Fachhochschule München
Fachbereich Informatik

Sommersemester 1999

DV-Systeme 2

Dozent: Dr. Thomas Tensi

Inhaltsverzeichnis

1 Einführung

1.1 Allgemeines

1.1.1 Sinn eines Betriebssystems

Ein DV-System gliedert sich in

- Hardware,
- Betriebssystem und
- Anwendungsprogramme.

Die Hardware stellt die elementaren Dienste zur Verfügung (Rechenleistung, Speicherung, Ein-/Ausgabemöglichkeiten). Die Anwendungsprogramme benötigen diese Ressourcen und manchmal mehr, als auf einmal zur Verfügung steht.

Das Betriebssystem vermittelt daher diese Ressourcen möglichst effizient an die Anwender/Anwendungsprogramme. Es abstrahiert dabei von speziellen Eigenarten der unterliegenden Hardware.

1.1.1.1 Anfänge

Anfangs wurden Programme händisch in den Computer instruktionsweise eingegeben und nach der Eingabe gestartet, d.h. der Programmierer war auch der Operator. Später gab es Eingabe über Bandlesegeräte, aber dies erfolgte meist in mehreren separaten Schritten (Compiler, Assembler).

Charakteristisch für dieses Vorgehen waren jedoch hohe Rüstzeiten für einen Rechenlauf und eine ineffiziente Ausnutzung des Systems.

1.1.1.2 Stapelbetrieb

Um die Rüstzeit zu reduzieren, wurde die Rolle des Operators eingeführt. Dieser bekam Programmaufträge angeliefert und hat sie nach Art gebündelt (z.B. alle COBOL-Programme). Ergebnisse wurden wiederum an den Programmierer geliefert (z.B. über Ausdrucke).

Problem war zu erkennen, wann ein Programm fertig war. Lösung dazu war, ein kleines Programm (den *Monitor*) im Speicher zu halten, der am Ende eines Programms angesprungen wurde. Außerdem konnte er Kommandos in der Eingabe verarbeiten. Jobs wurden als Lochkartenstapel mit eingeschobenen Kommandokarten angeliefert, die der Monitor interpretiert hat.

Daher hatte ein Monitor drei Aufgaben: ein Programm laden, Kommandos interpretieren und Folgejobs zu starten.

1.1.1.3 Spooling (simultaneous peripheral operation on-line)

Unpraktisch war, daß die Geschwindigkeit von E/A-Geräten und CPU stark differierten, d.h. die Auslastung der CPU war bei Ein- und Ausgabe sehr gering. Lösung war, die Information auf schnelle Zwischenspeicher abzulegen (auf Plattenspeicher über Satellitenrechner). Zugriffe auf

Karten oder Drucker wurden auf Plattenzugriffe abgebildet. Man konnte sogar zeitlich verschachtelt Daten eines Auftrags auf Platte schreiben, während Daten eine vorhergehenden Jobs von der CPU gelesen wurden.

1.1.1.4 Mehrprogrammbetrieb

Aufgrund des Spoolingansatzes waren im Hintergrundspeicher mehrere Aufträge verfügbar. Um die CPU-Ausnutzung zu optimieren, wurde ein Auftrag aus der Menge genommen, der gerade nicht auf irgendetwas wartete (Eingabe, Gerät o.ä.) und dieser solange ausgeführt, bis dieser selbst warten muß usw.

Um die Zeit für die Prozeßwechsel gering zu halten, mußten die verschiedenen Aufträge im Hauptspeicher gehalten werden. Daher mußte der Programmkode für Mehrprogrammbetrieb verschiebbar sein oder ein leistungsfähiges Speichermanagement zur Verfügung stehen.

1.1.1.5 Time-Sharing

Logische Fortsetzung des Mehrprogrammbetriebs ist das Time-Sharing. Die Prozeßwechsel erfolgen zeitgesteuert in sehr kurzer Folge. Dadurch entsteht der Eindruck, daß alle Prozesse gleichzeitig ablaufen.

Unterschiede zum Stapelbetrieb sind:

- Ablauf der Schritte eines Auftrags ist beim Stapelbetrieb festgelegt; bei Time-Sharing bestimmt dies der Benutzer interaktiv.
- Kommunikationsmedium ist bei Time-Sharing das Terminal, bei Stapelbetrieb sind es Ein- und Ausgabegeräte oder -dateien.
- Antwortzeiten sollten für ein Time-Sharing-System nicht sehr hoch sein.
- Um Time-Sharing zu unterstützen braucht man ein *Online-Dateisystem*, bei Stapelbetrieb nicht notwendigerweise.

Heutzutage sind viele Betriebssysteme auf Time-Sharing-Betrieb ausgelegt (z.B. UNIX). Im Großrechnerbereich gibt es auch Systeme, die für Stapelbetrieb optimiert sind (MVS).

Für Time-Sharing braucht man verschiedene Aspekte im Betriebssystem:

- Mechanismen für Rechnerkernzuteilung (Scheduling)
- Speichermanagement (virtueller Speicher),
- Hintergrundspeicherverwaltung,
- Online-Dateisystem und
- Zugriffsschutz (aufgrund mehrerer Anwender).

1.1.1.6 Verteilte Systeme

Um beispielsweise die Leistung eines DV-Systems zu steigern, kann man mehrere Prozessoren koppeln. In enggekoppelten Systemen gibt es in einem Rechner mehrere Prozessoren mit gemeinsamen Speicher, in lose-gekoppelten Systemen viele Rechner mit jeweils eigenem Speicher gekoppelt über Kommunikationsleitungen.

Vorteile von verteilten Systemen:

- Mehrfachnutzung von Ressourcen, Informationsaustausch (Idee eines Netzwerks)
- Leistungsverbesserung (Lastverteilung) und
- Erhöhung der Zuverlässigkeit (Redundanz).

1.1.1.7 Echtzeitsysteme

Eingebettete Computersystemen (z.B. in Automobilen oder zur Maschinensteuerung) nehmen Daten auf, verarbeiten sie und steuern Aktoren.

Im Kontext von diesen Systemen gibt es oft strenge Anforderungen an die Antwortzeit auf ein Ereignis. Dies kann ein Time-Sharing-System nicht unbedingt erfüllen.

Echtzeitsysteme sind meist einfach strukturiert, um diese Antwortzeiten zu gewährleisten: kein Hintergrundspeicher (bestenfalls EEPROM), keine komplexe Speicherverwaltung usw.

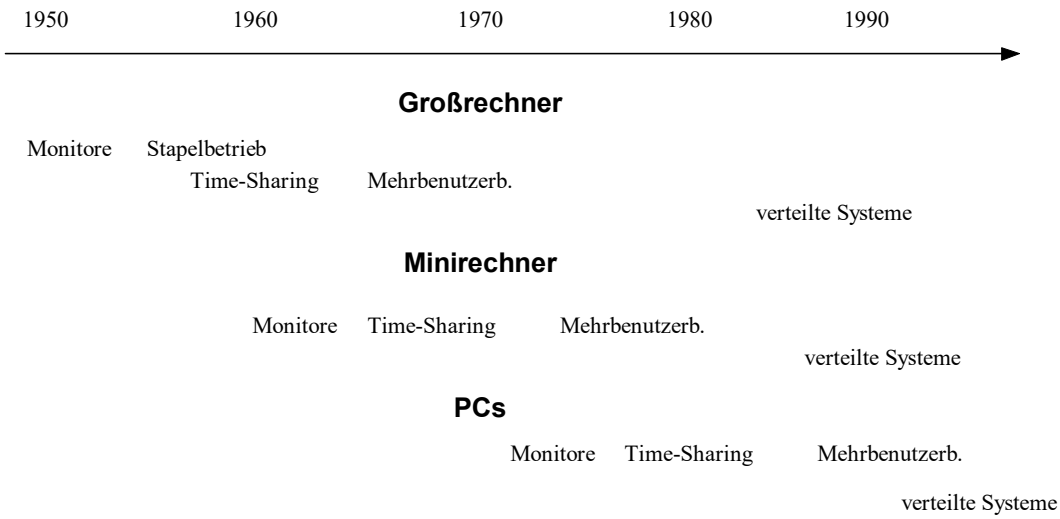


Bild 1-1: Zeitskala für Nutzung verschiedener Betriebssystemkategorien

1.2 Hardwarestrukturen als Grundlage von Betriebssystemen

Die Eigenschaften eines Betriebssystems hängen stark von der unterliegenden Hardware ab. Beispielsweise ist es nützlich, wenn die Hardware verhindert, daß ein amoklaufendes Programm das Betriebssystem zum Absturz bringen kann.

