Nichtvorhandensein bestimmter Attribute (wie Zugriffsrechte)

Unter L*i*nux erhält man wie folgt Informationen über die installierten (verfügbaren) Filesysteme – sie müssen dazu auf keinem gemounteten Gerät existieren:

```
cat /proc/filesystems
```

- Die meisten Dateisysteme auf einem Rechner sind lokal, d.h. sie liegen auf Geräten, die direkt an den Rechner angeschlossen sind. Darüber hinaus kann aber über ein Netzwerk auf Dateisysteme zugegriffen werden, die auf anderen Rechnern liegen. Man benötigt dazu ein spezielles weiteres Dateisystem, das die Verbindung zwischen lokalem und entferntem Dateisystem übernimmt (z.B. NFS).

- Außerdem sind medienlose Filesysteme realisiert. Hier täuscht das Filesystem bei den üblichen Zugriffen das Vorhandensein von Dateien vor, erzeugt die Daten aber selbst. Wir hatten in UNIX schon `/proc` und `/dev/fd` kennengelernt.

- Es macht durchaus Sinn, mehrere lokale Dateisysteme zur Verfügung zu haben. Eventuell ist eines besser (da schneller) für große Dateien, eines für viele kleine Dateien geeignet; eines eignet sich für große Platten, eines besser für Diskettenlaufwerke. Außerdem können so UNIX-fremde Formate wie MS-DOS- und Windows-Systeme integriert werden.

  Nachdem man allerdings ein Filesystem auf seiner Platte eingerichtet hat, ist es praktisch unmöglich, es ohne Zuhilfenahme eines Backup-Mediums in ein anderes zu konvertieren.

Einige verbreitete Filesysteme sind (L*i*nux-Bezeichnungen):

`affs`	**AMIGA fast file system** Unterversionen 0 bis 6, die im Bootblock identifiziert werden, Dateinamen 32 Zeichen (zusätzlicher Info-String mit 100 Zeichen), gehashte Directories, Blockgrößen 512 bis 32768 Bytes

`bfs`	**Boot File System** Spezielles L*i*nux-Filesystem für Bootloader und Kernel. Die Trennung von anderen Dateisystemen ermöglicht das Unterbringen in EPROMS etc.
`ext`	**Extended File System** das erste L*i*nux-eigene Dateisystem (auch `extfs`), längere Dateinamen und größere I-Nodes als das vorher übliche MINIX-System
`ext2`	**Extended File System 2** `ext` in der überarbeiteten Version (auch `ext2fs`), das momentan gebräuchlichste L*i*nux-Filesystem
`fdfs`	**File Descriptor File System** virtuelles Dateisystem von UNIX: File-Deskriptoren offener Dateien werden auf Pseudo-Dateien abgebildet, meist gemountet als `/dev/fd` (ggf. ähnlich: `fifofs`, `specfs`)
`hpfs`	**High Performance File System** das OS/2-eigene Filesystem
`iso9660`	**ISO-9660-Norm** Norm der International Standards Organisation für CD-ROMs
`minix`	**MINIX File System** UNIX-typisches Dateisystem, Dateinamen mit 14 (oder 30) Zeichen
`msdos`	**MS-DOS File System** Dateinamen im 8.3-Format, FAT-orientiert
`ncpfs`	**Novell Corporation File System** NetWare, verteiltes Dateisystem, NCP-Protokoll, Analogon zum NFS
`nfs`	**Network File System** Sun, verteiltes Dateisystem für TCP/IP, NFS-Protokoll
`proc`	**Process Information File System** virtuelles Dateisystem ab SVR4: nicht auf einem Medium untergebracht, Prozess- und Systemdaten werden auf Pseudo-Dateien abgebildet, meist gemountet als `/dev/proc`
`rfs`	**Remote File Sharing** AT&T, verteiltes Dateisystem
`smbfs`	**Server Message Block File System** verteiltes Dateisystem bei Windows NT/for Workgroups, LAN Manager
`swap`	**Swap File System** L*i*nux-Dateisystem zum Auslagern von Speicherseiten (in eine eigene Partition statt in eine Auslagerungsdatei)
`sysv`	**System 5** Dateisystem von SVR3, Dateinamen 14 Zeichen, Blockgröße 512 bis 2048 Bytes, meist nur noch aus Kompatibilitätsgründen unterstützt, Synonyme: `s5`, `s51k`
`ufs`	„**UNIX File System**“ (BSD) das sogenannte „fast filesystem“ von BSD, Dateinamen 255 Zeichen, Blockgröße 4 oder 8 KByte, Blockfragment-Verwendung

`umsdos`	**UNIxFied MS-DOS File System** MS-DOS-Dateisystem plus „lange Dateinamen“, Zugriffrechte, Links etc. Die Zusatzinformationen werden in der DOS-*Datei* (!!) „`--linux-.---`“ (eine pro Directory) abgespeichert
`vfat`	**Windows File System** (95^+/NT 3.5) „lange Dateinamen“ (255 Zeichen), weiterhin DOS-kompatibel, das DOS-System hieß auch FAT (von file allocation table)
`vfat32`	**Windows 32-Bit File System** schneller als `vfat`, bessere Raumnutzung, nicht mehr DOS-kompatibel
`xenix`	**XENIX File System** entspricht `sysv`
`xiafs`	**Xia File System** File System von Q. Frank Xia, nur L*i*nux, inzwischen weniger gebräuchlich, historisch zwischen `ext` und `ext2`

Nicht alle aufgeführten Dateisysteme bieten alle Dateioperationen. Beispielsweise hat `msdos` keine Datei-Zugriffsrechte und -Besitzer. Beim Zugriff von UNIX aus werden dann Standardrechte benutzt, und `root` ist der Besitzer aller Dateien. `proc` kann nicht beschrieben werden, etc. Viele der angegebenen Dateisysteme sind auch für L*i*nux verfügbar (ggf. muss man einen neuen Kernel compilieren).

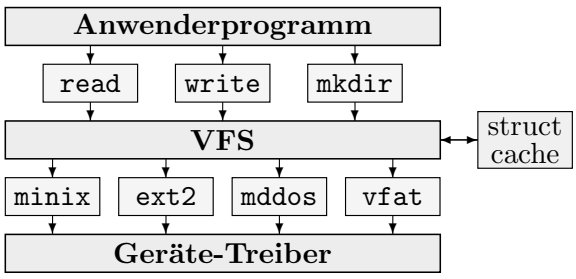
Praktische alle UNIX-eigenen Dateisysteme sind von der Struktur her Abwandlungen des klassischen `sysv`. Sie sind durch einen *Superblock* und eine Dateiverwaltung per *I-Nodes* gekennzeichnet. Die UNIX-fremden Dateisysteme wie `msdos` und `vfat` sind völlig anders aufgebaut.

13.4 Das Virtual File System (VFS)

Die unterschiedlichen Filesystem-Parameter sind Interna. Die Datei-Operationen von Programmen aus sollten immer exakt gleich aussehen, und ein Programm soll nicht bemerken, auf welches Filesystem es bei einer Operation tatsächlich zugreift.

Dazu gibt es in UNIX ab SVR4 (auch in L*i*nux) das „Meta-Filesystem“ VFS (*Virtual File System (Switch)*). Es ist ein Teil des Kernels, der eine Schnittstelle zu den anderen, konkreten Filesystemen darstellt.

Das VFS verhält sich nach außen absolut so, als spräche man direkt ein konkretes Filesystem an. Es hat sich aus urtümlichen UNIX-Filesystemen entwickelt und bildet daher die dort typischen Erscheinungen als Spiegelungen ab (es gibt dort VFS-I-Nodes und VFS-Superblocks, s.u.).



Der „Struktur-Cache“ wird direkt vom VFS verwaltet und nimmt zwischenzeitlich Directory- und I-Node-Informationen auf, damit sie bei erneutem Zugriff schnell verfügbar sind. Auf Ebene der einzelnen Filesysteme und der Gerätetreiber gibt es andere, reine Datenblock-Caches.

Beispiel: In L*i*nux werden Strukturen wie die folgenden verwendet, um Anfragen an das VFS an die speziellen Implementationen weiterzuleiten (`fs.h`):

```
struct file_operations
 {
   int (*lseek) (struct inode *, struct file *, off_t, int);
   int (*read) (struct inode *, struct file *, char *, int);
   int (*write) (struct inode *, struct file *, const char *, int);
   ...
 };

struct inode_operations
 {
   struct file_operations * default_file_ops;
   int (*create) (struct inode *,const char *,int,int,struct inode **);
   int (*lookup) (struct inode *,const char *,int,struct inode **);
   ...
 };

struct super_operations
 {
   void (*read_inode) (struct inode *);
   void (*statfs) (struct super_block *, struct statfs *, int);
   int (*remount_fs) (struct super_block *, int *, char *);
   ...
 };
```

Diese drei Strukturen dienen zum Ansprechen von Files, der Datei-Verwaltungsstrukturen (I-Nodes) und der Verwaltungsstrukturen des ganzen Filesystems (Superblöcke).

Jedes Programmsystem, das die obigen Funktionen als Schnittstellen zur Verfügung stellt, ist ein Filesystem, das VFS ansprechen kann. Es muss überhaupt kein zugrundeliegendes physisches Medium geben! Ein Beispiel dafür ist das Filesystem `proc`, das wir schon in 6.2.2 kennengelernt haben. Kernel-interne Informationen über Prozesse und Systemdaten werden über die Datei-Schnittstellen von oben nach außen zur Verfügung gestellt.

Beispiel: Die Quellen zu den Filesystemen sind bei L*i*nux in `src/fs` untergebracht. Für jedes gibt es dort je ein Unterdirectory, beispielsweise `src/fs/ext2`.

Die Initialisierung aller eincompilierten Filesysteme findet beim Bootvorgang statt, zu finden in `src/fs/filesystems.c`:

```
int sys_setup(void)
 {
   static int callable = 1;   // durch diesen Code kann die Routine
   if (!callable) return -1;  // nur einmal überhaupt aufgerufen werden
   callable = 0;

   device_setup();
   binfmt_setup();            // exec.c, Registrierung von bin, elf, aout, ...

#ifdef CONFIG_EXT2_FS
   init_ext2_fs();
```

```
#endif
#ifdef CONFIG_MINIX_FS
   init_minix_fs();
#endif
   ...                          // entsprechend für die anderen unterstützten FS

   mount_root();                // siehe super.c
   return 0;
 }
```

Die Filesysteme müssen sich zunächst vom VFS *registrieren* lassen. Das ist im Allgemeinen das einzige, was in ihrer `init_***_fs`-Routine geschieht.

Die Registrierungs-Struktur enthält Namen und einen Pointer auf die Routine, die ein solches Filesystem ins System einhängen kann (insbesondere liest sie den „Superblock“ mit den Systemparametern vom Medium):

```
struct file_system_type                 // Definition aus fs.h
 {   struct super_block * (*read_super)(struct super_block *, void *, int);
   const char *name;
   int requires_dev;
   struct file_system_type *next;
};
```

Für `ext2` sieht die Registrierung wie folgt aus (`fs/ext2/super.c`):

```
static struct file_system_type ext2_fs_type =
 {
   ext2_read_super, "ext2", 1, NULL
 };

int init_ext2_fs(void)
 {
   return register_filesystem(&ext2_fs_type);
 }

struct super_block *ext2_read_super
 (struct super_block *sb, void *data, int silent)
 { ... }
```

Die für die Registrierung verantwortliche Routine ist `register_filesystem` in `fs/super.c`:

```
static struct file_system_type *file_systems=(struct file_system_type *)NULL;

int register_filesystem(struct file_system_type *fs)
 {
   struct file_system_type ** tmp;
   if (!fs) return -EINVAL;
   if (fs->next) return -EBUSY;
   tmp = &file_systems;
   while (*tmp)
    {
```

```
        if (strcmp((*tmp)->name, fs->name) == 0) return -EBUSY;
        tmp = &(*tmp)->next;
      }
    *tmp = fs;
    return 0;
  }
```

13.5 Anlegen und Mounten von Dateisystemen in UNIX

Das Kommando `mkfs` (*make file system*) dient zum Erzeugen einer Dateisystem-Struktur auf einem Gerät:

`mkfs -t` *type device*

type ist der Name eines vom System unterstützten Filesystems (z.B. `ext2`), und *device* ist die Gerätedatei (z.B. `/dev/hda1`). `mkfs` ist eine Schnittstelle zu den eigentlichen, Filesystem-spezifischen Befehlen (z.B. `mkfs.ext2`). Optionen im `mkfs` werden an sie weitergereicht und passend interpretiert.

Es gibt ja nur einen Dateibaum in UNIX, in dem also im allgemeinen unterschiedliche Geräte und Dateisysteme zusammengehängt sind (auch beispielsweise Wechselmedien wie Disketten und CD-ROMs).

Das Zusammenhängen geschieht mit dem Kommando `mount`. Ein Teil-Dateisystem wird dabei an eine anzugebende Stelle in den Dateibaum eingefügt und erscheint danach ab dieser Stelle als Teilbaum. Mit `umount` kann der Teilbaum wieder entfernt werden. Beides ist normalerweise nur dem Superuser erlaubt.

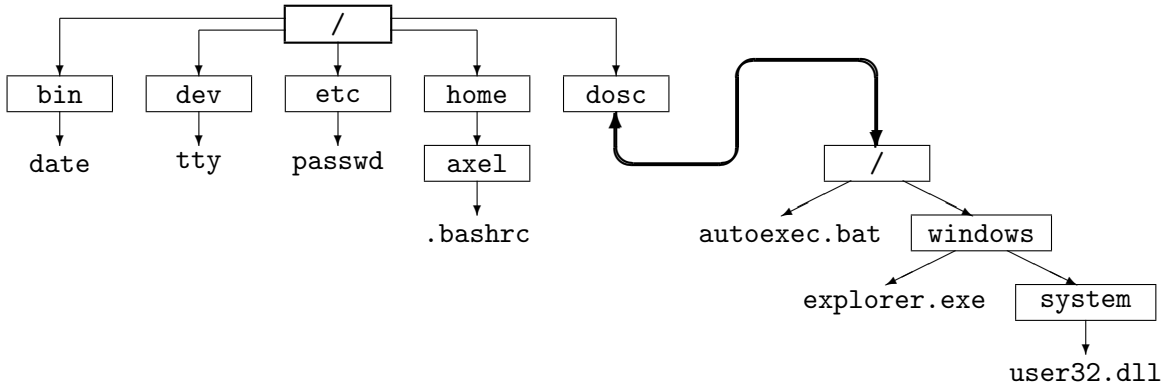
Der Typ des Dateisystems wird dabei mit `-t` angegeben:

`mount -t` *type device dir*

Das Verzeichnis *dir* heißt *Mount Point* des Dateisystems. Es muss schon existieren (meist leer) und wird ab jetzt durch den neuen Dateibaum überdeckt. Beispielsweise hängt das Kommando

```
mount -t vfat /dev/hda1 /dosc
```

ein `vfat`-Dateisystem (Windows) auf der Plattenpartition, die über die Gerätedatei `/dev/hda1` angesprochen wird, an der Stelle `/dosc` ein. Danach könnte also beispielsweise ein Dateiname wie `/dosc/AUTOEXEC.BAT` verwendet werden:



- Wenn ein Dateisystem über das Netz gemountet wird, wird keine Gerätedatei angegeben, sondern eine Angabe gemacht, die spezifisch für das verwendete verteilte Dateisystem ist, z.B. „`IP_address:remote_directory`“.

- Mit der Option „`-r`“ kann ein Dateisystem als read-only gemountet werden und ist dann schreibgeschützt. Das ist je nach Medium ohnehin Standard (CD-ROM), macht aber auch bei Festplatten-Partitionen Sinn, beispielsweise wenn ein Fehler auf ihnen aufgetreten ist und vor einem Rettungsversuch keine weiteren Daten zerstört werden sollen.

- `mount` trägt die eingehängten Dateisysteme in die Datei `/etc/mtab` (*mount table*) ein. Außer hierüber erhält man mittels „`mount`“ (ohne Argumente) entsprechende Informationen. In einigen Systemen gibt es außerdem `/proc/mounts`, die meist genau mit `/etc/mtab` übereinstimmt.

- Mit `mount -t` *type* erhält man Informationen über alle Dateisysteme des Typs *type*, die im Moment gemountet sind. Beispielsweise könnte „`mount -t ext2`“ liefern:

  ```
  /dev/hda8 on / type ext2 (rw)
  /dev/hda9 on /usr type ext2 (rw)
  ```

- Der Aufruf „`mount -a`“ (*auto*) mountet alle Dateisysteme entsprechend den Angaben in der Datei `/etc/fstab` (*file system table*). Dort ist die übliche Konfiguration abgelegt, damit nicht bei jedem Systemstart viele einzelne `mount`s abgesetzt werden müssen. „`mount -a -t` *type*“ mountet nur alle Dateisysteme vom Typ *type*.

  Die Datei enthält je Mount Point eine Zeile und ist in mindestens drei Spalten organisiert: *device dir type*, z.B.:

  ```
  /dev/hda8     /       ext2      defaults        1  1
  /dev/hda1     /dosc   vfat      defaults        0  0
  /dev/hdc      /cdrom  iso9660   ro,noauto,user  0  0
  none          /proc   proc      defaults        0  0
  ...
  ```

  In der *vierten* Spalte kann eine Liste von Flags angegeben werden. `defaults` entspricht einer Kombination von Standard-Flags. Einige der Flags sind folgende:

  - Einträge mit `noauto` werden bei „`mount -a`“ übergangen. Der Eintrag macht dennoch Sinn, da so ein nachträglicher kurzer `mount`-Aufruf möglich ist: „`mount` *device*“ oder „`mount` *dir*“, z.B. oben „`mount /cdrom`“.
  - `rw` steht für read-write, `ro` für read-only, also schreibgeschützt (entspricht dem „`-r`“ bei `mount`).
  - Ein `user`-Flag erlaubt es ausnahmsweise auch normalen Benutzern, das Filesystem zu mounten, nicht nur dem Superuser. Bei Wechseldatenträgern wie Disketten oder CD-ROMs ist das sicherlich sinnvoll. Standard ist `nouser`.
  - `remount` versucht, ein schon gemountetes Filesystem erneut zu mounten (z.B. mit geänderten Flags wie `rw`↔`ro`).

Die *fünfte* Spalte steht für „Dump Frequency“ und gibt an, wie oft (in Tagen) das System per `dump` ein Backup fertigen sollte.

Die *sechste* Spalte heißt „Pass Number“ und gibt die Reihenfolge an, in der `fsck` (s.u.) die Konsistenz der Filesysteme überprüfen sollte (0=Plazierung egal, 1=Root-System, höhere Nummern=Plazierung).

Um eine *neue Festplatte* zu integrieren, sind also grob folgende Schritte zu erledigen:

- Festplatte anschließen, ggf. in BIOS oder Monitor anmelden
- einen passenden Gerätetreiber im Kernel finden oder ggf. einen neuen Kernel mit einem neuen Treiber compilieren
- ggf. die Platte (Low-Level-) formatieren
- die Platte in Partitionen (eventuell nur eine) unterteilen (`fdisk` o.ä.)
- mit `mknod` die Special Files für den Zugriff anlegen (wenn es nicht schon welche „auf Vorrat“ gibt)
- mit `mkfs` ein Filesystem auf jeder Partition anlegen
- mit `fsck` das Filesystem überprüfen
- die Filesysteme in `/etc/fstab` eintragen (eventuell hat sich die neue Platte in der Nummerierung „zwischen“ alte geschoben; das muss hier berücksichtigt werden!)
- die Filesysteme mit `mount` einhängen (vorher ggf. mit `mkdir` Mount Points anlegen)
- eventuell systemspezifische Dinge (Quotas) einrichten

Spezifische Systeme haben ggf. eigene Programme, die einiges an Arbeit automatisieren.

L*i*nux sollte eine frisch angeschlossene Platte beim Booten erkennen und das melden. Mit `dmesg` kann man sich die eventuell vorbeigescrollten Boot-Meldungen noch einmal dahingehend anschauen.