In diesem Abschnitt werden also charakteristische Hardwarestrukturen betrachtet, die bestimmte Betriebssystemeigenschaften gewährleisten.

1.2.1 Unterbrechungssteuerung

(Siehe auch voriges Semester...)

Ein externes Gerät kann entweder durch Polling oder über Unterbrechungen abgefragt und damit verwaltet werden. Vorteil der Steuerung über Unterbrechungen ist eine effizientere Ausnutzung der Ressource „CPU“ und eine einfache Programmstruktur der beteiligten Programme.

Geräte werden wie folgt angesteuert: Die CPU schreibt Daten für den entsprechenden Auftrag an das Gerät in Register auf der Controllerkarte. Das Gerät führt den Auftrag aus und meldet sich über einen Interrupt bei der CPU, sobald der Auftrag abgeschlossen ist. In der entsprechenden Interruptroutine liest die CPU dann Daten aus internen Registern des Geräts aus und startet gegebenenfalls einen neuen Auftrag.

Wenn beispielsweise Zeichen an der seriellen Schnittstelle gelesen werden (von einem Modem), erzeugt der Schnittstellencontroller jedesmal einen Interrupt, wenn ein Zeichen anliegt. Die CPU liest das Zeichen und legt es in einem Puffer im Hauptspeicher ab, damit es von einem Programm bearbeitet werden kann. Bei einem schnellen Modem (50kBit/s) liegt alle 200µs ein Zeichen an. Dieses wird sicherlich in <2µs verarbeitet (circa 20 Maschinenzyklen), sodaß 198µs für andere Verarbeitung zur Verfügung steht. Langsame Geräte sind also sehr gut so zu behandeln.

Dies klappt aber nicht bei schnellen Geräten (Festplatten). Hier muß man den Gerätecontrollern Zugriff auf den Hauptspeicher gewähren (Direktspeicherzugriff, direct memory access, DMA). Dieser Zugriff wird über einen speziellen Chip verwaltet, den DMA-Controller. Er übernimmt Adressierung des Speichers (inkl. Adreßfortschaltung) und Setzen/Auslesen der Daten des Gerätecontrollers.

Bei einem Auftrag reserviert die CPU einen Speicherbereich und teilt diesen dem DMA-Controller mit. Außerdem wird festgelegt, wie dieser Speicher zu füllen ist und welche Adresse das Datenregister des Gerätecontrollers hat. Nach Abschluß des Transfers wird entweder ein Statusbit im DMA-Controller gesetzt, oder er löst einen Interrupt aus.

Beispiel (Einlesen von 4096 Byte von Festplatte): Ein Register des DMA-Controllers enthält die Startadresse, ein weiteres die Transferlänge (4096), ein drittes die Adresse des Ports des Festplattencontrollers und ein Statusregister merkt die Richtung und ob ein Interrupt auszulösen ist. Der Plattencontroller wird ebenfalls aktiviert und liefert seine Anforderungen an den DMA-Controller, der jedes eingelesene Byte übernimmt und ablegt. Bei Abschluß des Transfers wird die CPU informiert.

DMA-Controller können meist mehrere Geräte verwalten (DMA-Kanäle), typischerweise 4 oder 8. Für jedes Gerät ist dann ein eigener Registersatz vorhanden.

1.2.2 Zweimodusbetrieb

Anfangs waren Computer Einbenutzersysteme mit vollständiger Kontrolle seitens des Programmierers.

Wenn mehrere Programme gleichzeitig betrieben werden, ist es aber nicht sinnvoll, das Programme freien Zugriff auf alle Systemressourcen haben. Ein Programm kann dann den Ablauf anderer Programme beeinträchtigen.

Einige Probleme können unmittelbar durch die Hardware erkannt werden (beispielsweise eine ungültige Instruktion). Es wird eine Ausnahme ausgelöst und das fehlerhafte Programm beendet; möglicherweise wird auch ein Speicherabzug erzeugt.

Um einen hohen Schutz zu gewährleisten, müssen aber praktisch alle Fehler erkannt werden. Erster Schritt ist, daß Benutzerprogramme keine kritischen Operationen (z.B. eine Beeinflussung der Interruptbearbeitung) durchführen dürfen. Dies ist nur möglich für Systemprogramme. Dafür wird im Prozessor ein Statusbit eingeführt, daß sagt, ob der Prozessor im *Benutzermodus* oder *Systemmodus* läuft.

Bei Start eines Benutzerprogramms ist der Prozessor im Benutzermodus, bei einer Ausnahme wechselt der Prozessor vom Benutzer- in den Systemmodus. Potentiell gefährliche Instruktionen (wie z.B. Setzen der Unterbrechungssperre) sind nur im Systemmodus möglich: es sind *privilegierte Befehle*.

Auch Ein-/Ausgabe muß auch als privilegiert angesehen werden, weil sie unmittelbar mit der korrekten Initialisierung und Bearbeitung von Unterbrechungen zusammenhängt. Ein Benutzerprogramm kann Ein-/Ausgabe also nur an das Betriebssystem über eine Ausnahme deligieren. Diese spezielle Ausnahme (über einen spezifischen Maschinenbefehl) wird als Systemaufruf interpretiert (es könnte über einen ungültigen Befehlskode erfolgen).

1.2.3 Hardwareschutzmechanismen

Mehrere Probleme sind trotzdem noch nicht gelöst: Ein Programm kann

1. durch Schreiben in fremde Speicherbereiche andere Programme oder sogar das Betriebssystem zerstören,

2. erreichen, daß es selbst im Systemmodus ausgeführt wird (durch Änderung der Interrupttabelle), und

3. die CPU blockieren (durch eine langdauernde Berechnung).

1.2.3.1 Speicherschutz

Punkte 1. und 2. werden gelöst durch Speicherschutzmechanismen in der Hardware.

Simpelste Lösung ist folgende: Jedes Programm hat eine minimale und maximale physische Speicheradresse, die es ansprechen darf. Diese Adressen werden in zwei internen Registern der CPU abgelegt, die nur über privilegierte Instruktionen geändert werden können. In jedem Befehlszyklus im Benutzermodus wird geprüft, ob auf eine Adresse außerhalb dieses Bereichs zugegriffen wird. Wenn ja, wird eine Ausnahme erzeugt und das Programm abgebrochen.

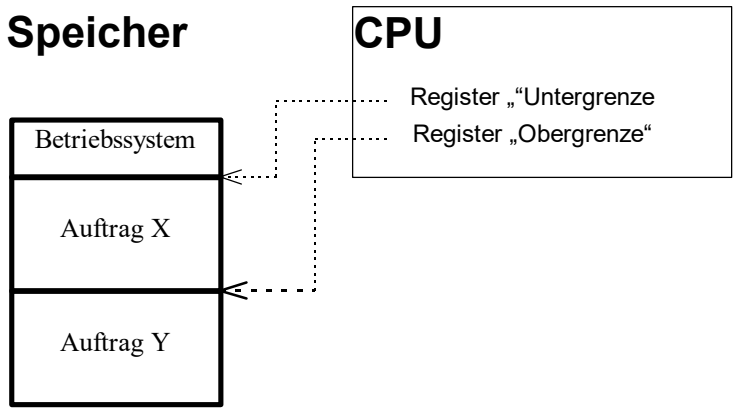


Bild 1-2: Einfacher Speicherschutzmechanismus

Im Systemmodus gilt diese Prüfung nicht, damit das Betriebssystem flexibel auf Speicherbereiche der Benutzerprogramme zugreifen kann.

Es gibt auch komplexere und flexiblere Lösungen, die beim Speichermanagement angesprochen werden.

1.2.3.2 Zeitscheiben

Um zu garantieren, daß das Betriebssystem immer wieder aktiv wird, könnte man verlangen, daß alle Programme relativ schnell nach jeder Zuteilung des Prozessors die Kontrolle wieder an das Betriebssystem zurückgeben (ihre Aufgaben in kleinen Schritten erledigen). Dies heißt *kooperativer Mehrprogrammbetrieb*. Er war beispielsweise in den ersten Version von MS Windows üblich.

Vorteil ist: Man muß keine Zusatzmaßnahmen treffen, damit alle Prozesse und das Betriebssystem irgendwann die Kontrolle über den Prozessor erhalten. Nachteil ist aber, daß die Programme entsprechend strukturiert sein müssen. Sie warten in einer Schleife auf die nächste Zuteilung des Prozessors und führen Minischritte aus.