13.6 Aufbau von UNIX-Dateisystemen

Ein System-V-ähnliches Dateisystem hat folgenden Aufbau:

BBl	**SBl**	**IBm**	**DBm**	**I-Nodes ...**	**D-Zone ...**

(BBl=Bootblock, SBl=Superblock, IBm=I-Node Bitmap, DBm=D-Zone Bitmap)

Der Bereich „D-Zone“ enthält reine Datenblöcke mit Teilen des tatsächlichen Inhalts einer Datei oder eines Directories. Alle anderen Blöcke werden zur Verwaltung der Platten- und Dateistruktur benötigt.

Alle Blöcke werden von 0 an aufsteigend durchnummeriert, und diese Nummern werden in den Verwaltungsstrukturen als Verweise („Pointer“) verwendet (ab 1, eine Null steht für einen leeren Verweis). Die maximale Größe einer Partition hängt daher auch davon ab, mit wieviel Bytes die Nummern abgelegt werden. (Bei 2 Bytes und 1-KByte-Blöcken kommt man z.B. nur auf 64-MB-Partitionen!)

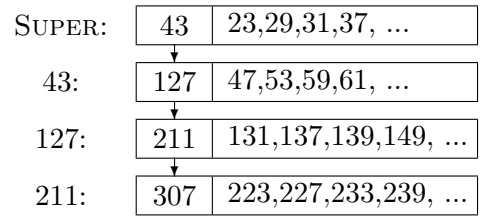
- Der **Bootblock** kann ein sehr kurzes Programm zum Booten des Betriebssystems enthalten (Boot Loader). (Er macht nur im Root-Dateisystem Sinn.)

- Der **Superblock** enthält grundlegende Informationen über die Parameter des Dateisystems, z.B.:
  - Name des Filesystems
  - Status (z.B. read-write oder read-only)
  - Anzahl der I-Nodes
  - Anzahl der Datenblöcke
  - Größe der I-Node-Bitmap
  - Größe der D-Zone-Bitmap
  - Blöcke pro Cluster (meistens als log_2(Blöcke pro Cluster), d.h. die Anzahl Bits, um die man schieben muss, um von einer Block- auf eine Cluster-Adresse zu kommen und umgekehrt)

  Viele Systeme legen an bestimmten Stellen Kopien des Superblocks an, damit bei einem Hardwarefehler gerade im Bereich des Superblocks die Platte nicht völlig unlesbar wird.

- Die I-Node-Bitmap ist eine Tabelle mit einem Bit pro möglicher I-Node, das genau dann gesetzt ist, wenn die I-Node momentan belegt ist. Entsprechendes gilt für die D-Zone-Bitmap.

Bei einer 4 GByte Platte und einer Blockgröße von 1 KByte braucht man allerdings schon 512 KByte für die Bitmaps! Deshalb arbeiten neuere Systeme (auch `ext` und `ext2` bei L*i*nux) nicht mit Bitmaps, sondern mit einer verketteten Liste von Tabellen, die im Superblock begonnen wird, siehe das Schema rechts.



Wenn ein freier Block benötigt wird, wird zunächst in der Tabelle im Superblock selbst nachgeschaut und der erste eingetragene verwendet. Sollte die Tabelle dort inzwischen leer sein, wird die Tabelle aus der ersten Verkettungsstufe (hier Block 43) in den Superblock kopiert und erneut gesucht, etc. Freigewordene Blöcke werden in die letzte Tabelle der Verkettung eingetragen – wenn diese schon voll ist, muss eine neue Tabelle angelegt und angekettet werden.

Durch Bereichs-Angaben („Blöcke n_1 *bis* n_2 sind frei“) kann der Speicherplatz für die Tabellen klein gehalten werden.

Eine ganz andere Möglichkeit, die ganz ohne zusätzlichen Speicher auskommt, ist es, die freien Datenblöcke selbst in einer Liste zu verketten. Bei einem Hardwarefehler müssen dann aber alle freien Blöcke neu zusammengesucht werden.

13.6.1 I-Nodes und Datenblöcke

Die Herkunft des Namens „I-Node“ ist ein wenig unklar – Vorschläge sind *Information Node*, *Index Node*, *Indirection Node*.

Es handelt sich jedenfalls um eine (Filesystem-abhängige) Datenstruktur (meist 64 oder 128 Bytes groß) mit allen relevanten Informationen über eine Datei (außer ihrem Namen):

- Dateiart
- Größe (in Bytes und in Blöcken)
- Zugriffsrechte

- Link-Zähler (Anzahl der Hardlinks auf die Datei)
- Liste der Nummern der Datenblöcke der Datei

Wie Datenblöcke auch, werden I-Nodes über ihre Nummer angesprochen („*I-Number*“). Sie ist vom Typ `ino_t` (z.B. **`unsigned long`**). Eine Datei ist im System eindeutig durch das Paar (*Geräte-ID*,*I-Number*) gekennzeichnet.

- „`ls -i`“ listet die I-Numbers mit auf, z.B.:

```
 6028 bin          1 dosc       12055 lib        16068 sbin
20121 boot     20081 etc           1 proc       14057 tmp
10062 dev       4023 home       2012 root           2 usr
```

- Beim *Formatieren* (`mkfs`) wird festgelegt, welcher Teil einer Partition für Daten und welcher für I-Nodes zuständig ist. Damit wird die Anzahl der I-Nodes – also die maximale Anzahl von Dateien – festgelegt! Einige neuere Systeme erlauben das nachträgliche „Umschichten“ vom I-Node- in den Datenbereich oder umgekehrt.

- Durch die I-Node-Struktur sind eventuell viele Lesekopf-Bewegungen notwendig: Directory-Eintrag, I-Node und Datenblöcke einer Datei liegen in eventuell weit auseinanderliegenden Bereichen der Platte. Es hat sich aber herausgestellt, dass in einem normalen UNIX-System die I-Node einer Datei ca. 4 mal so oft gelesen wird wie die Datenblöcke!

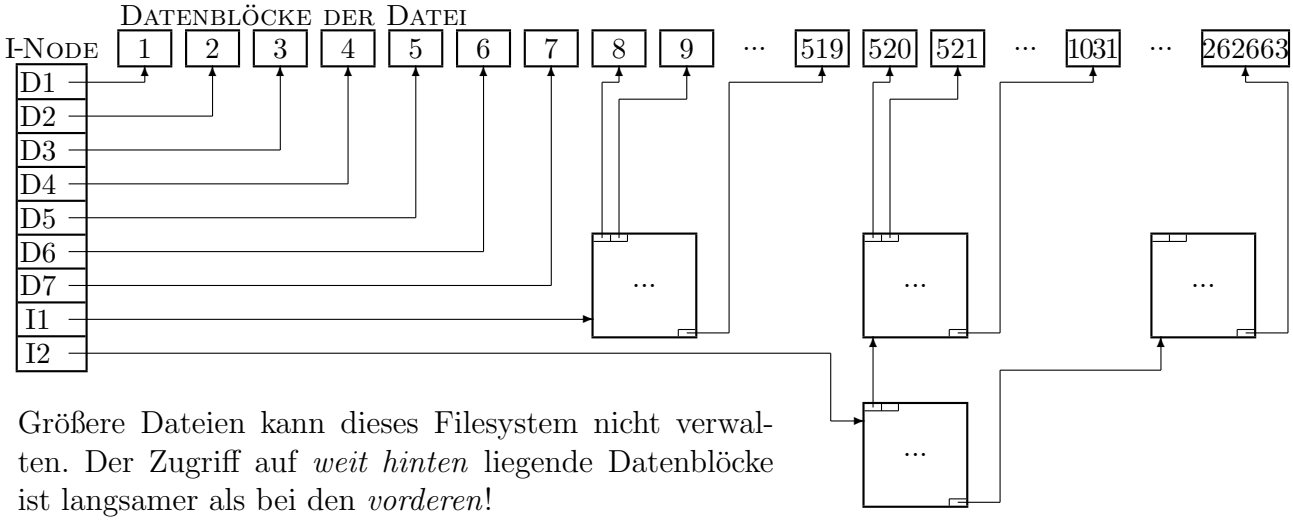
Die I-Nodes haben eine feste Länge und können daher meist nicht die Nummern *aller* Datenblöcke einer Datei aufnehmen. Dann wird mit Ergänzungsblöcken gearbeitet:

Im simpelsten UNIX-Dateisystem, dem von MINIX, enthält eine I-Node

– 7 Nummern, die **direkt** auf die ersten 7 Datenblöcke der Datei verweisen (reicht bei 1K-Blöcken bis 7 KByte)

– 1 Nummer eines Datenblocks mit **einfacher Indirektion**. Er enthält die Nummern der nächsten 512 Datenblöcke der Datei (reicht bis 519 KByte).

– 1 Nummer eines Datenblocks mit **doppelter Indirektion**. Er enthält die Nummern der nächsten 512 Blöcke mit *einfacher* Indirektion. So können weitere 2^{18} Datenblöcke angesprochen werden (reicht bis $512^2 + 512 + 7$ KByte ≈ 256.5 MByte).



Größere Dateien kann dieses Filesystem nicht verwalten. Der Zugriff auf *weit hinten* liegende Datenblöcke ist langsamer als bei den *vorderen*!

Neuere UNIX-Dateisysteme arbeiten mit einer *dritten* Indirektions-Stufe, die also Verweise auf doppelte Indirektions-Blöcke enthält. Die damit erreichbaren Dateigrößen dürften in der Praxis wohl kaum annähernd erreicht werden.

- Die Anzahl von Datenblöcken einer Datei liefert `ls` mit der Option „`-s`“ (*size*). Meistens werden echte Blöcke gezählt (z.B. à 1KByte), manchmal aber auch POSIX-Pseudoblöcke à 512 Bytes.

- In den meisten Dateisystemen sind die Datenblöcke wirklich *reine Datenblöcke*. Sie sind nicht miteinander verkettet und tragen auch keinen Verweis auf die I-Node, zu der sie gehören. Wenn die I-Node einer Datei durch einen Fehler verlorengeht, hat man Probleme!

- Die Dateien enthalten *kein spezielles Endekennzeichen* wie unter anderen Systemen. Die Größe ist allein durch den Größeneintrag in der I-Node festgelegt.

- *Verzeichnisse* werden als normale Dateien abgespeichert (genauer s.u.). Verzeichnisse wie `/` und `/usr`, auf die besonders häufig zugegriffen wird, werden meist so organisiert, dass die Directories möglichst klein bleiben und möglichst allein durch die I-Node angesprochen werden können. Dafür nimmt man für seltener angesprochene Unterdirectories symbolische Links in Kauf.

- Wenn eine neue Datei unter UNIX angelegt wird, geschieht also folgendes:
  - in der I-Node-Bitmap wird nach einer freien I-Node gesucht
  - in die I-Node werden Benutzernummer, Gruppennummer, Zugriffsrechte und Zeitwerte eingetragen
  - als Größe wird 0 eingetragen, als ersten Datenblock der Nullzeiger
  - die I-Number und der Dateiname werden im Verzeichnis eingetragen (ggf. muss dazu das Directory als Datei vergrößert werden, etc.)

  Wenn die Datei dann wächst (überhaupt Inhalt erhält), geschieht folgendes:

  - in der D-Zone-Bitmap wird ein freier Datenblock gesucht
  - die Nummer des ersten Blocks wird in die I-Node eingetragen
  - die ersten Daten werden in den Block eingetragen

  Wenn der erste Block voll ist, müssen weitere angelegt werden, und später ggf. zusätzliche Indirektions-Blöcke.

- Dateien können *Löcher* enthalten („*sparse files*“). Wenn man beispielsweise mit einem `lseek` über das Dateiende hinausfährt und dann schreibt, werden *nicht* entsprechend viele leere (mit Nullen gefüllte) Datenblöcke erzeugt. Vielmehr werden in der I-Node bzw. den Zusatzblöcken als Blocknummern für den leeren Bereich Nullen eingetragen. Auch zwischenliegende Indirektionsblöcke werden nicht belegt.

  Wenn aus dem leeren Bereich gelesen wird, erhält man Nullen. Wenn man hineinschreibt, müssen natürlich doch so viele echte Datenblöcke angelegt werden, wie nötig sind.

  Beispiel: Wir erzeugen eine ca. 1 MByte große Datei, die nur (z.B.) 4 Datenblöcke belegt:

```
int main()
 {
   int fd=open("1M.tst",O_WRONLY|O_CREAT);
   if (fd>0)
    {
      write(fd,"HELLO\n",6);
      lseek(fd,1000000L,SEEK_SET);
      write(fd,"HELLO AGAIN!\n",13);
      close(fd);
    }
 }
```

Z.B. im Linux-Filesystem `ext2` ergibt sich folgende Situation: Wir benötigen zwei Datenblöcke – einer für den ersten Text, einer für den zweiten. Die I-Number des ersten wird direkt in die I-Node eingetragen. Der erste Indirektionsblock wird nicht gebraucht (er reicht nur bis 268 KByte). Also wird zusätzlich nur noch ein doppelter Indirektionsblock und von dort aus ein einfacher Indirektionsblock belegt.

„`ls -l`“ liefert 1000013 Bytes als Dateigröße – dagegen liefert „`ls -s`“ folgendes:

```
4 1M.tst          5 a.out           1 make1M.c
```

Beispiel: Unter Linux sind VFS-I-Nodes in `fs.h` definiert (hier gekürzt):

```
struct inode
 {
   umode_t  i_mode;
   uid_t    i_uid;
   gid_t    i_gid;
   time_t   i_atime;
   ...
 };
```

Kommandos, die Manipulationen an einer physischen I-Node vornehmen wollen (beispielsweise `chmod`), müssen über den Pfad/Namen zu einer VFS-I-Node kommen. Dazu dient die VFS-Routine `namei` (in `fs/namei.c`):

```
int namei(const char *pathname, struct inode **res_inode);
```

Wenn die I-Node verändert werden soll, muss sie für den Zugriff von anderen Prozessen aus gesperrt werden. Das geschieht mit `iget` (aus `fs/inode.c`):

```
struct inode *__iget(struct super_block * sb, int nr, int crossmntp);
```

Nach der Manipulation muss die I-Node herausgeschrieben und freigegeben werden:

```
void iput(struct inode * inode);
```

Die eigentlichen Operationen auf einer physischen I-Node werden vom VFS vorgenommen. Sie sind Filesystem-abhängig und müssen im entsprechenden FS-Code (über Pointer in der `inode_operations`-Struktur) zur Verfügung gestellt werden.

13.6.2 ext2

Das erste L*i*nux-Dateisystem war 1991 ein von Linus Torvalds geschriebenes MINIX-System. Das erste L*i*nux-eigene Dateisystem ext wurde 1992 von Remy Card (Universität Paris) begonnen. Das inzwischen gebräuchliche ext2 (1993) behebt einige Mängel von ext. I-Nodes werden über eine verkettete Liste verwaltet und bieten Platz für zusätzliche Attribute, die vom Betriebssystem noch gar nicht unterstützt werden (zum Restaurieren versehentlich gelöschter Dateien, erweiterte Zugriffskontrolle, Gruppenunterteilung, Stabilität in vernetzten Systemen).

- Blöcke dürfen zwischen 1 und 4 KByte groß sein.
- Blocknummern werden mit 4 Bytes abgespeichert ⇒ theoretisch Partitionsgrößen bis 16 TByte möglich (momentan auf 2 GByte begrenzt).
- In die indirekten Blöcke passen zwar nur noch 256 Verweise, es gibt aber eine *dritte* Indirektions-Stufe eingeführt. Die maximalen Dateigrößen bei 0,1,2 und 3 Indirektionen sind 12 KByte, 268 KByte, ≈ 64.02 MByte und $256^3 + 256^2 + 256 + 12$ KByte ≈ 16.06 GByte.
- Symbolische Links können allein durch ihre I-Node dargestellt werden („*fast symbolic links*“) – statt der Blocknummern wird der Pfad abgespeichert, sofern er kurz genug ist.
- Angebrochene Blöcke werden weiterverwendet („Fragmente“).
- Das gesamte System ist in Gruppen zu je 8192 Blöcken aufgeteilt, die fast wie Partitionen einer Festplatte anzusehen sind. Jede Gruppe hat einen Gruppenblock, der in seiner Funktion etwa dem Superblock entspricht. Kleine Defekte haben so nicht gleich Auswirkungen auf das gesamte System. Durch die Gruppierung liegen Datenblöcke näher an den zugehörigen I-Nodes (schnellerer Zugriff).

Der genaue Aufbau von I-Nodes, Superblock etc. ist in src/.../ext2_fs.h definiert (**struct** ext2_inode, **struct** ext2_super_block, etc.), abgekürzt:

```
#define EXT2_NDIR_BLOCKS                12
#define EXT2_IND_BLOCK                  EXT2_NDIR_BLOCKS
#define EXT2_DIND_BLOCK                 (EXT2_IND_BLOCK + 1)
#define EXT2_TIND_BLOCK                 (EXT2_DIND_BLOCK + 1)
#define EXT2_N_BLOCKS                   (EXT2_TIND_BLOCK + 1)

struct ext2_inode
 {
   unsigned short i_mode;                              // File mode (→ stat)
   unsigned short i_uid;                               // Owner Uid
   unsigned long  i_size;                              // Size in bytes
   unsigned long  i_atime, i_ctime, i_mtime, i_dtime; // Timestamps
   unsigned short i_gid;                               // Group Id
   unsigned short i_links_count;                       // Links count
   unsigned long  i_blocks;                            // Blocks count
   unsigned long  i_flags;                             // File flags
   ...
```

```
    unsigned long  i_block[EXT2_N_BLOCKS];          // Pointers to blocks
    ...
};
```

Für `ext2` gibt es außerdem u.a. einen Debugger `debugfs`, ein Tuning-Tool `tune2fs` und Informations-Ausgabe per `dumpe2fs`. Beispielsweise könnte „`dumpe2fs /dev/hda8`“ liefern:

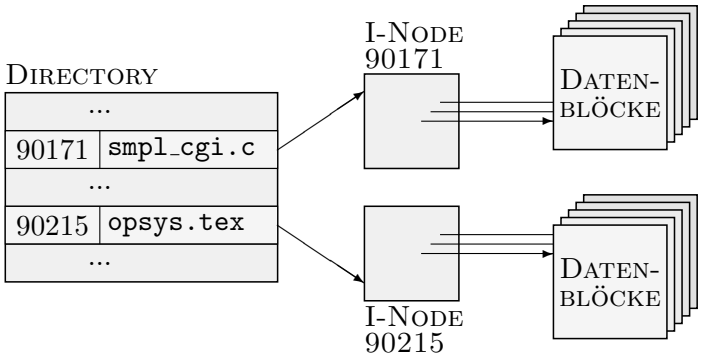
```
Filesystem state:         not clean
Filesystem OS type:       Linux
Inode count:              26104
Block count:              104391
Free blocks:              23287
Free inodes:              18873
First block:              1
Block size:               1024
Last mount time:          Sat Dec  5 16:05:32 1998
Last write time:          Sat Dec  5 22:19:27 1998
Mount count:              8
Maximum mount count:      20
Last checked:             Sun Nov 29 02:42:47 1998
...
```

13.6.3 Directories

In UNIX ist ein Directory immer einfach eine Tabelle von Paaren (*I-Number*,*Name*) in Form einer Binärdatei. Der Rest der Datei-Informationen steckt in den I-Nodes.

Wenn wir einen Hardlink auf eine der Dateien setzen, enthalten einfach zwei Directory-Einträge (ggf. im selben Directory) dieselbe I-Number.

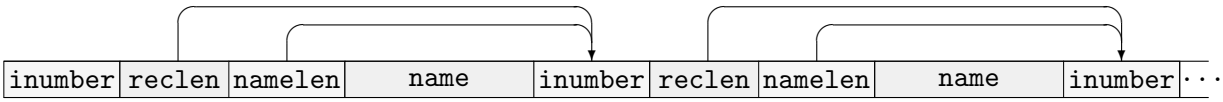
Die Einträge „`.`“ und „`..`“ sind echte Directory-Einträge (Hardlinks auf die Directory-Datei selbst bzw. auf die vom Parent-Directory).