Beispielprogramm:

```
SOLANGE nochNichtFertig
  <<führe möglichst schnell einen Teil der Aufgabe aus>>
  <<gib Kontrolle an Betriebssystem zurück>>
ENDE-SOLANGE
```

Die Zeit für den Schritt muß möglichst kurz sein (im Millisekundenbereich). Daraus ergeben sich andere Nachteile des kooperativen Multitasking: Einige Aufgaben lassen sich zeitlich schlecht schätzen und auch nur schwer unterteilen (beispielsweise ein rekursiver Algorithmus). Nicht-kooperierende Programme umgehen komplett die Verteilung.

Für das Problem gibt es unterschiedliche Lösungen auf Basis eines programmierbaren Zählers. Dieser löst einen Interrupt aus (*Timerinterrupt*), wenn er auf 0 heruntergezählt hat:

- Ein Programm wird nach einem bestimmten Zeitintervall abgebrochen. Der Zähler wurde vorher prozeßspezifisch mit einem Zeitwert geladen. Bei der Unterbrechung wird der laufende Prozeß beendet.

- Ein Programm wird nach einem bestimmten Zeitintervall unterbrochen. Bei Ablauf der Zeit wird der Prozessor an einen anderen Prozeß vergeben und der aktuelle Prozeß in eine Warteschlange eingereiht.

Schreibzugriffe auf den Zähler sind natürlich privilegiert.

2 Übersicht über UNIX/LINUX

2.1 Geschichte

Ausgangspunkt von UNIX war die Entwicklung des ambitionierten Betriebssystems MULTICS am MIT und bei Bell Labs (Mitte der 60er Jahre). Dieses System wurde nie fertiggestellt. Einige Entwickler bei Bell (Ken Thompson, Dennis Ritchie und Brian Kernighan) übernahmen einige Ideen von MULTICS (z.B. bezüglich Struktur des Dateisystems) und programmierten sie auf einem unbenutzten Minicomputer in Assembler (1969).

Als das System wirklich genutzt werden sollte und auf eine größere Maschine portiert werden mußte, entwickelte Ritchie die maschinennahe höhere Programmiersprache „C“ (basierend auf BCPL). Fast alle Teile des Betriebssystems wurden in dieser Sprache neu programmiert (bis auf einige (zeit-)kritische Teile des Systemkerns und verbreitete sich rasch innerhalb von Bell Laboratories (Mitte der 70er Jahre).

AT&T (die Mutterfirma von Bell) durfte anfangs dieses System aus kartellrechtlichen Gründen nicht vermarkten, also wurde es Universitäten zur Verfügung gestellt.

Unix wurde sowohl bei AT&T als auch an Universitäten fortentwickelt, speziell an der Berkeley Universität. Es entstand bei AT&T das UNIX System V (mit Untervarianten), bei Berkeley das BSD-Unix (Berkeley Software Distributions) z.B. 4BSD.

Nachdem sich beide Varianten auseinanderentwickelten, wurde Ende der 80er Jahre versucht, die Systeme zu vereinheitlichen. Dies ist nur teilweise gelungen.

Aufgrund des guten Designs waren nichtkommerzielle Programmierer daran interessiert, ein UNIX-System mit OpenSource-Lizenzmodell zu entwerfen. Die Programmierer der FSF (Free Software Foundation) haben im GNU-Projekt praktisch alle Werkzeuge von UNIX nachprogrammiert und verbessert (Ende der 80er Jahre). Die Programmierung eines Systemkerns wurde zurückgestellt.

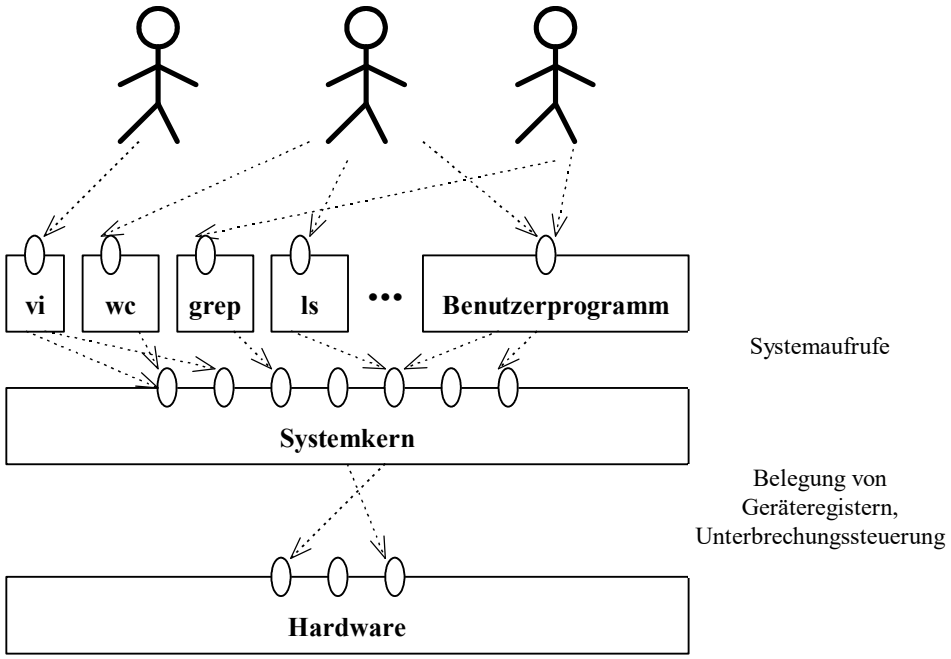
Ein Informatikstudent in Finnland, Linus Thorvalds, hat aus Spaß begonnen, ein simples Betriebssystem für Intel-Systeme zu bauen (Anfang der 90er Jahre). Er entwickelte es zu einem Systemkern für UNIX, insbesondere mit Programmierunterstützung aus dem Internet. Dieses System bildet - zusammen mit den Werkzeugen aus dem GNU-Projekt - das LINUX-System, das frei mit Quellkode verfügbar ist.

2.2 Designziele des UNIX-Systems

- Timesharing-Mehrbenutzer-/Mehrprozeßsystem; jeder Benutzer kann viele Prozesse laufen lassen
- einfache Benutzerschnittstelle (shell)
- gleiche Abstraktion für Dateien, Geräte und Interprozeßkommunikation (Bytestrom)
- komponentenorientiert: komplexe Programme können aus einfachen zusammengebaut werden (Werkzeugkasten, Werkzeuge mit klar definierten Aufgaben)
- hierarchisches Dateisystem (ohne Laufwerksbuchstaben, mit zur Laufzeit bindbaren Unterbäumen)
- in einer höheren Programmiersprache („C“) geschrieben (==>einfach lesbar und wartbar) plus Schichtenarchitektur ==> portabel
- einfaches Design (keine ausgefeilten Algorithmen) ==> robust
- System von Programmierern für Programmierer (z.B. mit textorientierten Werkzeugen)

2.3 Systemaufbau

Ein UNIX-System ist wie folgt strukturiert:



- Auf oberster Ebene kommen Kommandos und Benutzerprogramme, die sich auch gegenseitig benutzen können (als Unterprozesse). Zwischen Kommandos und Programmen gibt es keinen Unterschied. Beide laufen im Benutzermodus.

- Zentrale Komponente ist der Systemkern. Er erbringt die Dienste wie Dateiverwaltung, Speicherverwaltung, Benutzeranbindung, Zugriffsschutz usw. Die Dienste werden von

der darüberliegenden Schicht über Systemaufrufe angefordert (technisch ein Trap). Davon gibt es nicht viele (circa 60), die aber über Bibliotheken gekapselt werden.

– Unterste Komponente ist die Hardware. Sie wird - wie erwähnt - über Geräteregister programmiert und meldet sich über Unterbrechungen zurück.

2.4 Benutzersicht

2.4.1 Dateisystem

Dateien in UNIX sind Bytefolgen, deren innere Struktur durch das Betriebssystem nicht vorgegeben ist (keine feste Satzlänge, keine Unterscheidung zwischen Binär- und Textdateien).

Das Dateisystem ist ein Baum von Dateien mit einer Wurzel namens `/` (zyklenfrei!). Jeder innere Knoten des Baums ist ein Verzeichnis, Blattknoten sind Verzeichnisse, normale Dateien oder spezielle Dateien (z.B. Geräte).