- Directories schrumpfen in UNIX physisch nie. Wenn eine Datei gelöscht wird, wird der Eintrag als unbelegt gekennzeichnet (z.B. durch Setzen der I-Number auf 0). Er kann später für eine neue Datei wiederverwendet werden. Das Directory wird aber nie verkürzt und Zugriffe daher ggf. langsam (dann ist ein Verschieben in ein neues Directory ratsam).

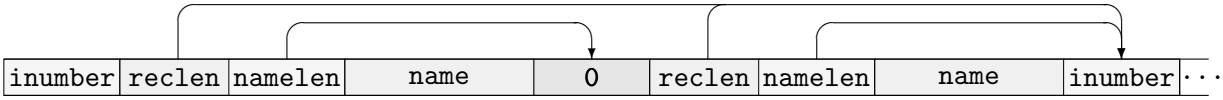
- In MINIX konnten Dateinamen maximal 14 Zeichen lang sein – was den Grund hatte, dass dann ein Directory-Eintrag genau 16 Bytes groß war und man mit dieser Zahl gut rechnen konnte.

  Wenn Namen variabel lang sein können, arbeitet man meist mit einer Art Verpointerung innerhalb des Directories: Jeder Eintrag enthält ganz vorn entweder seine Länge oder einen Verweis auf den nächsten (gültigen) Eintrag:



DIRECTORY

Wenn die zweite Datei gelöscht wird, wird der zweite Eintrag ungültig gemacht. Das erste `reclen` überspringt den Eintrag, sodass es nicht mehr auf dieselbe Stelle wie `namelen` verweist:



DIRECTORY

Das zweite `namelen` bleibt notwendig, um den zweiten Eintrag wieder neu belegen zu können.

Wenn der *vorderste* Eintrag ungültig ist, wird seine I-Number auf 0 gesetzt, und er muss beim Lesen des Verzeichnisses explizit übersprungen werden.

Andere Filesysteme, wie das von MS-DOS, führen ausführliche Directories, die alle Informationen enthalten, die unter UNIX in den I-Nodes steht.

Das AMIGA-Filesystem arbeitet mit Hash-Tabellen, um Dateien möglichst schnell zu finden. Aus dem Dateinamen wird eine kleine ganze Zahl berechnet, die als Index in eine Directory-intern Tabelle mit Block-Nummern verwendet wird. Der bezeichnete Block ist der File-Info-Block der Datei (analog I-Node). Bei Kollisionen (gleicher Index bei verschiedenen Namen) werden mehrere File-Info-Blöcke in einer linearen Liste verkettet.

13.6.3.1 Lesen von Directories

Auf alle Dateisysteme sind aber die in 3.2.2.1 beschriebenen Funktionen zum Einlesen von Directories (`opendir`, etc.) anwendbar. Es ist eine Aufgabe des jeweiligen Dateisystems, die ggf. ganz unterschiedliche Information in ein einheitliches Format umzuwandeln.

Die entsprechende Struktur **`struct dirent`** sieht unter L*i*nux z.B. wie folgt aus:

```
struct dirent
 {
   long           d_ino;
   off_t          d_off;
   unsigned short d_reclen;
   char           d_name[256];
 };
```

Der einzige von POSIX vorgeschrieben Eintrag ist `d_name` (die Länge ist nicht definiert). Fast alle UNIXe definieren aber auch den Eintrag `d_ino` mit der I-Number. `d_off` und `d_reclen` übernehmen die Rollen der „Verpointerungen“ von Seite 370.

BSD (in L*i*nux übernommen) definiert zwei nützliche Bibliotheks-Funktionen zum Einlesen ganzer Directories (in `dirent.h`):

UNIX

```
int scandir(const char *dir, struct dirent ***namelist,
    int (*select)(const struct dirent *),
    int (*compar)(const struct dirent *const *,
                  const struct dirent *const *));
```

Liest das Directory mit dem Pfad `dir` und legt selbständig Speicher für ein sortiertes Array `namelist` mit `dirent`s für die Einträge an. Für jeden Eintrag wird `select` aufgerufen, und nur Einträge, für die $\neq 0$ zurückgegeben wird, werden aufgenommen. Die Einträge werden mittels `compar` sortiert. (Die beiden Funktionspointer dürfen 0 sein, es wird dann nicht selektiert bzw. sortiert.) Liefert die Anzahl der Einträge zurück, -1 bei einem Fehler.

```
int alphasort(const struct dirent **a, const struct dirent **b);
```

Vergleicht zwei `dirent`s alphabetisch, kann beispielsweise für `compar` in `scandir` verwendet werden. Es wird zwischen Groß- und Kleinschreibung unterschieden. Der Rückgabewert ist wie bei `strcmp` zu interpretieren.

Die Vergleichsfunktion `compar` arbeitet mit `struct dirent **`, damit dort ggf. ganze `dirent`-Einträge physisch vertauscht werden können.

Beispiel: So kann man `scandir` dazu verwenden, alle `C`-Quelltexte im aktuellen Verzeichnis einzulesen und zu sortieren. Bei der Sortierung wird zunächst nicht zwischen Groß- und Kleinschreibung unterschieden, erst bei sonst vollständiger Übereinstimmung.

```
int dir_cmp(const struct dirent *const* e1, const struct dirent *const* e2)
 {
   int n=strcasecmp((*e1)->d_name,(*e2)->d_name);
   if (n==0) n=strcmp((*e1)->d_name,(*e2)->d_name);
   return n;
 }

int select_c(const struct dirent *e)
 {
   int l=strlen(e->d_name);
   return (l>=2) && (strcmp(e->d_name+l-2,".c")==0);
 }

int main()
 {
   struct dirent **dlist;
   int i,n;
   n=scandir(".",&dlist,select_c,dir_cmp);
   if (n<0) perror("scandir");
   else for (i=0;i<n;++i) { puts(dlist[i]->d_name); free(dlist[i]; }
   free(dlist);
 }
```

13.6.4 `fsck`

Das UNIX-Kommando `fsck` (und Verwandte) dient zum Überprüfen und ggf. Reparieren eines Filesystems (*file system check*):

`fsck` *filesystem*

Als *filesystem* kann die Gerätedatei oder der Mount Point angegeben werden (ein Check, während das System gemountet ist, ist nur für das Root-Filesystem erlaubt). Eventuell gibt es spezielle Kommandos (wie `fsck.ext2`, `e2fsck`), für die `fsck` nur eine Schnittstelle ist.

`fsck` testet z.B.:

- kommt ein Datenblock in mehreren Dateien vor?
- ist ein Block als belegt markiert, wird aber nicht benutzt?
- ist ein Block als unbelegt markiert, wird aber benutzt?
- wird ein Block als Datenblock angesprochen, obwohl er im I-Node-Bereich liegt?
- passen Dateilänge in der I-Node und Anzahl der benutzten Blöcke zusammen?
- passen die Directory-Links über `..` zusammen?

Schwere Dateisystem-Fehler kommen unter UNIX sehr selten vor. Halbwegs harmlose Fehler können (nach Nachfrage) von `fsck` repariert werden. Wenn allerdings Dateien ganz oder teilweise zerstört sein sollten, hilft nur das Einspielen eines Backups. Wenn das Root-Filesystem modifiziert worden ist, muss der Rechner neu gebootet werden.

Auf einigen Systemen gibt es `dfsck`, das zwei Geräte gleichzeitig (insgesamt schneller) überprüfen kann. Manchmal parallelisiert `fsck` auch selbständig (L*i*nux).

13.6.5 Informationen über Dateien, `stat`

In Abschnitt 3.2.5 hatten wir bereits die `stat`-Familie von UNIX-Systemaufrufen kennengelernt (`stat`, `lstat` für Link-Informationen, `fstat` für File-Deskriptoren). All ihre Informationen lesen sie aus der I-Node der Datei.

Der genaue Aufbau der Struktur **`struct`** `stat` ist nicht vorgeschrieben. Einige von POSIX festgelegte Felder sind folgende:

`mode_t st_mode;`	Modus der Datei, enthält in den untersten 12 Bit die erweiterten Zugriffsrechte, außerdem den Dateityp, siehe 3.2.5
`ino_t st_ino;`	Nummer der I-Node der Datei
`dev_t st_dev;`	ID des Geräts, auf dem die Datei liegt
`dev_t st_rdev;`	bei special files: ID des bezeichneten Geräts
`nlink_t st_nlink;`	Zähler für Hardlinks
`uid_t st_uid;`	UID des Besitzers
`gid_t st_gid;`	GID des Gruppe
`off_t st_size;`	Größe in Bytes (nicht für special files, bei symbolischen Links die Länge des Pfads)
`time_t st_atime;`	Access Time, Zeit des letzten Zugriffs (z.B. durch `read`), wird mit „`ls -lu`“ ausgegeben
`time_t st_mtime;`	Modification Time, Zeit der letzten Änderung (z.B. durch `write` oder Anlegen/Löschen von Dateien in einem Directory), wird mit „`ls -l`“ ausgegeben
`time_t st_ctime;`	Change Time, Zeit seit dem letzten Statuswechsel (z.B. durch `chmod`, `chown`, `unlink`), wird mit „`ls -lc`“ ausgegeben
`long` `st_blksize;`	Hinweis: bevorzugte Blockgröße
`long` `st_blocks;`	Anzahl belegter 512-Byte(!)-Blöcke (nicht für special files)

Zur *Änderung* der Zeitangaben gibt es übrigens den Systemaufruf `utime` (aus `utime.h`).

Viele der Dateisysteme aus der Tabelle von oben (auch reine UNIX-Systeme) sind nicht POSIX-kompatibel, da vorgeschriebene Informationen in den I-Nodes fehlen. Beispielsweise müssen die aufgeführten *drei* Datumseinträge pro Datei existieren, viele Systeme haben in den I-Nodes nur Platz für einen.

Das virtuelle System `VFS` bietet virtuelle I-Nodes an, in denen (im Hauptspeicher) alle drei Informationen untergebracht sind. Wenn eine In-Node aber auf Platte geschrieben werden muss (bei Pufferüberlauf oder Synchronisation), wird z.B. das *neueste* der drei Daten in die physische I-Node geschrieben.

13.6.6 Informationen über Dateisysteme, `statfs`

Es gibt keinen POSIX-Systemaufruf, der Informationen über ein gemountetes Filesystem einholt.

- Der System-V-Aufruf heißt `ustat` und liefert nur ganz rudimentäre Daten.

UNIX
`int ustat(dev_t dev, struct ustat *ubuf);` füllt die Struktur, auf die `buf` zeigt, mit Daten über das Filesystem mit der internen Gerätenummer (!) `dev`

  Die Struktur `ustat` (aus `sys/types.h`) liefert nur Daten über die Anzahl freier Blöcke und I-Nodes. Die benötigte Gerätenummer kann man über einen `stat`-Aufruf erhalten.

- Ein ausführlicherer und praktischerer Aufruf stammt aus BSD (auch in L*i*nux implementiert, deklariert in `sys/vfs.h`):

UNIX
`int statfs(const char *path, struct statfs *buf);` füllt die Struktur, auf die `buf` zeigt, mit Informationen über das Filesystem, zu dem die Datei mit dem Namen/Pfad `path` gehört.
`int fstatfs(int fd, struct statfs *buf);` wie `statfs`, aber es wird der File-Deskriptor einer offenen Datei auf dem Filesystem angegeben.

  Die verwendete Struktur ist wie folgt definiert:

```
struct statfs
 {
   long f_type;      // Filesystem-Typ
   long f_bsize;     // Optimale Blocktransfer-Größe
   long f_blocks;    // Anzahl Blöcke insgesamt
   long f_bfree;     // Anzahl freier Blöcke
   long f_bavail;    // Anzahl freier Blöcke für Normalbenutzer
   long f_files;     // Anzahl I-Nodes insgesamt ("file nodes")
   long f_ffree;     // Anzahl freier I-Nodes
   fsid_t f_fsid;    // Filesystem ID
   long f_namelen;   // Maximale Dateinamen-Länge
   long f_spare[6];  // in Reserve
 };
```

Der Aufruf spielt sich auf der Ebene des VFS ab. Daher liefern `files` und `ffree` die Anzahlen von möglichen „*Files*“ und nicht „*I-Nodes*“. Die unterliegenden konkreten Filesysteme brauchen ja gar nicht mit I-Nodes zu arbeiten.

Die Werte für `f_type` sind „Magic Numbers“, die in den Header-Dateien der entsprechenden Filesysteme definiert sind, beispielsweise `EXT2_SUPER_MAGIC`:

ext2	0xef53	msdos	0x4d44	proc	0x9fa0
minix	0x137f	nfs	0x6969	sysv	0x12ff7b5

Die „optimale Blocktransfer-Größe“ stimmt üblicherweise mit der Clustergröße (in Bytes) überein.

Ein Wert von -1 in einem der Felder bedeutet, dass das entsprechende Merkmal nicht vom angesprochenen Filesystem unterstützt wird.

Beispiel: Das folgende kurze Programm `fsinfo` gibt einige Informationen über das Filesystem aus, auf dem die ihm übergebene Datei liegt:

```
int main(int argc, char *argv[])
 {
   if (argc==2)
    {
      struct statfs buf;
      double size;
      int i;
      static unsigned char prefix[]=" KMGT";

      if (statfs(argv[1],&buf))
         fprintf(stderr,"huch -- kein statfs möglich für %s!\n",argv[1]);
      else
       {
         printf("Typ:                          0x%08lx\n",buf.f_type);
         printf("Blockgröße:                   %ld\n",buf.f_bsize);
         size=buf.f_blocks*buf.f_bsize;
         for (i=0;size>1024;size/=1024,++i);
         printf("Gesamt-Kapazität:             %.2f %cByte\n",size,prefix[i]);
         printf("maximale Dateinamen-Länge:  %ld\n",buf.f_namelen);
       }
    }
 }
```

Es könnten z.B. folgende beiden Ausgaben entstehen:

```
fsinfo /dosc
Typ:                          0x00004d44        // msdos
Blockgröße:                   16384             // 16K-Cluster!
Gesamt-Kapazität:             995.92 MByte
maximale Dateinamen-Länge:    12                // 8.3 DOS-Namen

fsinfo /usr
Typ:                          0x0000ef53        // ext2
```

```
Blockgröße:                 1024
Gesamt-Kapazität:           933.73 MByte
maximale Dateinamen-Länge:  255
```

13.7 MS-DOS-Filesystem

Die Zugehörigkeit von Datenblöcken zu Dateien (und ihre Reihenfolge) wird im MS-DOS-Filesystem mit einer Struktur namens **FAT** (*File Allocation Table*) abgebildet.

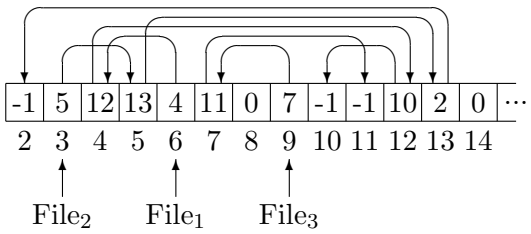
Dabei handelt es sich um eine Tabelle von 16-Bit-Werten (bei Festplatten, bei kleineren Medien ggf. nur 12 Bit), die jeweils einem Cluster zugeordnet sind.

- Die ersten beiden Einträge geben die Größe der Partition an.
- Ansonsten gibt der Wert die Nummer des nächsten Clusters in der Datei-Verkettung an (-1 als Ende-Kennzeichen).
- Der jeweils erste Block einer Datei wird im Directory-Eintrag abgelegt.
- Einträge unbelegter Cluster enthalten eine 0. Beim Löschen einer Datei müssen die zugehörigen FAT-Einträge mit Nullen überschrieben werden.

Beispiel: Wir betrachten die FAT einer Partition, die genau drei Dateien enthält, die folgendermaßen aus Blöcken aufgebaut sein sollen:

$File_1$: 6,4,12,10; $File_2$: 3,5,13,2; $File_3$: 9,7,11

Es ergibt sich das rechts angegebene Schema für den Beginn der FAT.



Bei Random-Access-Zugriff muss so natürlich diese Verpointerung nachverfolgt werden. Die FAT (oder der relevante Teil) passt aber in den Hauptspeicher, sodass man keinen Geschwindigkeitsnachteil hat.

Mit einer 16-Bit-FAT sind allerdings maximal 65534 Cluster ansprechbar, d.h. bei einer 4 GB-Platte müssen die Cluster 64 KByte groß sein! Die FAT selbst nimmt im Maximalfall 128 KByte ein. Mit 2K-Clustern kommt man nur bis 128 MByte.

Platten (bzw. Partitionen) sind wie folgt strukturiert:

Boot Sektor	enthält Daten über die Disk-Struktur (Anzahl Spuren/Sektoren, Anzahl der FATs) und ggf. ein sehr kurzes Programm, das beim Booten als erstes geladen und ausgeführt wird
FAT	die normale File Allocation Table
mehr FATs	ggf. Kopien der FAT aus Sicherheitsgründen bei Medienfehlern (alle FATs werden synchron aktualisiert)
Root Directory	liegt an einer festgelegten Stelle und kann maximal 112 Einträge aufnehmen
Daten	Platz für Dateien und andere Directories

Es gibt kein Analogon zu I-Nodes. Directories enthalten alle Information über die enthaltenen Dateien.

Wenn eine Datei gelöscht wird, wird der Directory-Eintrag dadurch ungültig gemacht, dass das erste Zeichen des Dateinamens mit einem Nullbyte überschrieben wird. Utilities wie `undelete` können versehentlich gelöschte Dateien wiederherstellen, sofern ihre Blöcke noch nicht anderweitig verwendet wurden. Der Benutzer muss dann jeweils das (alte/neue) erste Zeichen des Namens eingeben.

Der genaue Aufbau der Directory-Einträge wird beim VFAT-System von Windows 95 vorgestellt.

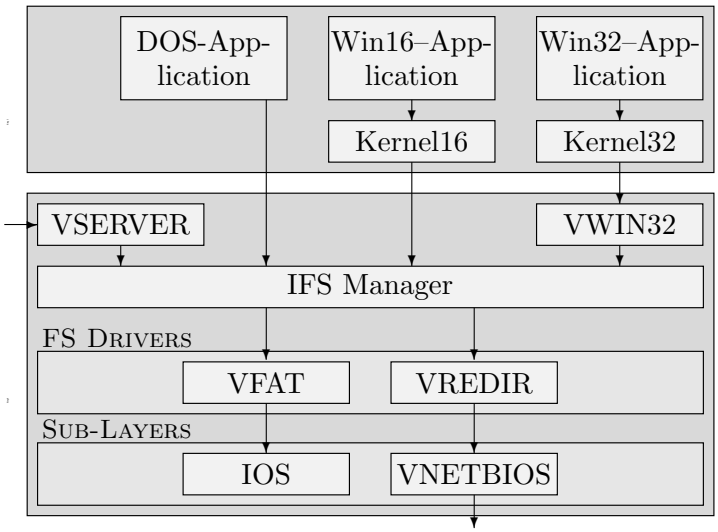
13.8 Filesysteme unter Windows 95

13.8.1 Der IFS Manager

Windows arbeitet im Gegensatz zu DOS mit beliebig vielen installierbaren Dateisystemen zusammen. Das Windows-Pendant zum VFS ist der IFS-Manager (IFSMgr, *Installable File System Manager*). Er wurde in einer eingeschränkten Version bereits in Windows for Workgroups 3.11 eingeführt, aber erst in Windows 95 zum vollen Umfang ausgebaut.

Er übernimmt die Umsetzung der Anfragen von Anwenderprogrammen (Clients) in spezielle Aufrufe der passenden konkreten Dateisystem-Treiber. Jeder Treiber stellt einen genau definierten Satz von Diensten zur Verfügung, die durch das Filesystem-API definiert sind.

Das folgende Bild stellt den IFSMgr und seine Beziehungen zu anderen Systemkomponenten dar (vgl. auch das folgende Kapitel):



Der IFS Manager hat zwei Schnittstellen:

Filesystem-API-Schnittstelle: Hier wird die Kommunikation zwischen IFSMgr und den einzelnen Filesystemen festgelegt. Eine Anfrage an einen FSD (File System Driver) erfolgt über eine Struktur `ifsreq` (IFS Request).

Rückmeldungen der Treiber gelangen über „Hooks“ zum IFSMgr, d.h. den Treiber-Aufrufen wird als Parameter die Adresse der Rückruf-Routine mitgegeben. Die Treiber füllen ebenfalls eine `ifsreq`-Struktur mit den angeforderten Daten.