Man kann Dateien entweder mit einem absoluten Pfadnamen bezeichen (ausgehend von der Wurzel, z.B. `/home/tt/test.txt`) oder mit einem relativen (ausgehend vom aktuellen Verzeichnis des Prozesses, z.B. `tt/test.txt`).

Es gibt die Möglichkeit, Verweise (Links) auf Dateien zu machen. Man kann also eine Datei unter verschiedenen Pfadnamen erreichen. Es gibt hard links und soft links. Hard links sind gleichberechtigt; es sind Zeiger auf eine interne Datenstruktur. Ein soft link ist dagegen ein Aliasname für einen konkreten Dateinamen. Wenn es keine Datei mit diesem Namen gibt, zeigt der soft link ins Leere. Auch ein Anlegen einer Datei erzeugt einen Verweis auf den Inhalt. Wenn der letzte Verweis verschwindet, kann der Inhalt entfernt werden.

Für den Benutzer sind Geräte (Terminal, Festplatte) ebenfalls Dateien, d.h. Byteströme. Man kann sie lesen und schreiben (wenn man darf...).

Jede Datei hat ihre Zugriffsrechte kodiert in einem Bitvektor. Dazu hat jeder Benutzer eine eindeutige Nummer im System und gehört zu einer Gruppe. Jeweils drei Bits kodieren die Lese-/Schreib-/Ausführungsrechte für Eigentümer, Gruppe des Eigentümers und alle anderen.

Es gibt Systemaufrufe für Dateien für Anlegen, Öffnen, Lesen, Schreiben, Schließen und Link löschen. Prinzipiell kann man Verzeichnisse als Dateien behandeln, aber damit könnte ihre innere Struktur zerstört werden. Daher gibt es eigene Systemaufrufe für Verzeichnisse.

2.4.2 Prozesse

Ein *Prozeß* ist ein Programm in Ausführung. In UNIX sind sie durch eindeutige Nummern identifizierbar.

Ein neuer Prozeß wird durch den Systemaufruf "fork" erzeugt (als Klon). Der neue Prozeß ("Kind") erhält eine Kopie des Adreßraums des alten Prozesses ("Vater") und wird asynchron gestartet. Resultat von fork ist entweder 0 (im Kind) oder die Prozeßnummer des Kinds (im Vater). Meist wird dann der Kindprozeß durch anderes ausführbares Programm ersetzt (über den Systemaufruf

"exec"). exec kehrt nur dann zurück, wenn das neue Programm nicht erfolgreich gestartet werden konnte. Ein Vaterprozeß läuft asynchron weiter, kann aber synchron auf die Beendigung des Kindprozesses warten (via "wait").

Beispiel: Es gebe ein Programm `copy` mit zwei Parametern p1 und p2, das die Datei p1 in p2 kopiert. Wir wollen noch eine Hülle `copy-verbose` darum bauen (als Prozeß), das Begleittext ausgibt (gegen die UNIX-Philosophie!!). `copy-verbose aaa bbb` kopiert die Datei "aaa" in "bbb" und gibt Begleittext aus:

```
main (int argc, char *argv[])
  /* argc ist die Zahl der Kommandozeilenargumente,
     argv ein String-Array mit den Kommandozeilenparametern */
{
  printf("Kopieren...\n");
  if (fork() == 0) { /* Erzeugung des Kindprozesses */
    /* Ersetzen durch neues Kommando */
    exec("copy", argv[1], argv[2], 0);
    /* hierhin gelangt man nur, wenn exec fehlgeschlagen ist */
  }
  wait(0); /* warten, bis Kindprozeß fertig ist */
  printf("Kopieren fertig.\n");
}
```

Typisches Beispiel für einen Vaterprozeß ist der Kommandoentschlüssler von UNIX, die *Shell*. Sie ist ebenfalls ein Programm, das Eingaben entgegennimmt und Unterprozesse startet: die Kommandos.

Prozesse entstehen in UNIX wie folgt: Es gibt einen universellen Vater, den init-Prozeß. Aus ihm werden für jedes Terminalgerät ein getty-Prozeß abgeleitet, der auf ein Login an einem Terminal wartet. Sobald dieses erfolgt, wird ein login-Prozeß aufgesetzt, der die Passwort-Kontrolle übernimmt. Wenn das geklappt hat, wandelt sich der login-Prozeß in eine Shell um, die ab dann die Kommandos des Benutzers entgegennimmt.

2.4.3 Werkzeugkastenansatz

Kommandos lassen sich in den Shells oder in Kommandodateien für Shells (Shellskripten) auf zwei Arten erweitern: durch Ein-/Ausgabeumlenkung und durch Verkettung über Pipes.

Generell gilt: UNIX-Programme lesen ihre Eingabe vom Terminal und schreiben die Ausgabe dorthin.

Man kann der Shell aber mitteilen, daß sie den Datenfluß umlenken soll auf Dateien (E/A-Umlenkung) und man kann auch die Ausgabe eines Programms unmittelbar zur Eingabe des nächsten machen durch eine *Pipeline*.

Wesentlich für den Erfolg dieses Ansatzes ist, daß Kommandos in UNIX klar definierte Aufgaben erfüllen und insbesondere simple Eingaben erwarten und simple Ausgaben (ohne überflüssigen Text) erzeugen.

2.5 Dienste des Systemkerns

Der Systemkern bietet folgende Dienste an:

- Prozeßsteuerung: anlegen, beenden, wartend setzen, kommunizieren lassen;
- CPU-Zuteilung: möglichst fair, zeitscheibengesteuert;
- Hauptspeicherverwaltung: einerseits Mechanismen für gemeinsam benutzten Speicher, andererseits Speicherschutz; Auslagern von belegtem Hauptspeicher auf Hintergrundspeicher (Swappen, Seitenwechsel);
- Hintergrundspeicherverwaltung: Dateisystem und Zugriffsschutz
- Geräteverwaltung: Terminals, Platten, sonstige Schnittstellen

Die Dienste abstrahieren von physikalischen Gegebenheiten: Beispielsweise wird eine Abbildung nötig sein zwischen Ansteuerung von Platten in Sektoren und der Abstraktion, daß Dateien aus einem Bytestrom bestehen.

3 Shells

3.1 Grundlagen

Eine Shell ist ein Kommandointerpreter zwischen Benutzer und Betriebssystem. Shells sind normale Programme, die zeilenweise Information vom Benutzer abrufen und darauf sofort reagieren. Die Kommandos sind also Textzeilen. Meist wird ein Programm gestartet, das durch das erste Wort der Zeile benannt wird; ganz selten handelt es sich um in die Shell eingebaute Befehle (wie beispielsweise `cd` für Verzeichniswechsel).

Jeder Benutzer hat eine Shell, die in seinem Profil definiert ist; man kann aber zwischen diversen Varianten wählen und sogar selbst eine Shell schreiben. Typische Vertreter sind die

- sh (Original-Shell von Steve Bourne)
- bash ("Bourne again"-Shell des GNU-Projekts)
- csh (C-Shell)
- tcsh (erweiterte C-Shell)
- ksh (Korn-Shell)

Shells haben relativ viele Gemeinsamkeiten bezüglich Syntax. Sie unterscheiden sich in Syntax von Ablaufkonstrukten, von Variablendefinition und in Komfortfunktionen wie automatischer Ergänzung von Dateinamen oder Kommandohistorie. Von exotischen Shells abgesehen fallen alle in zwei Kategorien: die Shells mit Syntax analog zur `sh` (`sh`, `bash`) und die mit Syntax ähnlich zu C (`csh`, `tcsh`).

3.2 Funktion

Alle Shells funktionieren wie folgt:

```
WIEDERHOLE
  gib Eingabeaufforderung (Prompt) aus
  lies nächste Zeile
  istFertig := (Eingabeende nicht erreicht ODER Zeile="exit")
  WENN NICHT istFertig DANN
    interpretierte Shellspezifika in Zeile und entferne sie
    führe Kommando(s) in Zeile aus
  ENDE-WENN
SOLANGE BIS istFertig
```

3.3 Syntax

Die Syntax eine Eingabezeile für die Shell ist wie folgt: Zeilen werden durch Leerzeichenfolgen in Worte eingeteilt. Das erste Wort bestimmt das auszuführende Programm, die restlichen Worte sind Parameter dieses Programms; es gibt also keine Klammern für die Parameterliste. Welche Parameter zulässig sind, wird durch das Programm festgelegt. Als Konvention werden Optionen eines Kommandos durch ein Minuszeichen eingeleitet und bestehen oft nur aus einem Buchstaben. Optionen modifizieren den voreingestellten Ablauf von Kommandos. Die meisten Kommandos

haben eine Vielzahl von Optionen. Diese können manchmal auch einen Wert enthalten, der als nächster Parameter auf die Option folgt.