Client-Schnittstelle: Von Anwendungen aus wird der IFSMgr durch Interrupts angesprochen. Der Windows-Kernel nimmt beim Start die entsprechende Initialisierung vor. Die Interrupt-21h-Funktionen 0 bis 71h haben aus Kompatibilitätsgründen dieselbe Funktionalität wie unter DOS, sind aber teilweise ein wenig erweitert – beispielsweise unterstützen sie lange Dateinamen. Es gibt zusätzliche Funktionen bis Nummer E7h.

Anfragen der drei Arten von Anwendungen nehmen unterschiedliche Wege zum IFSMgr:

- Eine MS-DOS-Anwendung (im virtuellen 8086-Modus) löst direkt einen Interrupt aus (die infragekommenden Nummern sind 17h, 21h, 25h, 26h, 2Fh). Er wird zu IFSMgr-Code umgelenkt. Der 8086-Interrupt muss in einen 32-Bit-Protected-Mode-Interrupt umgesetzt werden.
- Eine 16-Bit-Windows-Anwendung ruft eine dateibezogene Win16-API-Funktion auf, die vom 16-Bit-Kernel letztlich auf Interrupt-21h-Aufrufe zurückgeführt wird. Der 16-Bit-Protected-Mode-Interrupt muss in einen 32-Bit-Interrupt umgesetzt werden.
- Eine 32-Bit–Anwendung ruft eine Win32-API-Funktion auf. Ein VxD-Treiber ist zum Dispatching zwischengeschaltet und löst über den Ring-0-Service `Exec_PM_Int` einen Interrupt 21h aus, der beim IFSMgr landet.

Ein Client im Ring 0 ist z.B. `VSERVER`, die Server-Seite in einem Microsoft-Netzwerk. Netzanfragen landen zunächst hier und werden in IFSMgr-Aufrufe umgesetzt.

Es gibt unterschiedliche Arten von FSD:

lokale FSDs: Sie implementieren die Abbildung von System-Strukturen (wie Directories, Files und deren Attribute) auf Aufrufe eines lokalen Block-Gerätetreibers. Die eigentlichen I/O-Aufrufe setzen sie über den IOS (I/O Supervisor) von Windows ab.

Ein typisches Beispiel ist VFAT – die virtualisierte Version des FAT-Systems von MS-DOS.

Remote-FSDs: Sie dienen zur File-orientierten Kommunikation mit anderen Rechnern, d.h. entfernten Filesystemen. Die zu transportierenden Daten werden u.a. in Pakete zerlegt, die file-orientierten Aufrufe in Aufrufe der passenden Netzwerk-Protokolle umgesetzt (in Microsoft-Netzwerken z.B. SMB für die Verwaltung gemeinsam genutzter Dateien, NETBEUI für den eigentlichen Transport). Der Netz-I/O wird von einem anderen Windows-Subsystem übernommen, VNETBIOS.

13.8.2 VFAT

VFAT ist die virtualisierte Version des MS-DOS-Dateisystem-Treibers. Da gleichzeitig mit seiner Einführung auch die physische Darstellung leicht modifiziert wurde (lange Dateinamen), steht die Abkürzung VFAT auch für das Ablageformat auf dem Datenträger, und FAT für das alte Format.

Mit Windows-95-Updates wurde optional eine neue Version namens VFAT32 eingeführt, die mit 32-Bit-Strukturen arbeitet und daher unter anderem größere Partitionen unterstützt. Dafür ist sie nicht mehr von DOS aus lesbar. Die alte Version wird seitdem oft auch VFAT16 genannt.

Mit „langen Dateinamen“ sind nicht nur solche gemeint, die das alte Format von der Länge her sprengen. Es können beliebige Unicode-Buchstaben oder Ziffern und `$%'-_@~'!()^#&+,;=[]` dargestellt werden, insbesondere also Kleinbuchstaben und Umlaute. Dateinamen, die sich nur in (Unicode)-Groß- und Kleinschreibung unterscheiden, beziehen sich dabei auf dieselbe Datei.

Es gibt drei verschiedene Arten von VFAT-Directory-Einträgen:

Shortname Entry: ein normaler FAT-Eintrag, mit 8.3-Dateinamen

Longname Entry: Teil der Darstellung eines langen Dateinamens, bis zu 13 Unicode-Zeichen

Alias Entry: ein Shortname Entry, über den eine Datei mit Longname Entry alternativ (und von DOS aus) angesprochen werden kann

Die Eintrag-Typen sehen als C-Struktur wie folgt aus (jeweils 32 Bytes lang):

```
typedef struct                // Shortname Entry oder Alias Entry
 {
   char  deName[8];           // Namensteil vor dem Punkt
   char  deExtension[3];      // Namensteil nach dem Punkt
   BYTE  deAttributes;        // Attribute
   BYTE  deReserved[6];
   WORD  deLastAccessDate;    // erst ab Win95: letzter Zugriff
   WORD  deEAhandle;
   WORD  deCreateTime;        // Erzeugungszeit
   WORD  deCreateDate;        // Erzeugungsdatum
   WORD  deStartCluster;      // erster Datencluster der Datei
   DWORD deFileSize;          // Größe in Bytes
 }
DIRENTRY;

typedef struct                // Longname Entry
 {
   char    leSequence;        // Nummer in der Sequenz, ab 1, letzter +0x40
   wchar_t leName[5];         // 5 Zeichen des Namens (Unicode)
   BYTE    leAttributes;      // Attribute, immer 0x0F
   BYTE    leType;            // LongEntryType, immer 0
   BYTE    leChksum;          // Checksumme fur Alias-Korrespondenz
   wchar_t leName2[6];        // noch 6 Zeichen des Namens (Unicode)
   WORD    leZero;            // reserviert
   wchar_t leName3[2];        // noch 2 Zeichen des Namens (Unicode)
 }
LONGDIRENTRY;
```

- Bei der Einführung der langen Dateinamen war Microsoft bemüht, zu DOS und zu alten (fremden) Directory- und Filesystem-Utilities zu sein. Daher wird nun ein langer Dateiname durch entsprechend viele Einträge dargestellt, die fast so wie alte FAT-Einträge aussehen. In ihrem Attribut-Feld ist 0x0F eingetragen, was die meisten Programme akzeptieren. Da insbesondere das Hidden-Flag gesetzt ist, werden solche Einträge normalerweise nicht angezeigt. Aus denselben Kompatibilitätsgründen müssen die Zeichendaten im Eintrag ziemlich verstreut werden. Es sind maximal 255 Zeichen erlaubt (plus ein abschließendes Null-Zeichen).

- Ein Dateiname entspricht dem 8.3-Schema, wenn er maximal 8 Zeichen vor und 3 nach dem Punkt enthält, und zwar nur `$%'-_@~'!()^#&` oder Großbuchstaben. Wenn eine Datei angelegt wird, deren Name so aufgebaut ist, braucht nur ein Shortname-Entry angelegt zu werden.

  Ansonsten wird automatisch eine Sequenz von Longname-Entries und ein Alias-Entry angelegt. Der Alias-Name entsteht wie folgt: Es werden die ersten sechs Zeichen des Präfix des langen Namens übernommen. Großbuchstaben werden dabei in Kleinbuchstaben umgewandelt, Unicode-Buchstaben oder -Ziffern außerhalb des ISO-Latin1-Bereichs und die Sonderzeichen „`+,;=[]`“ werden dabei durch '_' ersetzt.

  Dann wird sukzessive „`~1`“, „`~2`“, ... und maximal die ersten drei Zeichen der Extension angehängt, bis es keinen gleichlautenden kurzen Dateinamen (Shortname oder Alias) gibt. Kommt man mit einer Ziffer nicht aus, wird das Präfix auf 5 Zeichen verkürzt, etc.

  Aus „`e=mc[2].formula`“ wird also „`E_MC_2~1.FOR`“ – vorausgesetzt, der Name existiert nicht schon. Danach wird aus „`e=mc[2+x].formula`“ ein „`E_MC_2~2.FOR`“, etc.

- Probleme gibt es immer, wenn von DOS oder DOS-Anwendungen aus Dateien gelöscht werden, die einen langen Dateinamen besitzen. Dann wird natürlich nur der Alias-Entry gelöscht, und die Longname-Entries sind keiner Datei mehr zugeordnet. Programme wie `scandisk` können solche übriggebliebenen Namen dann löschen.

13.8.3 API-Funktionen

In 16-Bit-Versionen gab es keine offiziell dokumentierten Windows-Funktionen für den Datei-Zugriff. Inoffiziell gab es die Funktionen `_lopen`, `_lread`, etc., die natürlich letztendlich DOS-Interrupts erzeugten, aber komfortabel zu bedienen waren.

Abgesehen von High-Level-Routinen, die mit File-Requestern zum Laden und Speichern arbeiten, gibt es ab Windows 95 folgende Datei-Aufrufe:

- `CreateFile` Datei anlegen, liefert File-Handle zurück
- `ReadFile` aus offener Datei lesen (per File-Handle)
- `WriteFile` in offene Datei schreiben (per File-Handle)
- `CloseHandle` offene Datei schließen (per File-Handle)

Es gibt außerdem wie unter UNIX die Möglichkeit, Dateien in den Hauptspeicher einzublenden (siehe `mmap`, Seite 67):

- `CreateFileMapping` Einblenden vorbereiten
- `OpenFileMapping` Zugriff auf schon eingeblendete Datei
- `MapViewOfFile` entspricht einem UNIX-„attach“
- `FlushViewOfFile` Synchronisation (Zurückschreiben von Änderungen)
- `UnmapViewOfFile` entspricht einem UNIX-„detach“

14 Windows, DOS und Prozesse

14.1 MS-DOS

Unter MS-DOS ist kein eigentliches Multitasking möglich („gleichzeitig“ *ablaufende* Programme). Es ist nicht re-entrant, d.h. es können nicht „gleichzeitig“ mehrere Instanzen einer DOS-Funktion durchlaufen werden. Man hat das auch im Lauf der Entwicklung nicht geändert, um keine Kompatibilitätsprobleme hervorzurufen. In späten MS-DOS-Versionen können aber mehrere Programme auf halbwegs sinnvolle Weise gleichzeitig *im Speicher existieren*, wie wir unten sehen werden.

14.1.1 `COM`s, `EXE`s und PSPs

Ein Prozess kann mit DOS-Aufrufen (oder auf einer höheren Ebene mit den `C`-Bibliotheksfunktionen der `exec`-Familie) einen neuen Prozess erzeugen und starten. Er ist dann aber bis zu dessen Beendigung zur Untätigkeit verdammt (wie z.B. `COMMAND.COM`). Es entsteht kein Prozess-*Baum*, sondern eine lineare *Liste*, bei der nur der Task am Ende aktiv ist.

Es gibt zwei ziemlich unterschiedliche Arten von DOS-Prozessen:

Prozesse aus `.COM`-Dateien: diese Art von Prozess wurde von CP/M geerbt. Die Datei darf maximal ein Speichersegment (64 KByte) groß sein und enthält ein *exaktes Speicherabbild* des Prozesses. Wenn es absolute Adressen enthält, muss das Programm an dieselbe Adresse geladen werden, von wo aus es einmal abgespeichert wurde.

Der gesamte Prozess liegt in *einem* 64-KByte-Segment: Code, Daten und Stack – alle Segmentregister werden auf denselben Wert gesetzt. Der Stack liegt dabei am oberen Ende und wächst rückwärts (Startwert `SP=0xfffe`).

Da die Datei keine Informationen über den zusätzlichen Speicherbedarf zur Verfügung stellt (CP/M-Prozesse waren immer alleine), teilt MS-DOS dem Prozess automatisch den gesamten Hauptspeicher zu. Wenn er einen anderen Prozess starten möchte, muss er dafür erst explizit Speicher freigeben.

Prozesse aus `.EXE`-Dateien: Bei diesem Format (erst ab MS-DOS 1.1) darf der Prozess *mehrere* Segmente verwalten: eines für den Stack, jeweils eines oder mehrere für Code, Daten und Extra-Segmente. Die Datei enthält Relokations-Angaben, so dass der Code an beliebige Hauptspeicheradressen geladen werden kann.

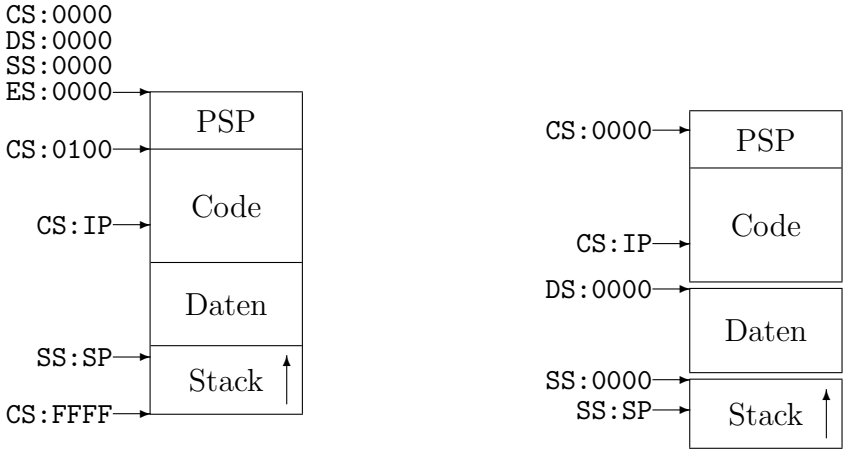
Jeder Prozess (aus `.COM`s oder `.EXE`s) beginnt im Speicher mit einer (von CP/M übernommenen) 256 Byte langen Struktur namens **PSP** (**Program Segment Prefix**).

Das PSP enthält u.a. folgendes:

- Adresse des Endes des belegten Speicherbereichs
- Adresse des Environment-Blocks (Tabelle von Strings „*var*=*value*“)
- Adresse des PSP des Elter-Prozesses
- Rücksprungadresse bei Beendigung des Prozesses
- Adresse des `CTRL-C`-Handlers (analog zu einer `SIGINT`-Handler-Routine)

- Anzahl Zeichen in der Aufrufzeile
- Inhalt der Aufrufzeile (mit `CR` abgeschlossen, max. 127 Zeichen)
- Kopien von Interrupt-Vektoren
- Funktionsbruchstücke zum erleichterten Interrupt-Aufruf

Unten ist die prinzipielle Speicherverteilung bei Prozessen dargestellt, die aus einer `.COM`- bzw. aus einer `.EXE`-Datei hervorgegangen sind:



`.COM`-Prozess

`.EXE`-Prozess

Wenn DOS einen neuen `.COM`-Prozess kreiert, geschieht folgendes:

- DOS belegt den kompletten Hauptspeicher (bis zur 640K-Grenze) für den Prozess.
- Der PSP wird aufgebaut. Es gehen Informationen aus dem File-Header, der Aufrufzeile und dem Elter-PSP (Environment, Rückverkettung) ein.
- Der komplette Datei-Inhalt wird unverändert in den reservierten Speicher geladen.
- Das Programm wird durch Sprung an die Adresse PSP+256 gestartet.

Beim Anlegen eines `.EXE`-Prozesses ist einiges anders:

- DOS liest aus dem File-Header die Größe des benötigten Speichers, sucht einen genügend großen Speicherblock und reserviert ihn.
- Der PSP wird aufgebaut wie oben.
- Die Code- und Datenteile werden geladen. Danach findet mit Hilfe einer Tabelle aus der Datei eine Relokation statt.
- Die Startadresse wird aus dem File-Header entnommen (sie muss natürlich auch reloziert werden). Das Programm wird durch Sprung an diese Adresse gestartet.

Ein Programm *beendet* sich durch Aufruf der DOS-Funktion Nummer 0 (im Interrupt `0x21`). In MS-DOS 2.0 wurde die Möglichkeit eingebaut, dem Elter-Prozess einen *Rückgabewert* zu übermitteln, indem stattdessen die Funktion `0x4c` verwendet wird. Benutzer-Programme können regulär überhaupt erst ab MS-DOS 2.0 selbst Prozesse starten, und zwar mit der Funktion `0x4b`.

Die bei diesem Verfahren zu verwaltenden Datenstrukturen sind etwas kompliziert. Von C aus sollte man daher immer die Bibliotheksfunktion exit und die aus der exec-Familie verwenden.

Beispiel: Mit der DOS-Funktion 0x62 (erst ab MS-DOS 3.0) kann man die Adresse des eigenen PSP ermitteln. Leider ist aber bei den üblichen C-Compilern keine Datenstruktur für den PSP definiert, sodass man immer mit Pointern und Offsets auf ihn zugreifen muss.

Der Offset der Adresse des Environments im PSP ist 0x2c. Das folgende Programm gibt das Environment des aktuellen Prozesses aus, so wie es das Kommando „set“ (ohne Parameter) tut.

```
#include <dos.h>
#include <stdio.h>

int main()
 {
   union REGS regs;
   char c, huge *psp, huge *E;

   regs.h.ah=0x62;              // Funktion 0x62: PSP-Adresse ermitteln
   intdos(&regs,&regs);         // PSP steht in BX:0000

   psp=(char huge *)((long)regs.x.bx<<4);
   E=(char huge *)((long)*(unsigned short huge *)(psp+0x2c)<<4);

   while (*E!=0)
    {
      while ((c=*E++)!=0) putchar(c);
      putchar('\n');
    }

   return 0;
 }
```

14.1.2 Vererbung

Ein Kind-Prozess erbt den Environment-Block des Elter-Prozesses, kann aber nachträglich Veränderungen an seiner Kopie durchführen. Das Weitergeben dieser Informationen an den Elter ist nicht möglich.

Das Kind erbt normalerweise auch alle offene Dateien des Elter, inklusive deren Modus und File-Position. Will man das für eine Datei unterbinden, muss der Elter sie mit einem speziellen Systemaufruf (aus io.h) öffnen:

DOS
int sopen(**const char** *path, **int** access, **int** shflag, **unsigned int** mode);
öffnet eine Datei mit speziellen **Shared**-Zugriffsbits.

Die Funktion ist auch für Zugriffe in einem Netzwerk gedacht. In unserem Zusammenhang ist der einzig sinnvolle Wert für shflag (außer 0) die symbolische Konstante O_NOINHERIT (definiert in share.h).

14.1.3 TSR-Programme

Um die Effekte der Abwesenheit echten Multitaskings zu mildern, wurde die Möglichkeit für eine Art von „Pop-Up-Programmen“ geschaffen (das bekannteste ist wahrscheinlich „`doskey`“). Sie werden meistens in `AUTOEXEC.BAT` gestartet und inaktiv im Speicher gehalten.

- Normalerweise wird bei Beendigung eines Prozesses (durch den DOS-Aufruf `0x4c`) sein gesamter Hauptspeicher wieder freigegeben. Wird das Programm dagegen mit der Funktion `0x31` beendet, bleibt der Speicher reserviert und behält seinen Inhalt. Dieser Aufruf heißt „**Terminate and Stay Resident**“ (TSR). Die Programme, die hiervon Gebrauch machen, wurden im Lauf der Zeit TSR-Programme genannt.

- Dieses Verfahren macht natürlich nur Sinn, wenn die zurückgebliebenen Programme auch wieder auf irgendeine Weise aufrufbar sind. Das geschieht normalerweise über eine spezielle Tastenkombination („**Hotkey**“), meistens mit Beteiligung von `CTRL` oder `ALT`. Auf diese Weise können – während die eigentliche Applikation läuft – kurzzeitig andere kleine Anwendungen zwischengeschaltet werden, etwa Kalender, Taschenrechner, etc. Nach deren Beendigung wird die Applikation wieder aktiv.

- Viel Unterstützung durch DOS-Funktionen erhalten diese Programme allerdings nicht. Damit der Hotkey-Aufruf funktioniert, müssen die TSRs bei ihrem ersten Start den Interrupt-Vektor manipulieren und den Tastatur-Interrupt auf eigenen Code umlenken. Außerdem müssen sie dafür sorgen, dass sie selbst (!) die Umgebung der laufenden Applikation vollständig wiederherstellen (z.B. Bildschirminhalt).