Beispiele:
`ls` gibt die Namen aller Dateien in einem Verzeichnis aus und `ls -l` eine detaillierte Liste aller Dateien. `gcc` übersetzt ein C-Programm und die Option `-o` gibt an, wohin die ausführbare Datei abgelegt werden soll. Bei `gcc xyz.c -o fred` wird der C-Quelltext `xyz.c` übersetzt und in die ausführbare Datei `fred` abgelegt.

3.4 Kommandofolgen, E/A-Umlenkung, Pipes

Die Shell interpretiert zwar zeilenweise, aber man kann in einer Zeile mehrere Befehle durch Strichpunkt trennen. Sie werden nacheinander ausgeführt und auch die Ausgaben der Kommandos erfolgen nacheinander. Beispiel

$$kommando_1 \ ; \ kommando_2; \ kommando_3$$

Generell lesen Programme von Standardeingabe und schreiben auf die Standardausgabe. Man kann jedoch die Information umleiten und von einer Datei lesen bzw. auf eine Datei schreiben. Diese Umleitung wird durch Größer- und Kleinerzeichen gekennzeichnet. Sie ist beliebig anwendbar und für das Programm unsichtbar. Beispielsweise schreibt

```
ls -l /usr >otto
```

den Inhalt des Verzeichnisses `/usr` in Langform in die Datei `otto`. Das Größerzeichen besagt also: „leite Ausgabe um in“. Analoges gilt für die Eingabeumlenkung. Das Kommando

```
wc <hugo
```

zählt die Zeilen, Worte und Zeichen seiner Eingabe, hier also der Datei `hugo`.

Man kann auch die Ausgabe eines Programms unmittelbar zur Eingabe des nächsten machen durch eine *Pipeline*. Das Kommando

```
ls /usr | wc -l
```

zählt die Anzahl der Zeilen in der Ausgabe von `ls /usr`, gibt also die Zahl der Dateien im Verzeichnis `/usr` aus.

3.5 Hintergrundprozesse

Die Shell wartet üblicherweise darauf, daß der gestartete Prozeß beendet wird und gibt erst dann eine neue Eingabeaufforderung aus. Man kann aber auch der Shell mitteilen, daß der Prozeß im Hintergrund laufen soll und die Shell sofort für Kommandointerpretation zur Verfügung steht. Dies passiert durch das Sonderzeichen "&" am Ende des Kommandos.

3.6 Kontrollstrukturen

Shells verstehen Kontrollstrukturen. Eine Wiederholungsschleife in einer Bourne-Shell sieht wie folgt aus:

```
while Kommandofolge1
do Kommandofolge2
done
```

eine Bedingung wie folgt:

```
if Kommandofolge1
then Kommandofolge2
else Kommandofolge2
fi
```

Die Syntax ist streng, d.h. Schlüsselworte müssen in der Zeile vor anderen Dingen stehen, die Kommandofolgen können auch in eigenen Zeilen stehen. Kommandofolgen sind durch Strichpunkte getrennte Kommandos. Auch eine Bedingung wird also durch ein Kommandofolge ausgedrückt. “Wahr” bedeutet, daß das letzte Kommando in `Kommandofolge1` erfolgreich war (liefert einen positiven Rückgabewert), “falsch” bedeutet, daß das letzte Kommando gescheitert ist.

In der Regel möchte man keine Kommandofolge als Bedingung angeben, sondern eine richtige Bedingung. UNIX-Trick: es gibt ein Kommando `test`, das Bedingungen als Parameter hat, sie auswertet und entsprechend erfolgreich ist oder scheitert. Beispielsweise liefert `test $1 = "x"` “wahr”, wenn $1 nach Expansion gleich x ist und “falsch” sonst (Achtung: Leerzeichen sind signifikant, `test` hat drei Parameter...). Als Abkürzung kann man eckige Klammern um die Bedingung schreiben (wieder mit Leerzeichen abgesetzt), also statt `test $1 = "x"` einfacher `[ $1 = "x" ]`

Es gibt auch die Operatoren `&&` bzw `||` (für UND und ODER wie in C oder C++). Bei $kommando_1$ `&&` $kommando_2$ wird der zweite Befehl nur ausgeführt, wenn der erste erfolgreich war; bei $kommando_1$ `||` $kommando_2$ wird der zweite Befehl nur ausgeführt, wenn der erste gescheitert ist.

Eine Verteilungsanweisung sieht wie folgt aus:

```
case wort in
  muster1) Kommandofolge1 ;;
  muster2) Kommandofolge2 ;;
      ...
  musterN) KommandofolgeN ;;
esac
```

Das Wort (z.B. ein Kommandozeilenparameter) wird nacheinander mit allen Mustern verglichen. Beim ersten passenden Muster wird die entsprechende Kommandofolge ausgeführt. Muster enthalten Buchstaben und Metazeichen "*" für beliebige Buchstabenfolge, "?" für beliebigen Buchstaben und "[...]" für eine Menge von zulässigen Buchstaben. Das letzte Muster ist daher oft `"*"`, weil es auf alles paßt.

Eine FOR-Schleife hat folgende Syntax:

```
for name in wort1 wort2 ...
do Kommandofolge
done
```

Die Shellvariable (s.u.) `name` wird nacheinander mit `wort1, wort2` usw. belegt und damit die Kommandofolge durchgeführt.

3.7 Shellvariablen

Shellvariablen sind Variablen mit Namen und einer Zeichenkette als Inhalt. Ein Name beginnt mit einem Buchstaben und darf zusätzlich Ziffern oder Unterstreichungszeichen enthalten.

Eine Wertdefinition findet durch Zuweisung statt, beispielsweise belegt

```
benutzer=hase
```

die Variable `benutzer` mit dem Wert `hase`.

Benutzt wird eine Variable durch Voranstellung von $. Also würde `echo $benutzer` als Ergebnis `hase` ausgeben. Anwendung von Shellvariablen sind Expansionen und textuelles Einkopieren. Es gibt Situationen, wo dieser Text direkt an anderen anschließen soll. Man kann prinzipiell den Namen einer Variablen bei Anwendung in geschweifte Klammern schreiben also z.B. würde `echo ${benutzer}nfuss` als Ergebnis `hasenfuss` ausgeben.
`echo $benutzernfuss` klappt ebensowenig wie `echo $benutzer nfuss`(warum?).

Vordefinierte Variablen sind beispielsweise:

? Ergebniswert des letzten Kommandos

$ Prozeßnummer der aktuellen Shell (nützlich für eindeutige Namensdefinition von temporären Dateien z.B. temp$$)

@ sämtliche Parameter der aktuellen Shelldatei

HOMEVerzeichnis, das beim Einloggen erreicht wird (Standardwert bei `cd`)

PATH Liste von Verzeichnissen mit Kommandos (getrennt durch Doppelpunkt) z.B. `PATH=.:/bin:/usr/bin:$HOME/bin`

Man kann sich alle Variablen mit dem Kommando `set` ansehen. Shellvariablen sind normalerweise lokal für den jeweiligen Shellprozeß. Man kann eine Shellvariable aber für alle Shells mit `export variablenname` sichtbar machen.

3.8 Kommandodateien

Eine Shell kann auch ihre Kommandos aus einer Textdatei lesen und ausführen. Das Kommando

```
sh textdatei argument1 argument2 ...
```

bewirkt, daß die Shell die Kommandos aus der Textdatei liest. Die Argumente werden in der Kommandodatei über die Shellvariablen $1, $2 usw. angesprochen.

Wenn beispielsweise in der Datei `countFiles` die Zeile `ls $1 | wc -l` steht, bewirkt das Kommando

```
sh countFiles /bin
```

daß das Kommando `ls /bin | wc -l` ausgeführt wird. Damit gibt `countFiles` die Anzahl der Dateien in dem als Argument gegebenen Verzeichnis zurück.

Wenn eine Kommandodatei im Suchpfad der Shell liegt und mit dem Zugriffsattribut "ausführbar" versehen ist, kann es direkt aufgerufen werden (ohne `sh`).

Lokale Variablendefinitionen werden nur dann in der umschließenden Shell sichtbar, wenn sie exportiert werden. Dies leistet beispielsweise `export benutzer tempDatei` für die zwei angegebenen Variablen.