  Leider ist MS-DOS ja nicht re-entrant und ist deshalb von TSRs aus eigentlich nicht verwendbar! Wenn MS-DOS gerade aktiv war, als der Hotkey ausgelöst wurde, würde der Aufruf einer DOS-Routine eine Verschachtelung bedeuten und schnell zum Absturz führen.

  Es gibt aber undokumentierte DOS-Funktionen, auf die sich die meisten TSRs verlassen. Mit ihnen kann man den Zeiger auf den PSP des aktuellen Prozesses verbiegen und somit „echte“ DOS-Prozesse für sich nutzbar machen. Außerdem kann man mit ihnen feststellen, ob gerade DOS aktiv war. Dann manipuliert man den Programmzähler des Prozesses, lässt DOS zu Ende arbeiten und wird dann automatisch reaktiviert.

14.2 16-Bit-Windows

Windows bis zur Version 3.11 bietet *kein präemptives*, sondern *kooperatives* Multitasking, das noch aus der Zeit als einfacher DOS-Aufsatz stammt.

Es gibt keinen Scheduler im normalen Sinn. Das Multitasking wird hauptsächlich über den *Austausch von Nachrichten* zwischen Message-Queues von *Fenstern* gesteuert. Jeder Task (der mindestens ein Fenster steuert), erhält automatisch eine Message Queue.

Da wir uns hier nicht mit den Fenster-Details beschäftigen können, können wir auch das Task-Switching nur relativ oberflächlich betrachten.

- Windows unterscheidet folgende Begriffe:

  Programm-Modul: ausführbarer Code und initialisierte Daten in einer Programmdatei

  Instanz: geladene Kopie eines Moduls im Speicher

  Task: Instanz, die selbständig ausgeführt werden kann (inklusive zugewiesener Betriebsmittel, eine DOS-Umgebung mit Umgebungs-Variablen, etc.)

- In Windows beginnt die Ausführung eines Tasks immer bei der Funktion `WinMain`. Sie hat folgenden vorgeschriebenen Prototyp:

  ```
  int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
      LPSTR lpszCmdLine, int nCmdShow);
  ```

  `PASCAL` (in Win32 stattdessen `APIENTRY`) gibt an, dass die Parameter nicht wie in `C`, sondern wie in `PASCAL` übergeben werden sollen. (Als Windows entwickelt wurde, hatte sich, anders als auf allen anderen Systemen, auf Intel-PCs noch nicht `C` als beherrschende Sprache durchgesetzt.)

  Tasks, Fenster und Diverses mehr werden durch „*Handles*“ (Indizes in eine Windows-interne Tabelle) angesprochen. Der allgemeine Datentyp ist `HANDLE`; der Form nach gibt es speziellere wie `HWND` für Fenster, `HINSTANCE` für Instanzen, etc. (alle zunächst definiert in einer einzigen, gigantischen Header-Datei `windows.h`).

  Das Handle `hInstance` bezeichnet die eigene Instanz des benutzten Moduls. Es muss z.B. bei der Registrierung von Fenstern angegeben werden. Wenn noch eine früher gestartete Version desselben Moduls läuft (beide teilen sich den Code!), enthält `hPrevInstance` die entsprechende Instanz (sonst 0).

- Um von den speziellen Speichergrößen verschiedener Compiler unabhängig zu sein, definiert Microsoft in `windows.h` diverse weitere Typen wie `WORD` (16-Bit-Integer ohne Vorzeichen), `LPSTR` (segmentübergreifender Pointer auf einen 0-terminierten String), etc. Sie sind immer durch Großschreibung zu erkennen.

Es gibt folgende grundlegenden Funktionen:

Windows
`HANDLE GetCurrentTask();` liefert das Handle auf den aktuellen Task
`HANDLE GetWindowTask(HWND hwnd);` liefert das Handle des Tasks, der das Fenster mit dem Handle `hwnd` besitzt
`int GetNumTasks();` liefert die Gesamtanzahl von Tasks im System
`WORD GetCurrentPDB();` liefert die Segmentnummer der aktuellen „Program Data Base“ PDB (identisch mit dem PSP) zurück
`LPSTR GetDOSEnvironment();` liefert einen Zeiger auf den Environment-String des aktuellen Tasks (hauptsächlich in in DLLs sinnvoll).

Nur Windows selbst kann einen Task anlegen – nämlich durch Laden eines Moduls aus einer Programmdatei. Es gibt für ein Anwenderprogramm keine Möglichkeit, selbständig Tasks anzulegen (z.B. à la `fork`).

- Ein Task, der die CPU-Kontrolle erhält, kann *unbegrenzt* weiterlaufen und ggf. das gesamte System blockieren.
- Benutzeraktivitäten (Mausklicks in Fenster von nicht aktiven Tasks) werden über *Interrupts* von Windows *angenommen*. Sie gehen also nicht verloren – die anderen Tasks können aber (noch) nicht auf sie reagieren!
- Echte *Hintergrund*-Aktivitäten sind sehr schwer zu realisieren.

14.2.1 Nachrichten

Folgende Funktionen beschäftigen sich mit Nachrichten und dem Task-Switching:

Windows
`VOID Yield();` gibt die CPU-Kontrolle an Windows zurück
`BOOL GetMessage(LPMSG lpMsg, HWND hwnd, WORD uMsgFilterMin, WORD uMsgFilterMax);` liest eine Nachricht aus der Warteschlange des Fensters `hwnd` nach `lpMsg`, gibt bei leerer Schlange die CPU-Kontrolle zurück
`VOID WaitMessage();` gibt die CPU-Kontrolle zurück, falls die Warteschlange für Nachrichten leer ist
`BOOL PeekMessage(LPMSG lpMsg, HWND hwnd, WORD wMsgFilterMin, WORD wMsgFilterMax, WORD wRemoveMsg);` ähnlich wie `GetMessage`, Nachrichten brauchen aber nicht aus der Schlange gelöscht zu werden (`wRemoveMsg`=`PM_NOREMOVE` oder `PM_REMOVE`; beim Wert `PM_NOYIELD` wird nicht blockiert)
`BOOL TranslateMessage(LPMSG lpMsg);` setzt virtuelle Tasten-Nachrichten in Zeichen-Nachrichten um
`LONG DispatchMessage(LPMSG lpMsg);` übergibt eine Nachricht an die Fenster-Funktion (`WndProc`, s.u.)

Dabei ist `LPSMG` ein Zeiger auf eine `MSG`-Struktur. `uMsgFilterMin/Max` sind Grenz-Indizes in den Nachrichten-Puffer.

Das kooperative 16-Bit-Windows-Multitasking ist dadurch definiert, wie ein Prozess, der einmal die CPU-Kontrolle erhalten hat, diese wieder abgeben kann. Das geschieht auf vier mögliche Weisen – die ersten drei sind:

1. Er ist zu Ende.
2. Er wartet auf eine Nachricht mit `GetMessage`, `WaitMessage` oder `PeekMessage` (ohne `PM_NOYIELD`).

3. Er ruft die Funktion `Yield` auf (günstig bei längeren Berechnungsaktionen) – `Yield` kann aber immer durch `PeekMessage` ersetzt werden.

In allen Fällen geht die CPU-Kontrolle an Windows zurück, das nach diversen Verwaltungsaktionen (z.B. auch Veränderungen an den DOS-Strukturen wie PSPs!) einen anderen Task aufweckt:

- wenn für einen wartenden Task (`Message`-Funktionen) eine Nachricht vorliegt, irgendeinen von diesen,
- ansonsten irgendeinen, der `Yield` aufgerufen hatte.

Diese Task-Auswahl ist eher zufällig. Es gibt beispielsweise keine Task-Prioritäten!

Die vierte Möglichkeit ist folgende:

4. Der Task schickt mit `SendMessage` einem Fenster eine Nachricht.

Das Verhalten von Windows in diesem Fall ist etwas kompliziert.

- Windows aktiviert den Task, dem das Fenster gehört. Es landet dort (meist) bei einem `GetMessage` o.ä. Der jetzt laufende Task ruft `DispatchMessage` auf und damit die Handler-Routine des angesprochenen Fensters.
- Wenn die Fenster-Routine zu Ende ist, geht die Kontrolle zurück an den Task, der die Nachricht geschickt hatte!
- Die Handler-Routine sollte normalerweise schnell wieder beendet sein. Wird dort aber `ReplyMessage` aufgerufen, reißt der Fenster-Task die Kontrolle ganz an sich!

Die Funktion zur Nachrichten-Behandlung wird beim Anlegen eines Fensters „mitregistriert“ d.h. ihre Adresse wird in die interne Window-Struktur eingetragen. Bei ihr kommen meist Nachrichten an, die Windows selbst verschickt (bei Größenänderung, Mausklicks, Tastatur-Ereignissen, etc.)

Beispiel: Das folgende kurze 16-Bit-Programm öffnet ein Fenster und zeichnet die beiden Diagonalen ein. Sie passen sich einer Größenänderung automatisch an. Mit dem Schließbutton des Fensters (oder äquivalent `ALT-F4`) wird das Programm beendet.

Durch die technischen Details wird der Quelltext schon vergleichsweise lang (das `.EXE`-File wird aber nur ca. 4KByte groß). Die oben nicht aufgeführten Funktion erklären sich aus dem Zusammenhang selbst.

```
#include <windows.h>

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
 {
   static int width,height;                    // wird bei WM_SIZE gesetzt
   PAINTSTRUCT lpPaint;                        // für WM_PAINT benötigt
   HDC hdc;                                    // für WM_PAINT benötigt
```

```
  switch (iMsg)                                  // Verteilung der Nachrichten
   {
     case WM_SIZE:                               // Größenänderung
        width=LOWORD(lParam);                    // Breite aus lParam herausholen
        height=HIWORD(lParam);                   // Höhe aus lParam herausholen
        return 0;
     case WM_PAINT:                              // Update des Inhalts notwendig
        hdc=BeginPaint(hwnd,&lpPaint);           // Update anmelden
        Rectangle(hdc,0,0,width,height);         // Inhalt löschen + Rahmen
        MoveToEx(hdc,0,0,NULL);                  // erste Diagonale
        LineTo(hdc,width-1,height-1);
        MoveToEx(hdc,0,height-1,NULL);           // zweite Diagonale
        LineTo(hdc,width-1,0);
        EndPaint(hwnd,&lpPaint);                 // Update abmelden
        return 0;
     case WM_DESTROY:                            // Fenster schließen
        PostQuitMessage(0);                      // QUIT-Message verschicken
        break;                                   // "fall-through" zum DefProc
   }

  return DefWindowProc(hwnd,iMsg,wParam,lParam);    // Standard-Reaktion
}

int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR     lpCmdLine, int       nCmdShow)
{
  MSG msg;
  HWND hwnd;
  static char szAppName[]="WinTest";

  static WNDCLASS wndclass=                      // Struktur zum Registrieren
   {                                             // einer neuen Fenster-"Klasse"
    CS_HREDRAW|CS_VREDRAW|CS_DBLCLKS,
    WndProc,                                     // zugehörige Fenster-Prozedur
    0, 0, 0, 0, 0, 0, NULL, szAppName
   };

  wndclass.hInstance=hInstance;
  wndclass.hbrBackground=(HBRUSH)GetStockObject(NULL_BRUSH);
  wndclass.hCursor=LoadCursor(NULL,IDC_ARROW);
  RegisterClass(&wndclass);                      // Fensterklasse registrieren

  hwnd=CreateWindow(szAppName,szAppName,WS_OVERLAPPEDWINDOW,
     CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
  if (hwnd==NULL) return 0;
  ShowWindow(hwnd,nCmdShow);                     // Fenster anzeigen
  UpdateWindow(hwnd);                            // und neu aufbauen (WM_PAINT)

  while (GetMessage(&msg, NULL, 0, 0))           // Nachrichten abholen
   {
```

```
    if (msg.message==WM_QUIT) break;          // bei QUIT Programm beenden
    TranslateMessage(&msg);                   // Tastendrücke übersetzen
    DispatchMessage(&msg);                    // an WndProc(s) weiterleiten
  }

  return msg.wParam;                          // Return-Wert aus der QUIT-Msg.
}
```

`wParam` und `lParam` sind 16- bzw. 32-Bit-Parameter, deren Bedeutung vom Typ der Nachricht abhängt.

Bei längeren Berechnungen sollte man in kurzen Abständen nach Nachrichten schauen, z.B. alle 1 oder 2 Sekunden. Ansonsten kann auch nicht auf Ereignisse wie Mausklicks reagiert werden, was den Benutzer im allgemeinen schnell irritiert:

```
int PASCAL WinMain(...)
 {
   ...
   for (;;)
    {
      while (PeekMessage(&msg,0,0,0,P_REMOVE))
       {
         if (msg.message==WM_QUIT) goto Quit;
         TranslateMessage(&msg);
         DispatchMessage(&msg);
       }
      ...          /* Berechnung, z.B. eine Zeile einer Grafik berechnen     */
    }

   Quit: ...
 }
```

In 16-Bit-Windows gibt es leider nur eine einzige, systemweite Warteschlange für Ereignisse. Wenn ein Prozess keine Nachrichten mehr abholt, erhält niemand – auch Windows selbst nicht – die weiteren Nachrichten. Man kann dann auch keine Fenster mehr verschieben oder nach vorn holen, etc.

Win32 führt für jeden Prozess eine eigene Warteschlange, genauer:

- eine *gemeinsame* Warteschlange für alle 16-Bit-Anwendungen,
- eine private Warteschlange für jeden *Thread*.

Auf diese Weise kann eine Amok laufende 16-Bit-Anwendung zwar *alle anderen 16-Bit-Anwendungen* blockieren, aber nicht mehr das ganze System.

14.2.2 Signale

Unter Windows gibt es kein direktes Analogon zu den UNIX-Signalen. Ihre Funktionalität kann manchmal aber annähernd mit Nachrichten erreicht werden.

Beispielsweise kann man `SIGINT` mit dem Verschicken der Nachricht `WM_QUIT` verglichen werden.

Das Signal SIGALRM und der Aufruf alarm kann mit den folgenden Funktionen simuliert werden:

Windows
`WORD SetTimer(HWND hwnd, int nIDEvent, WORD wElapse, FARPROC lpTimerFunc);` weist Windows an, dem Fenster alle `wElapse` Millisekunden eine `WM_TIMER`-Nachricht zu schicken. Gibt die ID des Timers zurück.
`BOOL KillTimer(HWND hwnd, nIDEvent);` schaltet die periodische Nachrichten-Erzeugung ab. `nIDEvent` ist der Rückgabewert des entsprechenden SetTimer-Aufrufs.

Dabei ist `wParam`=`nIDEvent`, womit mehrere Timer mit verschiedenen Perioden unterschieden werden könnten. Wenn `lpTimerFunc`≠`NULL`, muss es die Adresse einer Fensterfunktion sein („Rückruffunktion“), die dann beim Timer-Ereignis von `DispatchMessage` direkt aufgerufen wird.

Timer-Perioden werden auf Vielfache von internen „Ticks“ (54.925 ms) gerundet. Kürzere Perioden als ein Tick sind nicht möglich.

Vorsicht: Der empfangende Task wird *nicht* (wie bei den UNIX-Signalen) direkt unterbrochen. Auch Nachrichten wie `WM_QUIT` oder `WM_TIMER` landen in der normalen Warteschlange! Wenn der Task mit Berechnungen beschäftigt ist und die Nachrichten nicht abfragt, bemerkt er nichts. (Ausnahme: DLLs dürfen direkt Hardware-Timer-Signale empfangen.)

Beispiel: Ohne Rückruffunktion kann das ganze wie folgt aussehen:

```
int PASCAL WinMain(...)
 {
   ...
   WORD nId1, nId2;
   nId1=SetTimer(hwnd,1,1000,NULL);
   nId2=SetTimer(hwnd,10,10000,NULL);
   ...
   KillTimer(hwnd,nId1);
   KillTimer(hwnd,nId2);
 }

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
 {
   switch (iMsg)
    {
      ...
      case WM_TIMER:
         switch (wParam)
          {
            case 1: ...       /* Signal vom Sekunden-Timer   */
            case 10: ...      /* Signal vom Zehn-Sekunden-Timer    */
          }
    }
   return DefWindowProc(hwnd,message,wParam,lParam);
 }
```

14.3 32-Bit-Windows

In Win32 (in Windows 95$^+$ und Windows NT) ist präemptives Multitasking (mit Timeslicing) und sogar Multithreading implementiert. Aber Vorsicht:

- In Windows 95 werden 16-Bit-Programme noch genauso behandelt wie in den 16-Bit-Windows-Versionen. Wenn ein 16-Bit-Task die CPU-Kontrolle hat, kann er sie also endlos behalten und alle anderen 16-Bit-Tasks sperren.
- In Windows NT laufen 16-Bit-Programme dagegen in einer speziellen Umgebung und werden in das Time-Slicing einbezogen.

14.3.1 Threads

Jeder Task besteht am Anfang aus einem einzigen Thread („*primary thread*“). Er kann beliebig viele Threads selbst neu erzeugen, die sich die Task-Ressourcen teilen (offene Dateien, Speicher, insbesondere statische Variablen), aber einen eigenen Stack und Prozessor-Status haben. Mit TLS-Funktionen („*Thread Local Memory*“) kann man einzelnen Threads eigenen statischen Speicher zuordnen. *Jeder* Thread erhält eine eigene Message Queue. Threads können sich gegenseitig beenden.

Die normale Thread-Steuerung erfolgt mit den Funktionen `_beginthread` und `_endthread` (beides aus `process.h`). Es muss außerdem mit einer Compiler-Option (wie „multithreaded“) bekanntgegeben werden, dass Threads benutzt werden.

	Win32
unsigned long _beginthread**(void (***func**)(void *), unsigned int** ss**,** **void ***par**);** startet einen neuen Thread, der mit der Ausführung der Funktion `func` beginnt. `ss` gibt die Stackgröße an (0=Stackgröße des Elter). `par` ist ein Parameter, der der Funktion `func` beim Start übergeben wird.	
void _endthread**(void);** beendet den aktuellen Thread.	

```
// include stdio windows process

void MyThread(void *str)
 {
   printf("Start von Thread %s!\n",str);
   ...
   _endthread();
 }

int main(int argc, char *argv[])
 {
   _beginthread(MyThread,0,"1");
   _beginthread(MyThread,0,"2");
   ...
 }
```

14.3.2 Thread-Synchronisation

Die Win32-Threads haben direkten Zugriff auf die Ressourcen ihres zugrundeliegenden Tasks. Zur Kollisionsvermeidung bietet Win32 einige Funktionen für die Verwaltung von kritischen Bereichen. Als Datentyp wird dazu die Struktur `CRITICAL_SECTION` verwendet:

Win32
`void InitializeCriticalSection(LPCRITICAL_SECTION lpCriticalSection);` initialisiert eine CS-Struktur
`void EnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection);` Betreten eines kritischen Bereichs, blockiert ggf.
`BOOL TryEnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection);` Versuch, einen kritischen Bereich zu betreten – blockiert nicht (gibt dann `FALSE` zurück)
`void LeaveCriticalSection(LPCRITICAL_SECTION lpCriticalSection);` Verlassen eines kritischen Bereichs
`void DeleteCriticalSection(LPCRITICAL_SECTION lpCriticalSection);` Auflösen einer CS-Struktur

Es darf immer nur maximal ein Thread einen `Enter`-Aufruf abgesetzt haben. Weitere Threads werden beim `Enter`-Aufruf blockiert, bis der andere einen `Leave`-Aufruf tätigt.

```
CRITICAL_SECTION cs;
InitializeCriticalSection(&cs);

for (;;)
 {
   EnterCriticalSection(&cs);
   ...            // kritischer Bereich
   LeaveCriticalSection(&cs);
   ...            // unkritischer Bereich
 }

DeleteCriticalSection(&cs);
```

Außerdem gibt es System-Objekte für zählende Semaphore und binäre Semaphore (Mutex).