3.9 Textschutz, Dateinamenerzeugung und Kommandosubstitution

Manchmal möchte man die Interpretation von Teilen der Parameterliste durch die Shell ausschalten und die entsprechenden Texte vor der Shell schützen. Dies kann auf mehrere Arten passieren:

- für einzelne Buchstaben durch Voranstellen eines Rückwärtsschrägstrichs (z.B. `datei\ $name`) - auch vor einem Zeilenvorschub - oder
- für größere Textpassagen durch Klammern mit Einfach- oder Doppelapostrophen (z.B. `"datei mit leerzeichen"` oder `'datei$name'`)

In einer Kommandozeile können Muster für Dateinamen enthalten sein. Sie haben dieselbe Syntax wie Muster in case-Anweisungen, also die Metazeichen "*" für beliebige Buchstabenfolge, "?" für beliebigen Buchstaben und "[...]" für eine Menge von zulässigen Buchstaben.

Beispielsweise gibt `ls -l /home/[a-z]*/??` alle Dateien in Benutzerverzeichnissen aus, die aus exakt zwei Buchstaben bestehen.

Zu beachten ist, daß ein Muster sich nur auf Dateinamen in einem Verzeichnis bezieht und nicht über einen Schrägstrich hinausgehen kann (`/home/hugo/test/ab` würde also nicht durch das obige Muster abgedeckt). Außerdem werden Dateien, die mit Punkt beginnen, nicht erfaßt (der Punkt muß explizit angegeben werden).

Man kann die Standardausgabe eines Kommandos direkt als Text in der Shell verwenden, wenn man dieses Kommando in Rückwärtsapostrophe einbettet. Beispielsweise legt man mit

```
verzeichnis=`pwd`
```

in der Shellvariable `verzeichnis` den Wert des aktuellen Verzeichnisses ab. Der gesamte String zwischen den Rückwärtsapostrophen wird als Kommando ausgewertet und durch seine Standardausgabe ersetzt. `wc -c `ls /bin`` gibt die Größe für alle Dateien im Verzeichnis /bin aus (das geht hier auch mit `wc -c /bin/*`).

Auswertung einer Zeile für die Shell passiert wie folgt:

4. Shellvariablen- und Kommandosubstitution wird durchgeführt. Einmal ersetzte Zeichenketten werden nicht nochmals expandiert.

5. Danach wird die Zeile in Wörter (getrennt durch Leerzeichenfolgen) aufgeteilt. Durch Apostrophe geklammerter Text zählt als ein Wort. Einfachapostrophen schützen Text unbedingt, innerhalb von Doppelapostrophen findet Variablen- und Kommando-substitution statt.

Will man einen expandierten Text nochmals expandieren lassen, muß man dies explizit mit `eval` tun. So ergibt

```
x=fred
```

```
y=$x
eval echo $y
```

die Ausgabe fred.

4 Die Programmiersprache C

4.1 Lexikalische Konventionen

4.1.1 Allgemeines

C ist frei formatierte Sprache: Leerzeichen, Tabulatoren, Zeilenvorschübe und Kommentare spielen keine Rolle außer zur Trennung von syntaktischen Einheiten (*Tokens*) also z.B. zwischen zwei Bezeichnern. Es wird immer das längstmögliche Token genommen;

Kommentare werden mit /* ... */ geklammert und können nicht geschachtelt werden.

Bezeichner sind Folgen von Buchstaben (und "_") und Ziffern. Das erste Zeichen darf keine Ziffer sein; Groß-/Kleinschreibung ist wesentlich. Unterscheidung erfolgt auf 31 Stellen (in C++: auf alle).

Es gibt reservierte Bezeichner (*Schlüsselwörter*), die nicht anders benutzt werden dürfen.

4.1.2 Konstanten

Ganzzahlen werden normalerweise als Dezimalzahlen interpretiert, aber wenn Zahl mit "0" beginnt, dann als Oktalzahl und bei "0x" als Hexadezimalzahl.

Zeichenkonstanten sind eingeschlossen in einfache Anführungszeichen (z.B. 'p'). Der Wert eines Zeichens ist der numerische Interncode (Ganzzahl!). Als Sonderzeichen ist '\' zur Darstellung nichtdruckbarer Zeichen einsetzbar (z.B. ist '\n' Zeilenvorschub, '\t' Tabulator und '\123' das Zeichen mit oktalem Interncode 123).

Bei Fließpunktzahlen ist der Punkt das Dezimalkomma und der Exponententeil wird eingeleitet durch "E" oder "e".

Zeichenkettenkonstanten sind Folgen von Zeichen, die in Doppelanführungszeichen eingeschlossen sind. Sie werden behandelt als konstante Buchstabenfelder; der Übersetzer fügt als letztes Zeichen automatisch '\0' zur Terminierung an.

4.2 Deklarationen

Jeder Bezeichner bekommt im Programm durch eine Deklaration einen Typ zugeordnet. Ein Bezeichner benennt eine Variable oder Konstante;

Es gibt skalare Typen (Zahlen, Zeichen, Aufzählungstypen, Zeiger), Strukturtypen (Verbunde und Vereinigungen) und den Typ **void** ohne Ausprägung.

Eine Deklaration erfolgt durch Angabe des Typs und des Bezeichners (eventuell mit Zusatzinformation) z.B. **int** j

Meist ist komplexere Notation notwendig. Idee dabei ist, daß bei einer Deklaration ein Bezeicher so notiert wird wie bei einer Anwendungsstelle, an der er den entsprechenden Typ ergibt (z.B. **int** (*a) [22]).

Deklarationen werden durch ";" abgeschlossen; mehrere Bezeichner können durch Komma getrennt gemeinsam einen Typ zugeordnet bekommen (**int** a, b, c;)

4.2.1 skalare Typen

Grundtypen sind **char** (Zeichen), **int** (Ganzzahlen), **float** (Fließpunktzahlen), **double** (doppelt genaue Fließpunktzahlen), **enum** (Aufzählungstypen).

Diese Grundtypen können durch Genauigkeitsangaben (**short**, **long)** und Vorzeicheninformation (**signed**, **unsigned**) ergänzt sein.

Kombinationen von maximal einem Element aus allen drei Gruppen sind möglich; Standardwerte sind **signed** und **int**;

```
unsigned float ff;
signed x;        entspricht        signed int x;
unsigned long double abc;
```

Aufzählungstypen werden durch das Schlüsselwort **enum** eingeleitet; in geschweiften Klammern stehen mögliche Wertausprägungen:

enum {rot, grün, blau} ampelfarbe;
dann kann später `ampelfarbe` z.B. den Wert `rot` haben;

Für logische Werte gibt es keinen eigenen Typ: sie werden abgebildet auf **int** (0 ist FALSE, Rest TRUE);

4.2.2 Zeiger

Variablen vom Typ "Zeiger" speichern die Adresse einer Variable. Die Zuordnung einer Adresse zu einer Variable erfolgt häufig mittels des Adressoperators. Zeiger sind typgebunden, d.h. der Typ der Variable, auf die verwiesen wird, ist fest.

Deklaration z.B. eines Zeigers auf eine Ganzzahl

```
int *zeiger;
```

*-Operator dient in Anweisungen zur Dereferenzierung des Zeigers;

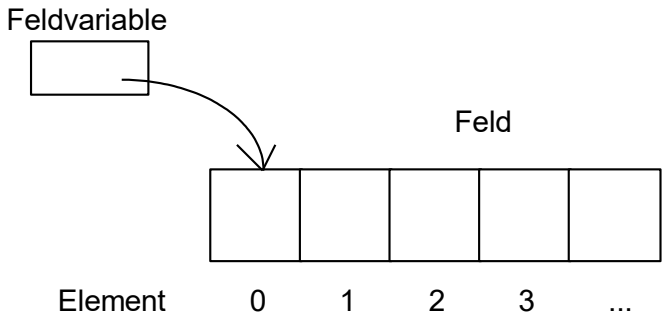
Vorsicht, Falle: `int *a, b` deklariert einen Zeiger und eine Ganzzahlvariable!

Um solche Fallen auszuschließen und auch portablen Kode zu erzeugen ist generell eine Typdefinition mittels **typedef** sinnvoll:

```
typedef int * ganzzeiger;
ganzzeiger a, b;
```

4.2.3 Felder

Felder sind kein eigenes Konzept in C. Im Prinzip sind Feldvariablen Zeiger auf das erste Element des Feldes. Die Indices eines Feldes beginnen mit 0.