Win32
`HANDLE CreateSemaphore(LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,` `long lInitialCount, long lMaximumCount, LPCWSTR lpName);`
`HANDLE OpenSemaphore(DWORD dwDesiredAccess, BOOL bInheritHandle,` `LPCSTR lpName);`
`BOOL ReleaseSemaphore(HANDLE hSemaphore, long lReleaseCount,` `long *lpPreviousCount);`
`HANDLE CreateMutex(LPSECURITY_ATTRIBUTES lpMutexAttributes,` `BOOL bInitialOwner, LPCSTR lpName);`
`HANDLE OpenMutex(DWORD dwDesiredAccess, BOOL bInheritHandle,` `LPCSTR lpName);`
`BOOL ReleaseMutex(HANDLE hMutex);`

14.3.3 Thread-Scheduling

Win32 arbeitet mit einer Art Priority-Scheduling. Eine Priorität ist dabei eine Zahl von 0 bis 31 (größere Zahl=bevorzugter Prozess).

Jeder Thread erhält eine „Basis-Priorität“, die die Grundlage für die Berechnung der aktuellen „echten“ Priorität bildet. In diese Rechnung gehen diverse situations-spezifische Werte ein.

Die Prozesse werden in vier Prioritäts-Klassen aufgeteilt und vom Scheduler in vier Schlangen verwaltet: **Idle**, **Normal**, **High** und **Real-Time**. Die ihnen untergeordneten Threads erhalten (wenn man sie nicht explizit ändert) eine entsprechende Basis-Priorität.

Real-Time ist nur für sehr hardwarenahe Threads (von Treibern erzeugt) gedacht. Der Kernel selbst läuft unter High (Threads mit Basis-Priorität 13), normal gestartete Anwendungen unter Normal (8).

Der Task-Manager von Windows 95 (`taskman`) zeigt nicht alle Prozesse (oder gar Threads) an. System-Prozesse haben das Attribut „hidden“ und werden immer versteckt. Es gibt aber spezielle Tools, die Informationen über alle Prozesse und die ihnen untergeordneten Threads grafisch anzeigen (etwa `sinfo`). Sie liefern beispielsweise folgende Ausgabe (viele Spalten ausgelassen):

Process	Priority	Module	Size	Type	Used Modules
`KERNEL32`	high	`KERNEL32.DLL`	434176	32-bit	`USER32.DLL, GDI32.DLL ...`
8	13				
...					
1	13				
`QUICKRES`	normal	`QUICKRES.EXE`	19392	16-bit	`KERNEL32.DLL`
1	8				
`EXPLORER`	normal	`EXPLORER.EXE`	217088	32-bit	`VERSION.DLL, OLE32.DLL ...`
2	8				
1	8				

Das präemptive Multitasking von Win32 läuft üblicherweise mit einer Zeitscheibe von 20 Millisekunden ab, die aber noch unter verschiedenen Threads aufgeteilt werden kann.

Win32 unterscheidet zwischen zwei verschiedenen Scheduler-Stufen:

- Der **primäre Scheduler** wird nach Ablauf einer Zeitscheibe aufgerufen und ordnet jedem Thread eine aktuelle Priorität zu. Alle Threads mit der höchsten Priorität werden für die nächste Zeitscheibe zugelassen.
- Der **Zeitscheiben-Scheduler** bestimmt aus den Prioritäten (und abhängig vom internen Zustand des Systems) den Prozentsatz der Zeitscheibe, die die zugelassenen Threads erhalten.

Man kann einem Thread auch das Attribut „exklusiv“ geben. Er wird dann immer eine ganze Zeitscheibe lang laufen (aber dann eben entsprechend seltener).

Das Scheduling bemüht sich redlich um die faire Vergabe der Prozessorzeit:

- Ein Tastendruck oder ein Mausklick erhöht die Priorität des zugehörigen Threads.

- Der Besitz einer wichtigen exklusiven Ressource, die bereits anderweitig nachgefragt wird, erhöht die Priorität (bis zur Freigabe).
- Virtuelle DOS-Maschinen erhalten höhere Priorität, wenn sie im Ganzbildschirm-Modus oder im aktiven Fenster laufen, sehr niedrige sonst.
- Die Priorität lang laufender Threads wird allmählich abgesenkt.

14.3.4 Thread-Abstürze

Win32 versucht Sorge dafür zu tragen, dass das gesamte System unter dem Absturz eines einzelnen Threads nicht leidet. Abstürze können durch diverse Programmfehler ausgelöst werden (z.B. unzulässiger Speicherzugriff, illegaler oder privilegierter Befehl).

Dazu läuft die Fehlerbehandlung als eigener Thread, der normalerweise selbst nicht unter den Auswirkungen des Fehlers im auslösenden Thread leiden kann.

Außerdem wird für alle belegten Ressourcen (Speicher, geöffnete Dateien, Geräte, Fenster) mitgespeichert, welcher Thread sie belegt hat. Im Absturzfall können sie dann freigegeben werden und stehen anderen Threads wieder zur Verfügung.

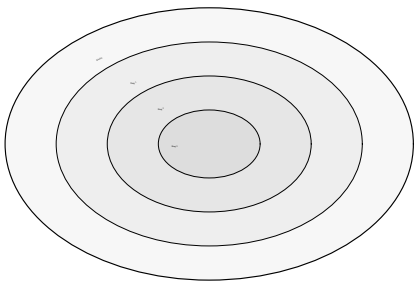
14.4 Der Aufbau von Windows 95

Die grobe Struktur der verschiedenen Windows-Versionen hatten wir bereits in Abschnitt 2.6 betrachtet. In diesem Kapitel wollen wir uns etwas genauer allerdings nur mit der am weitesten verbreiteten Version, nämlich Windows 95, beschäftigen. Windows 98 bietet keine wesentlichen Neuerungen auf der eigentlichen System-Ebene.

14.4.1 Ringe

Die Intel-Prozessoren unterstützen seit dem 80386 privilegierte Befehle. Sie sollen also dem Normalbenutzer nicht zur Verfügung stehen, sondern sind dem System vorbehalten. Werden diese Befehle im Normal-Modus verwendet, lösen sie eine Fehlerbehandlung (durch eine „Schutzverletzung“) aus. Anders als bei den meisten Systeme werden nicht nur *zwei* Privilegstufen („dual-mode-instructions“) unterschieden, sondern *vier* – die vier sogenannten *Ringe* 0 bis 3.

Durch ineinanderliegende Ringe wird die Teilmengen-Beziehung der jeweils erlaubten Befehle widergespiegelt.



- Ring 0 ist die höchste Privilegstufe, in der alle Befehle verfügbar sind. Die Entwickler der Prozessoren haben ihn nicht für das gesamte Betriebssystem vorgesehen, sondern nur für den kritischsten hardwarenächsten Bereich.
- Ring 3 ist die niedrigste Privilegstufe. Sie ist für normale Anwender-Programme gedacht, die nicht direkt auf Teile des Betriebssystems – und schon gar nicht direkt auf die Hardware – zugreifen können sollen.

Windows benutzt nur die beiden **Ringe 0 und 3**, betreibt also doch ein Dual-Mode-System (manchmal als „Kernel/User-Modell“ bezeichnet). Benutzer-Programme (Windows- oder MS-DOS-Anwendungen) laufen natürlich immer im Ring 3, ebenso aber einige hardwareferne Teile des Systems selbst.

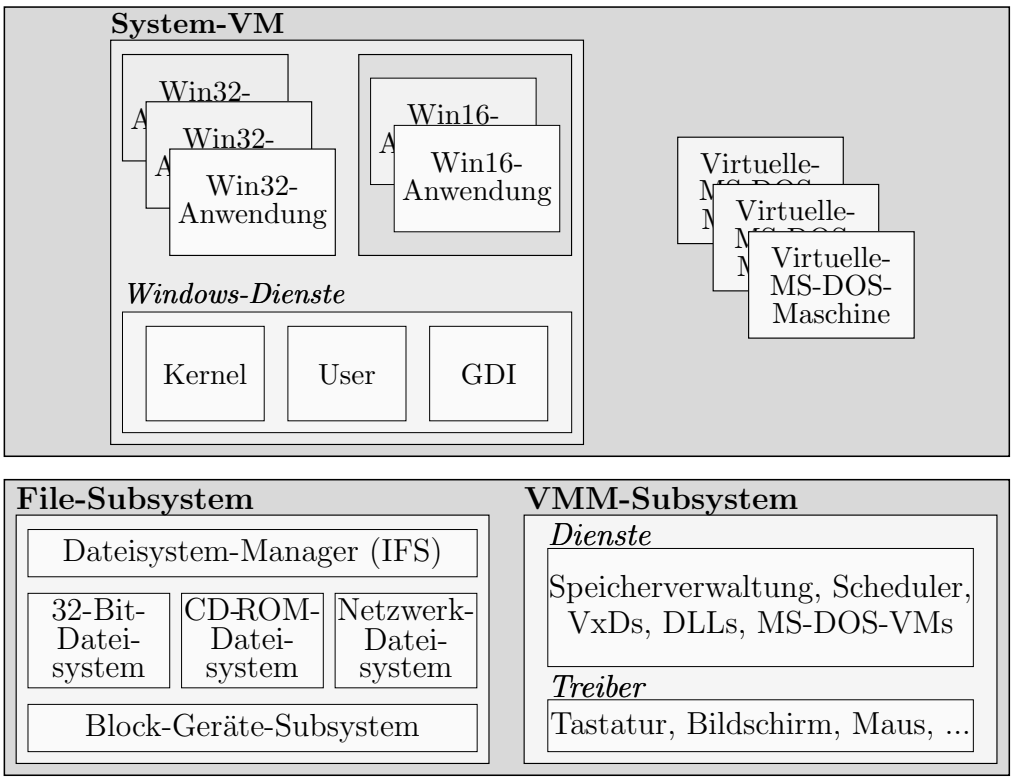
Eigentlich gehören nur die Teile der Treiber, die auf die Hardware direkt zugreifen, in den Ring 0. Ebenso haben eigentlich Hardware-Abstraktionen wie die Dateisysteme nichts im Ring 0 zu suchen. Für solche Dinge hatte Intel die Ringe 1 und 2 vorgesehen.

Es gibt beim 80x86 vier Möglichkeiten („*Gates*“), den Ring zu wechseln: *Call* (Systemaufruf), *Interrupt*, *Task* (Wechsel zu anders privilegiertem Task), *Trap* (Fehlerbehandlung). Bei Hin- und Rückwechsel sind einige Daten zu sichern bzw. wiederherzustellen. Da dies alles einiges an Zeit kostet, hat man sich für das vorliegende Design mit nur zwei Ringen entschieden.

Die entstehende Struktur ist im Diagramm auf der nächsten Seite dargestellt. Es bedeuten:

System-VM: Die virtuelle Maschine, die alle Windows-Anwenderprogramme und die nicht privilegierten Systemteile als Protected-Mode-Umgebung realisiert.

Win32-Anwendungen: 32-Bit-Programme, die also den Prozessor im Protected Mode sehen, und die die System-Schnittstelle Win32 verwenden. Hierunter fallen auch Windows-Bestandteile wie der Explorer, Editoren, etc. Jede solche Anwendung erhält ihren eigenen privaten Adressraum. Sie werden mit präemptivem Multitasking betrieben.



Win16-Anwendungen: 16-Bit-Programme, die den Prozessor wie einen 80286 (mit Speichersegmentierung) sehen und die alte Windows-Schnittstelle ansprechen. Sie laufen in einem

eigenen *gemeinsamen* Adressraum, den man als einen virtuellen Windows-3.1-Rechner ansehen kann. Sie arbeiten mit kooperativem Multitasking.

Virtuelle MS-DOS-Maschinen: Komplette virtuelle MS-DOS-Rechner, die mit Hilfe des virtuellen 8086-Modus betrieben werden. Der Benutzer kann explizit zwischen ihnen (und Windows-Anwendungen) hin- und herschalten. Ansonsten ist das entstehende Multitasking extrem langsam, aber fast präemptiv.

Windows-Dienste: Hier liegen drei Module.

Der **Kernel** kann (anders als im üblichen Betriebssystem-Sinn) als grober Steuermechanismus und Schnittstelle zu den hardwarenahen Teilen im Ring 0 gesehen werden.

User umfasst die Verwaltung der Fenster und anderer Einzelteilen der Benutzeroberfläche.

GDI beinhaltet die grafische Unterstützung auf niedrigerer Ebene, also Zeichen-Primitiva, Zeichensätze, Farben, etc.

Alle drei Module liegen in einer 16-Bit und einer 32-Bit-Version vor. Oft nehmen die 16-Bit-Versionen nur eine Umsetzung in 32-Bit-Funktionsaufrufe vor. Wenn aber eine 32-Bit-Version einer Funktion nur *länger* geworden wäre (durch 32-Bit-Adressen), aber nicht *schneller*, hat man weiterhin mit 16-Bit-Versionen gearbeitet.

File-Subsystem: In alten Versionen stützte sich Windows noch auf die Dateiverwaltung von MS-DOS. Windows 95 hat diese Aufgabe endlich selbst übernommen, was wesentliche Performance- und Sicherheits-Vorteile bietet. IFS steht für *Installable File System* und dient ähnlich wie das VFS bei UNIX als abstrakte Schnittstelle zu unterliegenden konkreten Filesystemen auf unterschiedlichen Geräten (siehe Seite 377).

VMM: Der Virtual Machine Manager ist das eigentliche Kernstück des Betriebssystems. Das Steuern der Prozesse, die Verwaltung von Speicher und Geräten erfolgt hier. Mehr als die Hälfte der über 700 API-Funktionen von Windows liegen im VMM.

14.4.2 Virtuelle MS-DOS-Maschinen

14.4.2.1 Der globale Kontext

Nach dem Start, und bevor Windows 95 in den Protected Mode umschaltet, läuft der Prozessor im Real Mode, verhält sich also wie ein 8086. Zum Initialisieren wird ein Modul namens `WINBOOT.SYS` verwendet, das sich an dieser Stelle wie eine vollständige MS-DOS-Version verhält. Es werden `CONFIG.SYS` und `AUTOEXEC.BAT` ausgeführt.

Wenn Windows 95 die Maschine in den Protected Mode überführt, wird eine Momentaufnahme der 8086-Maschine gemacht und aufbewahrt, insbesondere ein Speicherabbild der relevanten Teile des untersten Megabytes Speicher, inklusive der dort abgespeicherten Treiber, Umgebungsvariablen, TSRs, etc. Diese Momentaufnahme heißt „*globaler Kontext*“ der virtuellen Maschinen.

Beim Start eines MS-DOS-Programms wird eine virtuelle Maschine eingerichtet. Es wird dabei ein Megabyte Speicher reserviert und mit der gemachten Momentaufnahme initialisiert. Der

Speicher kann an beliebigen Stellen des physischen Hauptspeichers liegen und auch auf Platte ausgelagert werden: Über Mechanismen des Memory Management wird der Adressbereich hardwaremäßig auf den „gewohnten Adressbereich“ von 0 bis 1 MByte abgebildet.

Der globale Kontext wird nach dem Einrichten nicht mehr verändert und ist für den Benutzer auch nie selbst sichtbar oder erreichbar. Es werden also *nicht* jedesmal die Startdateien ausgeführt. Wenn man an ihnen Änderungen vornimmt (wie das Setzen von Umgebungsvariablen), hat das erst beim nächsten Start von Windows Auswirkungen. Wenn man eine Umgebungsvariable verändert, hat das nur Auswirkungen innerhalb der aktuellen virtuellen Maschine – nicht auf andere, gleichzeitig laufende, und auch nicht auf später gestartete. (Es gibt aber inzwischen ein Utility `winset` – analog zu `set` – das Variablen des globalen Kontextes nachträglich verändert – mit Auswirkungen auf *danach* gestartete virtuelle Maschinen.)

14.4.2.2 MS-DOS-Aufrufe

MS-DOS-Aufrufe erfolgen durch Software-Auslösung des Interrupts 21H, der unter Windows 95 (von Protected-Mode-Code im Kernel) abgefangen wird.

Aufrufe, die Dateien betreffen, landen dann nicht beim MS-DOS-Code in der ursprünglichen MS-DOS-Umgebung, sondern werden in Aufrufe des Filesystem-Managers umgesetzt. Nur so können beispielsweise mehrere virtuelle Maschinen „gleichzeitig“ auf die Platte zugreifen.

Aufrufe, die den Bildschirmaufbau betreffen, werden in Aufrufe des User-Moduls umgesetzt, sodass die Ausgabe (außer bei bestimmten Grafik-Modi) auch in einem Fenster erscheinen kann.

14.4.2.3 DPMI

Späte MS-DOS-Programme laufen nun aber leider nicht ausschließlich im Real Mode des 8086. Um auf den Speicher über 1 MByte zugreifen zu können, wurden viele Mechanismen entworfen, die letztendlich kurzfristig in den Protected Mode (von 80286 oder 80386) schalten müssen. Der wichtigste (da von Microsoft offiziell unterstützte) ist DPMI (DOS Protected Mode Interface).

Die Funktionen des DPMI sind über den Interrupt 31H erreichbar und bilden zusammen den „DPMI-Host“. Ein Programm, das sie benutzt, ist ein „DPMI-Client“.

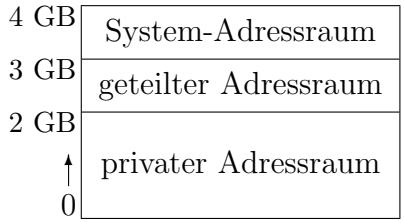
Da MS-DOS selbst nicht im Protected Mode laufen kann, dürfen Programme, die DPMI benutzen, nie direkt auf DOS-Funktionen zugreifen. Daher bedienen sie sich meist eines „DOS-Extenders“, das Umsetzungen auf DPMI-Funktionen übernimmt und einige weitere Dienste anbietet.

Um das Funktionieren solcher DOS-Programme zu ermöglichen, müssen in Windows 95 die DPMI-Interrupt-Aufrufe natürlich ebenfalls abgefangen und in Aufrufe der Windows-Module umgesetzt werden. Windows stellt also einen eigenen DPMI-Host zur Verfügung.

14.4.3 Adressräume

Unter Windows 95 hat jeder Prozess durch den Protected Mode einen möglichen logischen Adressraum von 4 GByte (2^{32} Byte wegen 32-Bit-Adressen), der durch Mechanismen von virtuellem Speicher fast beliebig auf Hauptspeicher- oder Swap-Teile abgebildet werden kann. Er ist immer in drei Teile aufgeteilt:

Die untersten (logischen!) 2 GByte stehen allein der Anwendung zur Verfügung und sind von außen (außer für den VMM) nicht sichtbar. Ein weiteres GByte ist für die gemeinsame Verwendung mehrerer Prozesse bestimmt (z.B. Nicht-System-DLLs). Das oberste GByte dient der Kommunikation mit dem System (hier werden z.B. System-DLLs eingeblendet).



- Code-Teile, Datenteile und Stack werden in den untersten 2 GByte untergebracht. Der Einsprungpunkt für 32-Bit-Anwendungen liegt üblicherweise bei 4 MB (also bei der logischen Adresse `003fffffH`), das kann man bei Bedarf aber (z.B. als Linker-Einstellung) variieren. Die untersten 16 KByte dürfen nie verwendet werden. Ein Zugriff führt zu einer Schutzverletzung (so können beispielsweise Fehlzugriffe mittels Nullpointer entdeckt werden).

- Bei dynamischer Anforderung von mehr Speicher (vom Heap) werden zusätzliche Speicherseiten in den Adressraum eingeblendet (bei der Anforderung von 64 KByte beispielsweise 16 Speicherseiten à 4 KByte).

  Windows 95 unterscheidet dabei zwei Heaps: den *lokalen* und den *globalen*. Anfragen zum lokalen Heap werden letztendlich auf den globalen zurückgeführt. Die Verwaltung des lokalen Heaps ist aber eventuell schneller, weil sie blockweise Speicher anfordert und nicht jedesmal die globalen Strukturen bemühen muss. Man kann sogar *mehrere* private lokale Heaps (für verschiedene Zwecke) anlegen lassen (mit dem API-Aufruf `HeapCreate()`).

- Die Anforderung von Speicher wirkt sich zunächst nur auf die internen Verwaltungsstrukturen aus, die den tatsächlich verwendeten Teil des 4 GByte-Adressraums darstellen. Die physischen Speicherseiten werden erst dann belegt, wenn sie das erste Mal *angesprochen* werden.