Die Deklaration **int** `feld[100]` deklariert Ganzzahlfeld mit 100 Elementen "feld[0]" bis "feld [99]"; (genaugenommen ist "feld" Zeiger auf die Ganzzahl "feld[0]"). Mehrdimensionale Felder sind durch mehrfache Anwendung der Klammeroperatoren realisierbar (keine Kurznotation a[3,7] möglich!).

4.2.4 Strukturen (Verbunde, Vereinigungen)

Häufig werden mehrere unterschiedliche Informationen zu einer Datenstruktur zusammengefaßt. Diese heißt Verbund.

Die Variable `eineKomplexeZahl` hat nach der Deklaration

```
struct { double re; double im } eineKomplexeZahl
```

die Komponenten (Teilvariablen) "re" und "im" (jeweils vom Typ **double**) und man kann auf diese später Zugriff durch `eineKomplexeZahl.re` und `eineKomplexeZahl.im` zugreifen.

Wenn Information wahlweise zur Verfügung stehen soll, dann kann man Vereinigungen verwenden. Die Variable `irgendwas` hat nach Deklaration

```
union { double dies; int jenes } irgendwas;
```

die Komponenten "dies" und "jenes"; jeweils ist nur eine aktiv (der Speicher wird überlagert).

Empfehlung: eher mit Vorsicht verwenden, selten wirklich nützlich.

4.2.5 Funktionsköpfe

Man kann Signaturen von Funktionen deklarieren (Name der Funktion, Parametertypen und Rückgabewert).

Die Deklaration

unsigned int diff_tage (**tag** ta, **tag** tb);

deklariert "diff_tage" als Funktion mit zwei Parametern vom Typ **tag** und Resultatwert **unsigned int**.

Nutzen von Signaturdeklarationen sind folgende:

- Wenn Funktion vor Definition verwendet werden muß, kann der Übersetzer die korrekte Verwendung prüfen (sog. Prototypen);
- Man kann auch Zeiger auf Funktionen haben: diese Zeiger sind gebunden an den konkreten Typ von Funktionen z.B. **int** (***fp**)(**int, int**). Vorteil: man kann Funktionen dynamisch zuweisen.

4.2.6 Initialisierungen

Jede Variable kann bei der Deklaration durch eine Zuweisung initialisiert werden. Werte der Zuweisung müssen konstante Ausdrücke sein Die Deklaration wird dann zur Definition.

Bei Strukturen und Feldern werden die zugewiesenen Elemente durch geschweifte Klammern begrenzt.

Beispiele:

```
int abc=10+2*3;
float f[20], *fzeiger=&f[5];
int einszwodrei[3] = { 1, 2, 3 };
```

4.2.7 Speicherungsklassen

Zu jeder Variable kann ihre Speicherungsklasse angegeben werden:

- **auto**: eine Variable gilt von ihrer Deklaration bis zum Ende des Blocks, in dem sie deklariert wurde (Standard);
- **static**: eine Variable behält ihren Wert bei Verlassen des Blocks, in dem sie deklariert wurde, ist aber nur dort sichtbar;
- **register**: eine Variable sollte in einem Register des Prozessors abgelegt werden; sonst wie **auto** (heutzutage extrem unüblich);
- **extern**: eine Variable ist in einer anderen Übersetzungseinheit (Quelldatei) definiert, wird aber in der aktuellen verwendet;

4.3 Ausdrücke

4.3.1 Elementarausdrücke

Folgende Terme sind Elementarausdrücke:

Bezeichner	
Konstante	
Zeichenkette	
(Ausdruck)	
Elementarausdruck [Ausdruck]	
Feldelementauswahl	
Elementarausdruck (Ausdruck, ...)	Funktionsaufruf
Elementarausdruck->Bezeichner	Dereferenzierung
Elementarausdruck.Bezeichner	Selektion

Anmerkung zur Dereferenzierung: Der Elementarausdruck bezeichnet hier einen Zeiger auf eine Struktur. Dereferenzierung und Selektion werden dann in einem Operator kombiniert.

4.3.2 Unäre Operatoren

Folgende Terme definieren unäre Operatoren:

* Ausdruck	Dereferenzierung
& Ausdruck	Adressbestimmung
! Ausdruck	logische Negation
- Ausdruck	
++ Ausdruck	Präinkrementierung
-- Ausdruck	Prädekrementierung
Ausdruck ++	Postinkrementierung
Ausdruck --	Postdekrementierung
(Typname) Ausdruck	Typanpassung

Anmerkungen zur Bedeutung:

- Dereferenzierung: der Zeiger wird einmal verfolgt (zum bezogenen Objekt);
- Adressbestimmung: die Adresse eines Objekts wird ermittelt;
- Prä-/Post-In/Dekrementierung: der Ausdruck muß eine Variable bezeichnen; diese wird nach Auswertung verändert (schmutzig!); Zeigervariablen werden um die Größe des bezogenen Typs erhöht.

```
x = i++;        /* x erhält Wert von i und i wird erhöht */
```

- Typanpassung: Falls der Typ des Ausdrucks nicht passend ist, kann er passend gemacht werden.

4.3.3 Sonstige Operatoren

Sonstige Operatoren sind

Arithmetik: +, -, *, /, % (modulo);
Relationen: <, >, <=, >=, == (gleich!!!) , != (ungleich);
Bitoperatoren: &, |;
Logische Operatoren (mit Abbruch der Auswertung): &&, ||;
IF-THEN-ELSE-Ausdruck: Ausdruck **?** Ausdruck **:** Ausdruck
z.B. x>0 ? x : -x
Zuweisungsoperator:
Ausdruck = Ausdruck (Unterscheidung zu Vergleich s.o. !!!)
Wert ist Wert der linken Seite; er ist auch in Kombination mit Operator zulässig
z.B. x += 2 gleichbedeutend mit x = x + 2
Kommaoperator:
Ausdrücke werden nacheinander ausgewertet, aber nur der letzte verwendet;

4.4 Anweisungen

Eine normale Anweisung ist ein durch ";" abgeschlossener Ausdruck.

Ein Block enthält Folge von Deklarationen (jeweils mit Strichpunkt abgeschlossen) und Folge von Anweisungen und ist geklammert durch {}.

Als Kontrollflußanweisungen gibt es in C

if (Ausdruck) Anweisung **else** Anweisung
while (Ausdruck) Anweisung
do Anweisung **while** (Ausdruck)
for (Init_Ausdr; Bed_Ausdr; Fortsch_Ausdr) Anweisung
switch (Ausdruck) Anweisung
mit **case**-Teilen
break als Unterbrechung einer Schleife
continue als Fortsetzung am Schleifenanfang
return Ausdruck

4.5 Programm

Ein Programm ist eine Folge von Externdefinitionen, d.h. Objekte, die nach außen sichtbar sind.

Externdefinitionen sind

- Variablendeklarationen z.B. **int** x = 23;
- Funktionsdeklarationen: Sie bestehen jeweils aus Funktionskopf (siehe oben) und Funktionsrumpf (einem Block) z.B.

```
int max (int a, int b)
     { if (a<b) return b; else return a;}
```

Einstieg in ein Programm ist die Funktion `main`.Sie hat meistens die Signatur

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

argc: Anzahl der Argumente auf Kommandozeile
argv: Argumente selbst (Feld von Strings);

4.6 Präprozessor

In C besonders wichtig (in C++ nicht so sehr): Quelltexte können um sog. Präprozessoranweisungen angereichert sein. Der Präprozessor ist ein Programm, das vor dem Übersetzerlauf textuelle Manipulationen vornehmen kann.

PP-Anweisungen beginnen mit # in der ersten Spalte und werden vom Präprozessor verarbeitet und entfernt.

Arten von Präprozessoranweisungen sind

- Makros:

  ```
  #define Bezeichner Ersetzungstext
  ```

  jedes Auftreten des Bezeichners in der Quelle wird durch den Ersetzungstext ersetzt.

`#define Bezeichner(Bezeichner,...) Ersetzungstext`
der Bezeichner wird an allen Stellen in der Quelle wie eine Textschablone mit Parametern expandiert.

- Dateieinfügung:

  `#include "dateiname"`
  die angegebene Datei wird an aktueller Stelle textuell einkopiert, eine Schachtelung bei Einfügungen ist möglich. Meist wird dies zur Einbindung von Schnittstellendefinitionen in eine Übersetzungseinheit verwendet (aufgrund des fehlenden Modulkonzepts in C/C++).