  Wenn man also 1 GByte Speicher anfordert und in das erste und letzte Byte darin schreibt, werden tatsächlich nur 2 Speicherseiten mit zusammen 8 KByte belegt.

- Es gibt für ein Anwenderprogramm keine Möglichkeit, physischen Speicher ab einer bestimmten Adresse zu belegen. Manche System-Interna benötigen dies allerdings, beispielsweise Geräte-Treiber. Der Interrupt-Controller kann beispielsweise nur auf den unteren Speicherbereich zugreifen. I/O- und DMA-Puffer müssen also unbedingt dort liegen.

- Wie in System-V-UNIX können außerdem *Dateien* in den Adressbereich eines Prozesses (oder mehrerer) *eingeblendet* werden (zwischen 2 und 3 GByte), siehe Seite 380.

14.4.4 Der Virtual Machine Manager

Der VMM ist die Kern-Komponente von Windows 95 und bleibt permanent im Hauptspeicher. Er bietet über 300 Funktionen an, die hauptsächlich vom Kernel und von Gerätetreibern aufgerufen werden. In Assembler sieht ein solcher Aufruf vom Prinzip wie folgt aus:

```
MOV     ESI,<Parameter1>
MOV     EDX,<Parameter2>
VMMcall <Nummer der VMM-Funktion>
```

Das Makro `VMMcall` erzeugt eine Anweisungsfolge, die letztendlich ein 80386-Gate auslöst.

Wie sein Name andeutet, verwaltet der VMM die virtuellen Maschinen unter Windows. Um dies zu können, gehen seine Befugnisse aber sehr weit. Er bietet u.a. Dienste zu:

- Speicherverwaltung: Belegung und Freigabe von physischem und virtuellem Speicher, Informationen über Speicherzustände
- Scheduling: Veränderung von Prioritäten von Gerätetreibern aus
- Synchronisation: Verwaltung von Semaphoren, Mutexes und kritischen Bereichen
- I/O-Trapping: zur Kontrolle einzelner I/O-Ports von Gerätetreibern aus
- Ereignis-Koordination: Verteilung der Ereignisse an die Threads, Registrierung von Rückrufdiensten (die bei Auftreten eines Ereignisses automatisch aufgerufen werden sollen)
- Gerätetreiber-Kommunikation (z.B. Schnittstellen zum Aufruf von MS-DOS-Treibern vom Protected Mode aus)
- Registry: zur Manipulation der Registry von Gerätetreibern aus
- Debugging: zur System-Inspektion für hardwarenahe Debugger

14.4.5 Die Registry

Die **Registry** (oder „*Registrierung*“) ist eine Windows-interne Datenbank, die viele Daten über die Konfiguration des Systems sammelt.

In älteren Windows-Versionen wurde ausschließlich mit `.INI`-Dateien gearbeitet, einer für jedes Programm, jeden Treiber, etc. Dabei handelte es sich um Textdateien, die mit `[]` in Abschnitte gegliedert waren und Definitionen der Art „Name=Wert“ enthielten, z.B. (aus `WIN.INI`):

```
[windows]
device=Universal/Nur Text,TTY,LPT1:

[FontSubstitutes]
Times=Times New Roman

[colors]
ButtonShadow=128 128 128
```

- Bei Microsoft war man der Auffassung, dass die Systemverwaltung mit einer Vielzahl einzelner Konfigurationsdateien zu unübersichtlich sei (so wie beim Verzeichnis `/etc` in UNIX). Daher packte man bei der Entwicklung von Windows NT fast alle relevanten Daten in eine Sammlung von strukturierten Binärdateien, die Registry.

  Außerdem ist die Registry wie ein Dateisystem hierarchisch gegliedert und daher (vielleicht) übersichtlicher, während die `.INI`-Dateien nur eine Unterteilung auf einer Ebene boten.

- Da unter Windows 95 auch noch alte Gerätetreiber (mit eigenen Dateien, sogar DOS-Treiber) unterstützt werden sollten, wählte man hier ein Gemisch aus Konfigurationsdateien und einer Registry, was insgesamt wiederum nicht zur Übersicht beiträgt.

Windows NT stellt nur noch `SYSTEM.INI` und `WIN.INI` zur Verfügung, um 16-Bit-Anwendungen zu unterstützen. Außerdem kann man Zugriffe auf `.INI`-Dateien automatisch auf die Registry umleiten lassen.

- Die Registry wird zwar physisch als mehrere Dateien gespeichert, der Zugriffsmechanismus versteckt diese aber unter einer einheitlichen Schnittstelle.

  Bei Windows NT liegen diese Dateien in `system32\config` (als `SYSTEM`, `ADMINxxx`, `USERxxx`, `SOFTWARE.LOG` und mehr), bei Windows 95 dagegen im Windows-Hauptverzeichnis (als `SYSTEM.DAT` und `USER.DAT`).

  `SYSTEM.DAT` enthält alle Informationen, die sich auf den Rechner beziehen, `USER.DAT` die, die sich auf einzelne Benutzer beziehen (Benutzer-Profile). Dadurch kann man beispielsweise getrennte Backups von alten System-Konfigurationen bzw. Benutzer-Konfigurationen erstellen.

  Außerdem bietet sich so für System-Administratoren die Möglichkeit, `SYSTEM.DAT` auf einem zentralen Rechner unterzubringen. Dadurch sind nur dort Änderungen durchzuführen, die sich auf alle Rechner im Netzwerk auswirken. Ein `USER.DAT` kann im Heimatverzeichnis jedes Benutzers (auch zentral) liegen, sodass er auf allen Rechnern, auf denen er sich einloggt, dasselbe Windows-Erscheinungsbild erhält.

  Wenn Windows erfolgreich gestartet wurde, werden die Registry-Daten nach `SYSTEM.DA0` bzw. `USER.DA0` kopiert. Wenn man Windows bei einer Umkonfigurierung so gründlich abschießt, dass es nicht mehr starten will, kann man diese Dateien über die Registry-Dateien kopieren und die alte (stabile?) Konfiguration wiederherstellen.

- Es gibt API-Funktionen, um Einträge in der Registry auszulesen oder zu schreiben. Das wird hauptsächlich von Windows selbst und den Gerätetreibern verwendet. Es gibt außerdem einen Editor `regedit` (unter NT `regedt32`), mit dem auch der Benutzer solche Manipulationen vornehmen kann.

  Man kann außerdem (mit entsprechenden Administrations-Tools) per Remote-Procedure-Call auf die Registry anderer Rechner in einem Netzwerk zugreifen. Ein System-Administrator braucht sich also nicht an den Rechner zu bemühen.

- Ein Nachteil des Konzepts liegt darin, dass man nun nicht mehr ganz einfach einzelne Dateien austauschen kann. Man sollte sie auch nicht selbständig (z.B. mit einem Binäreditor) verändern, weil sehr leicht die Struktur zerstört wird und das ganze System sofort und vollständig unbrauchbar wird.

  Versierte Administratoren kommen unter UNIX oft schneller zum Ziel, wenn sie ihnen gut bekannte Textdateien zu editieren haben und sich nicht durch eine Registry zu klicken haben.

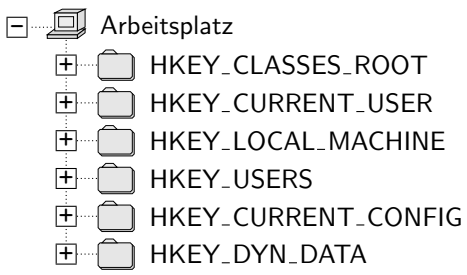
- Während unter Windows NT die meisten Teile der Registry geschützt sind und nur vom Administrator (und von ihm gestarteten Programmen) gelesen und verändert werden können, sind die Daten in Windows 95 vogelfrei. So können Benutzer z.B. Einträge anderer Benutzer lesen.

14.4.5.1 Der Aufbau der Registry

Die Registry ist wie ein Dateisystem hierarchisch gegliedert und wird auch genauso (wie im Explorer) im Editor dargestellt.

- Die **Verzeichnisse** werden dabei als *Schlüssel* (*key*), Unterverzeichnisse als *Teilschlüssel* (*subkey*) bezeichnet. Es gibt „Pfade“ wie im Dateibaum, die auch mit Hilfe des Backslash-Zeichens '\' aufgebaut werden. Die Wurzel des Baums heißt „Arbeitsplatz“ und steht für den gesamten Rechner.
- Die eigentlichen **Wert-Einträge** (*values*) sind die Blätter in diesem Baum. Jeder Eintrag ist ein Paar aus Werte-Name und Werte-Daten. Die NT-Version unterscheidet bei den Daten verschiedene Typen (String, Langwort, String mit enthaltenen Variablen, Binärdaten). Windows 95 kennt nur Binärdaten und Textdaten. Der Typ wird in `regedit` durch unterschiedliche Icons vor dem Namen angezeigt.

Rechts ist die oberste Hierarchie der Registry dargestellt. Wie allgemein in Windows, z.B. im Explorer, üblich, steht eine Mappe für ein Verzeichnis. Die vorstehenden Symbole '+' und '−' dienen dazu, ein Verzeichnis auf- bzw. zuzuklappen (hier sind noch alle geschlossen). Verzeichnisse ohne eines dieser Symbole enthalten keine Unterverzeichnisse.



Diese sechs obersten Ordner haben folgende Bedeutung:

HKEY_CLASSES_ROOT: Hier liegen Informationen für OLE (Object Linking and Embedding) und für die Verknüpfung von Dateitypen mit Programmen (sodass beim Doppelklick auf ein `.GIF` im Explorer ein Grafikprogramm aufgerufen wird, etc.).

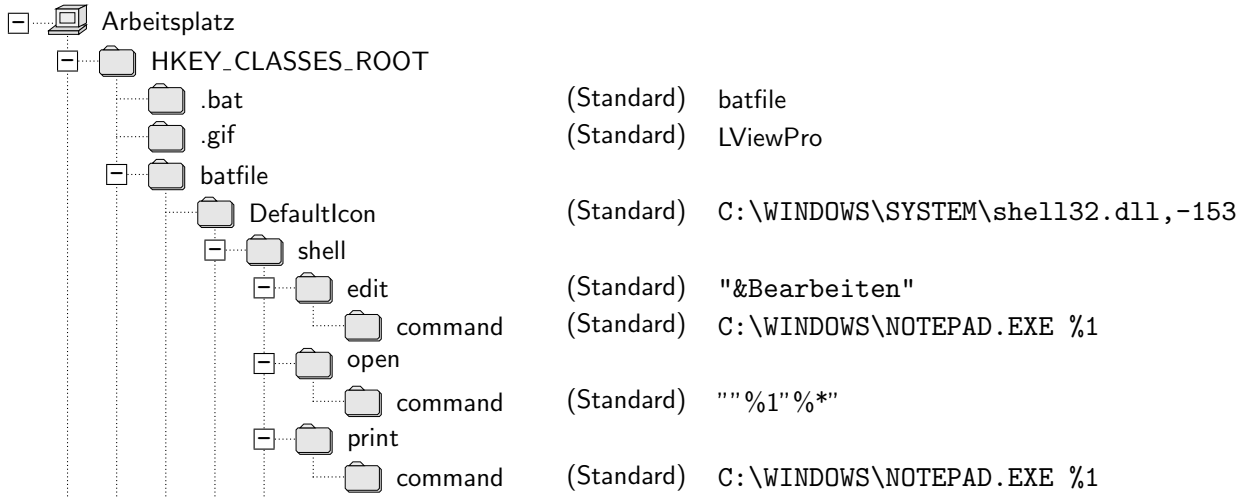
HKEY_CURRENT_USER: Hier ist das „Profil“ des aktuellen Benutzers zu finden, also seine persönlichen Einstellungen. Windows 95 verwaltet zwar keine vernünftigen Zugriffsrechte auf Dateien, immerhin aber können verschiedene Benutzer Desktop-Hintergrund und Desktop-Icons einstellen, Mausgeschwindigkeit, Tastenbelegungen, Farben, etc.

HKEY_LOCAL_MACHINE: In diesem Verzeichnis liegen alle Informationen über den Rechner, die vom jeweiligen Benutzer unabhängig sind. Dazu gehören alle Hardware- und Treiber-Informationen, globale Einstellungen diverser Programme. Außerdem liegen hier die Einstellungen der Benutzer, auf die von CURRENT_USER und USERS aus nur verwiesen wird.

HKEY_USERS: Verweise auf die Benutzerprofile aller Benutzer sind in diesem Verzeichnis abgelegt. Außerdem gibt es ein Profil namens „`.Default`“, das verwendet wird, wenn beim Einloggen kein Name angegeben wird. Es dient auch als Vorlage, wenn sich ein neuer Benutzer anmeldet.

HKEY_DYN_DATA: Hier werden diverse flüchtige Informationen untergebracht – physisch nicht in den Registry-Dateien, sondern im RAM (analog zu `/proc` in UNIX). Es liegen hier Informationen über Hardware, Netzwerk, Systemauslastung, etc.

Wir schauen uns als Beispiel einen Ausschnitt im CLASSES_ROOT-Schlüssel an:

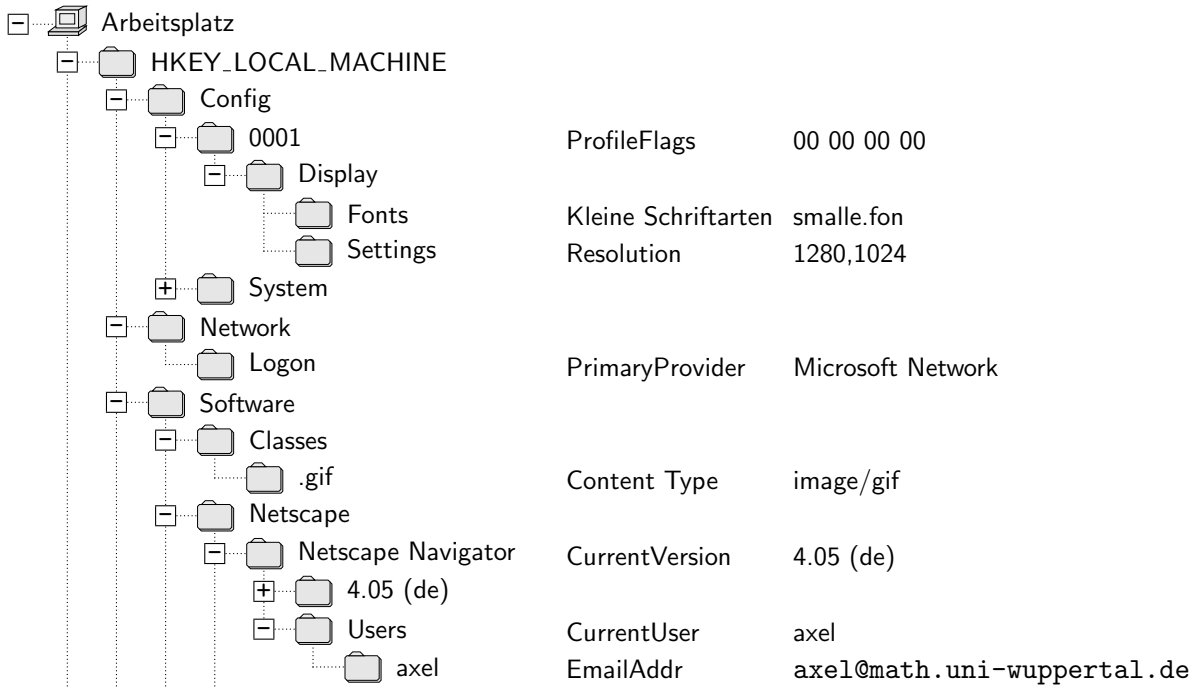


Rechts ist jeweils ein Wert-Eintrag aus dem entsprechenden Verzeichnis links aufgeführt (zufällig haben alle keinen echten Namen, was durch die Angabe „(Standard)“ angedeutet wird).

Die Endung `.bat` verweist auf den Eintrag `batfile`, unter dem Anweisungen zur Darstellung und Behandlung von Dateien mit dieser Endung angegeben sind, z.B.:

- das Icon im Explorer,
- die Aktionen bei Aufruf von `open` und `print` im Kontext-Menü (das bei Klicken mit der rechten Maustaste auf dem Dateinamen erscheint),
- ein zusätzlicher Menü-Eintrag `edit` mit dem deutschen Text „`Bearbeiten`“ (Short-Cut ist die Taste ‘`B`’ durch das vorangestellte ‘`&`’).

Als weiteres Beispiel, das sich nun selbst erklärt, ein Ausschnitt aus dem LOCAL_MACHINE-Schlüssel:



14.4.5.2 Registrierungs-Dateien

Im Registrierungs-Editor kann man mit den Funktionen „Registrierungsdatei importieren/exportieren“ Teile der Registry in Dateien hinausschreiben (die zuvor markierten) bzw. aus Dateien einlesen. Die Dateien haben standardmäßig die Endung `.REG` und sind ähnlich wie die alten `.INI`-Dateien aufgebaut.

Im Kontext-Menü solcher Dateien findet sich der Eintrag „Zusammenführen“, der die Einträge in die Registry übernimmt (ein Doppelklick auf den Namen bewirkt dasselbe). Manche Software- oder Treiber-Updates bestehen zum Teil aus solchen Dateien.

Wenn man im obigen Bild den Teilbaum „Netscape Navigator“ markiert und in eine Datei exportiert, erhält man dort in etwa folgenden Inhalt:

```
REGEDIT4

[HKEY_LOCAL_MACHINE\SOFTWARE\Netscape\Netscape Navigator]
"Current Version"="4.05 (de)"

[HKEY_LOCAL_MACHINE\SOFTWARE\Netscape\Netscape Navigator\4.05 (de)\Main]
"Java Directory"="C:\\Programme\\Netscape\\Communicator\\Program\\Java"
...
```

Man kann `regedit` auch über eine Kommandozeile aufrufen, ohne dass der Fenster-Editor gestartet wird:

`regedit file.reg ...`
: nimmt den Inhalt der Datei(en) in die Registry auf (Zusammenführen)

`regedit /e file.reg [key]`
: exportiert die gesamte Registry (oder nur den Schlüssel `key`) in eine Datei

`regedit /c file.reg`
: liest die komplette Registry neu aus einer Datei (ersetzt den alten Inhalt, Vorsicht!)

Mit den Optionen `/L:`*systempath* und `/R:`*userpath* kann man alternative Pfade zu den Dateien `SYSTEM.DAT` bzw. `USER.DAT` angeben, wenn nicht der Standard-Pfad verwendet werden soll.

X-Window unter UNIX bedient sich übrigens schon länger eines ähnlichen Mechanismus. Die Datenbank heißt dort `xrdb` (X Resource Data Base). Sie nimmt allerdings nur die für die Oberfläche und X-Anwendungen relevanten Informationen auf, keine, die das unterliegende System selbst betreffen.

Das UNIX-Kommando zur Verwaltung der Datenbank heißt ebenfalls `xrdb`. Die Entsprechungen zu den `regedit`-Aufrufen von oben heißen „`xrdb -merge`“, „`xrdb -query`“, „`xrdb -load`“.

14.4.6 Weitere Systemdateien

Windows 95 legt einige weitere Dateien außerhalb seines Verzeichnisses (im Hauptverzeichnis der Windows-Partition) ab, die teilweise dazu geeignet sind, das System nach einem Totalcrash

wieder zum Laufen zu bringen. Einige weitere im Windows-Verzeichnis sind in folgender Tabelle zusätzlich angegeben:

`autoexec.dos`	Kopie von `autoexec.bat` vor der Windows-Installation, wird bei der Bootmenü-Option „vorige MS-DOS-Version“ wieder in `autoexec.bat` umbenannt und ausgeführt (das Windows-`autoexec.bat` wird dazu zwischenzeitlich in `autoexec.w40` umbenannt)
`bootlog.txt`	Protokoll des letzten Systemstarts (falls im Bootmenü „protokolliert“ gewählt wurde)
`bootlog.prv`	Protokoll des vorletzten Systemstarts (previous)
`command.dos`	Kopie von `command.com` vor der Windows-Installation
`config.dos`	Kopie von `config.sys` vor der Windows-Installation
`detlog.txt`	Ergebnis der Hardware-Erkennung (bei der Installation oder über die Systemsteuerung)
`detlog.old`	Ergebnis der vorigen Hardware-Erkennung
`io.sys`	MS-DOS-Teil (7.0) von Windows (kombiniert `msdos.sys` und `io.sys`)
`logo.sys`	Windows-Startbildschirm, im Windows-Verzeichnis (BMP-Format)
`logos.sys`	Bildschirm „Sie können den Computer jetzt ausschalten“, im Windows-Verzeichnis (BMP-Format)
`logow.sys`	Bildschirm „Der Computer wird heruntergefahren“, im Windows-Verzeichnis (BMP-Format)
`msdos.dos`	Kopie von `msdos.sys` vor der Windows-Installation
`oemlog.txt`	Entsprechung zu `setuplog.txt` bei vorinstalliertem Windows
`scandisk.log`	Ergebnis des letzten `scandisk`-Durchlaufs
`setuplog.txt`	installierte Komponenten (nur bei der Windows-Installation, nicht bei Systemsteuerung/Software)
`setuplog.old`	vorige Version von `setuplog.txt` (d.h. vorige Windows-Installation)
`suhdlog.dat`	Kopie der Bootsektoren vor der Windows-Installation (werden bei einer Windows-Deinstallation zurückgeschrieben)
`suhdlog.bak`	Kopie von vorigem `suhdlog.dat`, entsteht bei Neuinstallation von Windows
`system.1st`	Kopie von `system.dat` (Registry) direkt nach der Windows-Erstinstallation
`win386.swp`	permanente Swap-Datei, falls in der Systemsteuerung so eingestellt
`winboot.sys`	Kopie von `io.sys` vor der Windows-Installation
`w95undo.dat`	alle Dateien, die bei einer Installation von Windows 95 über ein Windows 3.1 überschrieben wurden
`w95undo.ini`	Kontrolldatei zu `w95undo.dat`

Literatur

Betriebssysteme allgemein:

Abraham Silberschatz, James L. Peterson, Peter B. Galvin, *Operating System Concepts*, Addison-Wesley 1991

Andrew S. Tanenbaum, *Modern Operating Systems*, Prentice-Hall 1994

Andrew S. Tanenbaum, *Distributed Operating Systems*, Prentice-Hall 1995

Jürgen Nehmer, Peter Sturm, *Systemsoftware*, dpunkt.verlag 1998

C. Ritchie, *Operating Systems, incorporating UNIX and MS-DOS*, DP Publications 1992

Mamoru Maekawa, Arthur E. Oldehoeft, Rodney R. Oldehoeft, *Operating Systems – Advanced Concepts*, Benjamin/Cummings Publishing 1987

E.W. Dijkstra, *Solution of a problem in concurrent programming control*, Communications of the ACM 8, 9, 1965, p 569

E.W. Dijkstra, *Cooperation of sequential processes*, in *Programming Languages*, F. Geunys (Ed.), Academic Press N.Y. 1968, pp 43-112

UNIX:

S.R. Bourne, *The UNIX System*, Addison-Wesley 1983

Brian W. Kernighan, Rob Pike, *The UNIX Programming Environment*, Prentice-Hall 1984

Maurice J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall 1986

Keith Haviland, Ben Salama, *UNIX System Programming*, Addison-Wesley 1987

Eric Foxley, *UNIX for Super-Users*, Addison-Wesley 1985

Aeleen Frisch, *Essential System Administration*, O'Reilly 1991

Simson Garfinkel, Gene Spafford, *Practical UNIX Security*, O'Reilly 1991

John Bloomer, *Power Programming with RPC*, O'Reilly 1992

Janet I. Egan, Thomas J. Teixera, *Writing a UNIX Device Driver*, John Wiley & Sons 1989

Intel-Prozessoren:

Intel 80386 Systemprogrammierung, Markt & Technik 1989

William H. Murray, Chris H. Pappas, *80386/80286 Assemblerprogrammierung für Experten*, McGraw-Hill 1987

Penn Brumm, Don Brumm, *80386 Handbuch für Programmierer und Systementwickler*, Markt & Technik 1987

DOS:

Michael Tischer, *PC Intern – Systemprogrammierung*, Data Becker 1987

Robert S. Lai, *MS-DOS-Device-Treiber*, Addison-Wesley 1989

Ray Duncan, *Extending DOS*, Addison-Wesley 1991

Windows:

Charles Petzold, *Windows System Programming*, Microsoft Press 1990

Charles Petzold, *Programming Microsoft Windows 95*, Microsoft Press 1995

Jeffrey Richter, *Windows Programmierung für Experten*, Microsoft Press 1997

Stan Mitchell, *Inside the Windows 95 File System*, O'Reilly 1997

Paul Robichaux, *Managing the NT Registry*, O'Reilly 1998

Axel Rogat
Fachbereich Mathematik
Universität Wuppertal
Gaußstraße 20
42097 Wuppertal

G 15.23
(0202) 439-3553
axel@math.uni-wuppertal.de
http://www.math.uni-wuppertal.de/~axel

A UNIX-Kommandos

In der folgenden Tabelle sind die wichtigsten UNIX-Kommandos (teilweise in die Shells eingebaut), sowie typische Utilities stichwortartig aufgeführt (Seitennummern in eckigen Klammern).

`a2ps`	ASCII to PostScript, Konvertierungsprogramm
`ar`	Archiv-Verwalter
`as`	Assembler
`at`	Kommandos zu einer bestimmten Uhrzeit starten (POSIX)
`awk`	Sprache für String-/Muster-Verarbeitung
`bash`	Bourne-Again-Shell [96]
`batch`	Kommando ausführen, wenn das System wenig ausgelastet ist
`bg`	Prozess in den Hintergrund schieben [101]
`bison`	GNU-Pendant zum Parser-Generator `yacc`
`cal`	Kalender anzeigen (BSD)
`captoinfo`	Erzeugen einer `terminfo`-Datei aus einem `termcap`-Eintrag
`case`	Mehrfach-Fallunterscheidung in Shell-Skripten [115]
`cat`	Dateien anzeigen/verketten [60]
`cc`	systemspezifischer `C`-Compiler
`cd`	change directory, aktuelles Verzeichnis wechseln [70]
`chfn`	change full name (Angabe in `passwd`, wird z.B. bei '`finger`' ausgegeben)
`chgrp`	Gruppenbesitzer einer Datei ändern [74]
`chmod`	Zugriffsflags einer Datei ändern [73]
`chown`	Besitzer einer Datei ändern [74]
`chroot`	Kommando mit alternativem Root-Directory starten
`chsh`	Login-Shell ändern (Angabe in `passwd`)
`clear`	Bildschirm löschen (unter Benutzung von `terminfo`-Angaben)
`compress`	Datei komprimieren (Endung `.Z`)
`cp`	Datei(en) kopieren [60]
`crontab`	Crontab-Datei verwalten
`csh`	C-Shell [96]
`date`	Systemdatum und -zeit anzeigen oder setzen
`df`	disk free, Anzeige freien Plattenplatzes
`diff`	Vergleich zweier Datei-Inhalte
`do`	Blockbeginn in einem Shell-Skript [116]
`done`	Blockende in einem Shell-Skript [116]
`du`	disk usage, Anzeige verbrauchten Plattenplatzes
`echo`	Text anzeigen [98]
`ed`	Zeilen-Editor [60]
`egrep`	Datei(en) nach Muster durchsuchen, e=extended
`elm`	electronic mail, einfaches Mailer-System
`else`	Alternative in Shell-Skripten [112]
`env`	Environment ausgeben/Kommando mit Alternativ-Env. starten [82]
`esac`	Ende von `case` in Shell-Skripten [115]
`exec`	Überladen des aktuellen Shell-Prozesses [101]
`exit`	Beendigung des Shell-Prozesses [98]
`export`	Shell-Variable exportieren [109]
`expr`	mathematischen Ausdruck auswerten

`fg` Prozess in den Vordergrund holen [101]
`fi` Ende einer Fallunterscheidung in Shell-Skripten [112]
`fgrep` Datei(en) nach Muster durchsuchen, `f`=fixed (keine regul. Ausdrücke)
`file` Dateityp bestimmen
`find` Datei(en) suchen
`finger` Benutzer-Informationen ausgeben [97]
`flex` Alternativ-Version von lex
`for` Schleifenbeginn in Shell-Skripten [116]
`free` Speicher-Informationen ausgeben
`fsck` file system check, Filesystem-Test-/Reparaturprogramm [372]
`ftp` file transfer program, Dateiübertragung (`ftp`-Protokoll)
`fuser` zu einer Datei die IDs der sie benutzenden Prozesse ausgeben
`gcc` GNU-`C`-Compiler (`g++` für `C++`)
`getopts` Parameter-Auswertung in Shell-Skripten [120]
`gnuplot` GNU-Plot-Programm
`grep` Datei(en) nach Muster durchsuchen [104]
`groff` Textformatierungs-System, Layer um `troff` herum
`groups` Gruppen zu gegebenem User ausgeben
`gzip` GNU-Version von ZIP
`hostname` Hostname ausgeben oder setzen
`id` User- und Gruppen-IDs zu gegebenem User ausgeben
`if` Fallunterscheidung in einem Shell-Skript [112]
`infocmp` Vergleich zweier Terminal-Beschreibungen refpageinfocmp
`ipcrm` ipc rm, Löschen eines IPC-Objekts [218]
`ipcs` inter-process communication status, IPC-Status-Ausgabe [217]
`ispell` interactive spell checker
`kill` Signal an einen Prozess schicken (per PID-Angabe) [132]
`killall` Signal an einen Prozess schicken (per Namens-Angabe)
`last` Zeit der letzten Logins ausgeben (pro User/Terminal) [349]
`ld` Linker (statisch, der Runtime-Linker/Loader heißt meist `ld.so`)
`ldconfig` Konfigurierung des Run-Time-Linkers `ld.so`
`ldd` library dependency display (von einem File benötigte Shared Libraries)
`less` erweiterte Version von `more` [60]
`lex` Standard-Scanner-Generator (`lex`=lexikalische Analyse)
`ln` Datei-Link anlegen (Hardlink oder symbolisch) [78]
`login` Systemsitzung starten
`logname` Login-Namen ausgeben
`lpq` line printer query, Drucker-Status erfragen
`lpr` Druckauftrag starten (`r`=remote)
`lprm` Druckauftrag aus der Warteschlange entfernen
`ls` Directory (-ies) auflisten 3.2.2
`mail` e-mail abfragen / schicken
`make` komplexe Übersetzungsvorgänge verwalten
`man` Online-Manual-Seiten abfragen [53]
`mkdir` Directory anlegen [70]
`mkfifo` FIFO anlegen [212]
`mkfs` File-System anlegen [362]
`mknod` Special-File anlegen [324]

`mktemp` Temporärdatei anlegen
`more` Textdatei seitenweise anzeigen [60]
`mount` Dateisystem in den Dateibaum einhängen [362]
`mv` Datei bewegen (umbenennen, verschieben) [60]
`newgrp` Gruppen-Identität ändern [347]
`nice` Programm mit speziellem Nice-Wert starten [170]
`nohup` Programm starten und vor Hangups schützen
`od` octal dump, Dateiausgabe im Oktal-/Hexadezimal-Format, etc.
`onintr` Signal-Behandlung in der `csh` [158]
`passwd` Passwort ändern [55]
`pr` Datei zum Ausdruck formatieren [103]
`ps` Prozess-Status (-Tabelle) anzeigen [128]
`psnup` in PostScript-Dateien mehrere Seiten auf einer vereinigen
`pwd` print working directory, aktuellen Pfad ausgeben [70]
`renice` Nice-Wert von laufenden Prozessen ändern
`rlogin` remote login, startet Systemsitzung auf einem anderen Rechner
`rm` Datei löschen [60]
`rmdir` (leeres) Directory löschen [70]
`rsh` remote shell, startet Kommando auf einem anderen Rechner [248]
`runlevel` vorherigen und aktuellen Runlevel ausgeben [352]
`rwall` remote write all
`rwho` `who` auf allen Maschinen im lokalen Netzwerk
`sed` stream editor, Musterverarbeitung über reguläre Ausdrücke [105]
`set` positionale Parameter setzen [110]
`setenv` Shell-Variable setzen/exportieren bei der `csh` [109]
`sh` Bourne-Shell [96]
`shar` shell archive anlegen
`shutdown` System herunterfahren
`sleep` Prozess einige Sekunden schlafenlegen
`sort` Datei zeilenweise sortieren [104]
`split` Datei in Stücke zerlegen
`strace` system trace, Systemaufrufe und Signale eines Programms anzeigen
`stty` set tty, Terminal-Einstellungen anzeigen/ändern [335]
`su` substitute user, Benutzer-Identität wechseln [57]
`sync` Dateisystem-Puffer leeren [357]
`tail` die letzten Zeilen einer Textdatei anzeigen
`talk` Kommunikation mit anderem Benutzer
`tcsh` erweiterte C-Shell [96]
`tee` Filter, der inmitten einer Pipeline Ausgaben in eine Datei schreibt [103]
`telnet` Kommunikation mit anderem Rechner per `telnet`-Protokoll
`test` Auswertung von Bedingungen in Shell-Skripten (Filetypen, etc.) [114]
`then` bei Fallunterscheidungen in Shell-Skripten [112]
`tic` terminfo compiler [337]
`toe` Auflisten verfügbarer Terminal-Protokolle (table-of-entries) [336]
`touch` Zeitangaben einer Datei verändern
`tput` Terminal-Abfrage und -Steuerung [338]
`tr` Zeichencode-Umsetzung [104]
`trap` Signal-Behandlung in der Shell [158]

`troff`	Text-Formatierungsprogramm
`tty`	Name des aktuellen Terminals ausgeben [328]
`umask`	ufcm-Maske anzeigen/setzen [74]
`uname`	System-Information ausgeben (Hardware, Betriebssystem, etc.)
`uncompress`	Gegenstück zu `compress` (`.Z`)
`unshar`	shell archive auspacken
`until`	Schleifenbeginn in Shell-Skripten [116]
`unzip`	ZIP-Datei auspacken
`uucp`	unix to unix copy, Dateien zwischen UNIX-Systemen kopieren
`uudecode`	mit `uuencode` codiertes File entschlüsseln
`uuencode`	(Binär-)Datei in ASCII-Datei (7 Bit) verschlüsseln
`vi`	Standard-Texteditor [60]
`wait`	auf Beendigung eines Prozesses warten [126]
`wall`	Nachricht an alle Benutzer schreiben
`wc`	word count, Worte (Zeilen, etc.) einer Textdatei zählen
`whereis`	ausführbare Datei, Quellcode und Manpage zu einem Programm finden
`which`	absoluten Pfad eines Kommandos ausgeben
`while`	Schleifenbeginn in einem Shell-Skript [116]
`who`	eingeloggte Benutzer anzeigen
`whoami`	eigene User-ID ausgeben
`write`	Nachricht an anderen Benutzer schreiben
`yacc`	Standard-Parser-Generator (yet another compiler compiler)

B UNIX-Systemaufrufe und -Bibliotheksfunktionen

Hier sind die behandelten UNIX-Systemaufrufe und zusätzlich einige wichtige Bibliotheksfunktionen aufgelistet (Seitennummern der Beschreibung in eckigen Klammern).

`accept`	Akzeptieren einer Verbindung bei einem Server-Socket [242]
`access`	Abfrage der Zugriffsrechte [74]
`alarm`	Anmelden eines Timer-Requests [152]
`alphasort`	Vergleich zweier `dirent`-Strukturen [372]
`atexit`	Einhängen einer `exit`-Handlerroutine [137]
`bind`	Binden eines Sockets an eine Adresse [242]
`chdir`	Wechseln des aktuellen Verzeichnisses [71]
`chmod`	Ändern der Datei-Zugriffsrechte [74]
`chown`	Ändern des Datei-Besitzers [74]
`close`	Schließen einer Low-Level-Datei [63]
`closedir`	Beenden des Auslesens eines Verzeichnisses [72]
`closelog`	Schließen einer `syslogd`-Verbindung [351]
`connect`	Verbindungsaufbau bei einem Client-Socket [242]
`creat`	Anlegen einer Low-Level-Datei [63]
`dup`	Kopieren eines File-Deskriptors [159]
`dup2`	Kopieren eines File-Deskriptors mit bestimmtem Ziel [159]
`execl`	Ausführen eines Programms (Listen-Argumente) [146]
`execle`	Ausführen eines Programms (Listen-Argumente, Environment-Übergabe) [146]
`execlp`	Ausführen eines Programms (Listen-Argumente, Pfadsuche) [146]
`execv`	Ausführen eines Programms (Vektor-Argumente) [146]
`execve`	Ausführen eines Programms (Vektor-Argumente, Environment-Übergabe) [146]
`execvp`	Ausführen eines Programms (Vektor-Argumente, Pfadsuche) [146]
`fchdir`	Wechseln des aktuellen Verzeichnisses per File-Deskriptor [71]
`fchmod`	Ändern der Datei-Zugriffsrechte per Filedeskriptor [74]
`fchown`	Ändern des Datei-Besitzers per Filedeskriptor [74]
`fcntl`	Datei-Kontroll-Operation [235]
`fdopen`	`C`-Stream aus Filedeskriptor [66]
`fileno`	Filedeskriptor aus `C`-Stream [66]
`fork`	Erzeugen eines neuen Prozesses durch Verzweigung [137]
`fstat`	Einholen von Datei-Informationen per File-Deskriptor [79]
`fsync`	Dateipuffer hinausschreiben [67]
`ftok`	Konversion Pfadname → IPC-Schlüssel [217]
`getcwd`	Abfrage des Pfads des aktuellen Verzeichnisses [71]
`getegid`	Abfrage der effektiven Group-ID [76]
`getenv`	Lesen einer Umgebungsvariable [83]
`geteuid`	Abfrage der effektiven User-ID [76]
`getgid`	Abfrage der Group-ID [58]
`getgrgid`	Einholen von Gruppen-Informationen per GID [348]
`getgrnam`	Einholen von Gruppen-Informationen per Name [348]
`getgroups`	Gruppenzugehörigkeiten abfragen [348]
`gethostbyname`	Abfragen von Rechner-Informationen per Name [242]
`getitimer`	Abfrage eines Intervalltimer-Status [156]

`shmat`	Ankoppeln eines Shared-Memory-Segments [228]
`shmctl`	Kontroll-Operationen auf einem Shared-Memory-Segment [228]
`shmdt`	Abkoppeln eines Shared-Memory-Segments [228]
`shmget`	Anlegen/Anfragen eines Shared-Memory-Segments [228]
`sigaction`	erweiterte Signal-Behandlung [151]
`sigaddset`	Aufnehmen eines Signals in eine Signal-Menge [151]
`sigdelset`	Löschen eines Signals aus einer Signal-Menge [151]
`sigemptyset`	Leeren einer Signal-Menge [151]
`sigfillset`	Füllen einer Signal-Menge [151]
`sigismember`	Test auf Enthaltensein in einer Signal-Menge [151]
`signal`	Einhängen einer Signal-Handlerroutine [150]
`sigprocmask`	Definition von Signal-Blockierungen in Handler-Routinen [151]
`sigpending`	Abfrage noch nicht behandelter Signale [151]
`sigsuspend`	Prozess-Blockade bis zu bestimmten Signalen [151]
`sleep`	Prozess-Blockade für einen gewissen Zeitraum [155]
`socket`	Anlegen eines Sockets [238]
`stat`	Einholen von Datei-Informationen [79]
`symlink`	Anlegen eines symbolischen Links [78]
`syslog`	Senden einer Log-Message an `syslogd` [351]
`system`	Ausführen einer Shell-Befehlszeile [106]
`tcgetattr`	Terminal-Control: Attribute erfragen [332]
`tcsetattr`	Terminal-Control: Attribute setzen [332]
`ttyname`	liefert Namen der Geräte-Datei zu einem Terminal-Filedeskriptor [330]
`unlink`	Löschen eines Directory-Eintrags [79]
`unsetenv`	Löschen einer Umgebungsvariable [83]
`wait`	Warten auf das Ende eines Kind-Prozesses [142]
`waitpid`	Warten auf das Ende eines bestimmten Kind-Prozesses [142]
`write`	Schreiben in eine Low-Level-Datei [64]

Index