- selektive Übersetzung (nach Platform oder ähnlichem):

  `#ifdef Bezeichner  #else  #endif`
  wenn Bezeichner für den Präprozessor definiert ist (über eine `#define`-Anweisung), bleibt der Kodeteil bis `#else` erhalten, ansonsten der Kodeteil zwischen `#else` und `#endif`

5 Prozesse

Ein *Prozeß* ist ein Programm in Ausführung. Alle Prozesse werden in einer Prozeßtabelle geführt, deren Information mit dem UNIX-Kommando `ps` abgefragt werden kann.

Er kann "neu", "laufend im Benutzermodus", "laufend im Systemmodus", "bereit zur Ausführung", "wartend" und "Zombie" sein. In den Zuständen "bereit zur Ausführung" und "wartend" kann ein Prozeß sowohl im Hauptspeicher als auch auf Hintergrundspeicher ausgelagert sein. Abbildung 5-1 zeigt diesen Zustandsautomat im Detail. Die gestrichelten Linien spielen bei der Signalerkennung später eine Rolle. In der Prozeßtabelle wird der Status als "R" (für laufend), "S" (für schlafend), "T" (für gestoppt) und "Z" (für Zombie) vermerkt.

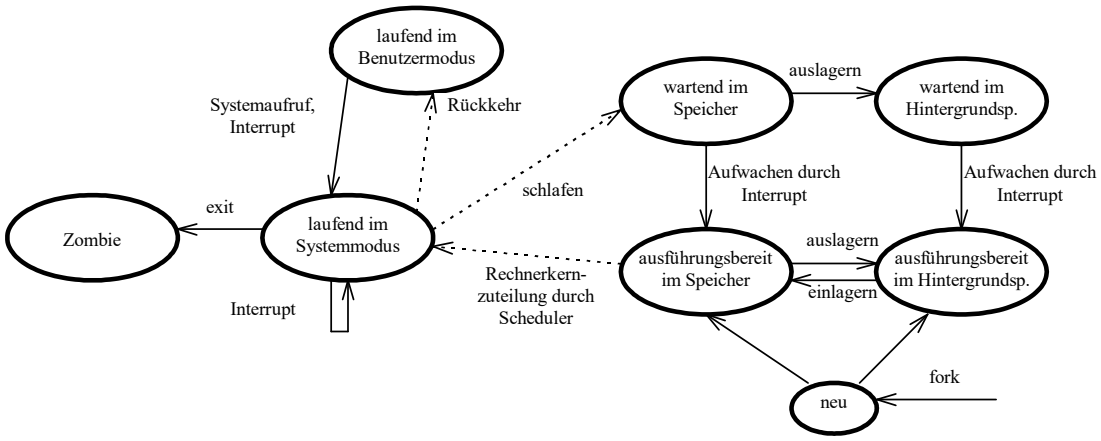


Abbildung 5-1: Zustandsübergänge von Prozessen in UNIX

Wie bereits in Kapitel 2 erläutert, entsteht ein neuer Prozeß durch den Systemaufruf `fork`. Der Kindprozeß erzeugt Kopien aller Speicherbereiche des Vaterprozesses, die veränderbar sind. Damit werden auch alle offenen Dateien übernommen.

Das folgende Beispielprogramm gibt jede Sekunde eine Zahl aus, teilt sich dann in zwei Prozesse, die parallel jede Sekunde dieselbe Zahl ausgeben.

```
#include <stdio.h>
  #define Stdio_printf printf
#include <unistd.h>
  #define Unistd_fork  fork
  #define Unistd_sleep sleep

int main()
{
  int i = 0;

  while(i < 5) {
    Stdio_printf("%d\n", i++);
    Stdlib_sleep(1);
  }
  Unistd_fork();
  while(i < 10) {
    Stdio_printf("%d\n", i++);
```

```
    Unistd_sleep(1);
  }
  return 0;
}
```

Vater und Kind haben unterschiedliche Prozeßnummern. Man könnte sie über den Systemaufruf `getpid` abfragen (get process id), man kann aber Vater und Kind auch über den Rückgabewert von `fork` unterscheiden: `fork` liefert 0 (im Kind) oder die Prozeßnummer des Kinds (im Vater) und einen negativen Wert, wenn die Aufteilung fehlgeschlagen ist. Zugriff auf die Prozeßnummer des Vaters erfolgt via `getppid` (get parent process id).

Beispiel für idiomatische Benutzung von fork:

```
#include <unistd.h>
  #define Unistd_fork fork

int main()
{
  int childProcessId = Unistd_fork();
  if(childProcessId < 0) {
    /* fork() fehlgeschlagen */
  else if(childProcessId == 0) {
    /* hier steht Kode für Kindprozeß */
  } else {
    /* hier steht Kode für Vaterprozeß */
  }
  return 0;
}
```

Kommunikation zwischen Prozessen möglich über Dateien/Pipelines oder *Signale*.

5.1 Signale

Signale informieren Prozesse, daß ein asynchrones Ereignis eingetreten ist. Signale können aus einem Prozeß heraus explizit mittels des *kill*-Systemaufrufs erzeugt werden, oder aus dem Systemkern intern. Es gibt diverse Signaltypen. Sie tragen darüber hinaus keine weitere Information.

Es gibt nur einen begrenzten Satz von Signalen (mit symbolischen Namen). Sie entstehen beispielsweise, wenn ein Prozeß beendet wird, bei Fehlersituationen (Privilegverletzung, Ressourcenüberschreitung) oder bei externen Ereignissen vom Benutzer (Drücken von Ctrl-C wird in Signal SIGINT übersetzt).

Signale werden vom Systemkern erkannt und an die Prozesse weitergeleitet. Diese Signalerkennung erfolgt bei den gestrichelt gezeichneten Zustandsübergängen in Abbildung 5-1.

Ein Prozeß kann auf verschiedene Antworten auf ein Signal reagieren: er beendet sich oder setzt sich wartend, er ignoriert das Signal oder eine spezifische Funktion im Prozeß wird aufgerufen. Bei Fehlern im Programmablauf wird vor Beenden des Prozesses oft ein Speicherabzug (core dump) erzeugt.

Name	Abkürzung für	Beschreibung	Standardbehandlung
SIGABRT	abort	durch Funktion abort() ausgelöst	Speicherabzug und Ende
SIGALRM	alarm	Zeitintervall ist abgelaufen, das mit alarm() oder settimer() gesetzt worden ist	Ende
SIGCHLD	death of a child	ein Child-Prozeß ist beendet	Ignorieren
SIGCONT	continue	der gestoppte Prozeß soll fortgesetzt werden (z.B. bg oder fg nach Ctrl-Z)	Fortsetzen
SIGFPE	floating-point exception	arithmetischer Fehler (z.B. Division durch Null).	Speicherabzug und Ende
SIGHUP	hangup	der Prozeß ist von seinem kontrollierenden Terminal getrennt worden	Ende
SIGILL	illegal instruction	unzulässiger Befehl	Ende
SIGINT	interrupt from keyboard	Abbruch von der Tastatur (via Ctrl-C)	Ende
SIGQUIT	quit from keyboard	Abbruch von der Tastatur (via Ctrl-|).	Speicherabzug und Ende
SIGKILL	kill	unwiderrufliches Beenden eines Prozesses	unbeeinflußbar Ende
SIGSEGV	segmentation violation	unzulässiger Speicherzugriff (z.B. bei falschen Zeigern)	Speicherabzug und Ende
SIGPIPE	broken pipe	Schreiben in Pipe, obwohl der lesende Prozeß am anderen Ende beendet ist	Ende
SIGTERM	terminate	Aufforderung zum Abbruch	Ende
SIGUSR1, SIGUSR2	user-defined signal	Freigehalten für benutzerprogrammspezifische Zwecke.	Ende
SIGSTOP	stop process	Prozeß anhalten, z.B. an Haltepunkt im Debugger.	unbeeinflußbar Stop
SIGTSTP	stop typed at tty	Prozeß anhalten nach Aufforderung von der Tastatur (z.B. Ctrl-Z).	Stop
SIGTTIN	tty input for background process	Prozeß im Hintergrund versucht von der Konsole zu lesen.	Stop
SIGTTOU	tty output for background process	Prozeß im Hintergrund versucht auf die Konsole zu schreiben.	Stop

Abbildung 5-2: Liste der POSIX.1-Signale

Als Beispiel für das Versenden von Signalen via `kill` (mit Parametern Prozeßnummer und Signal) betrachten wir das Zusammenspiel von zwei Prozessen, wobei der Kindprozeß drei Sekunden lang Zahlen ausgibt und dann von seinem Vaterprozeß beendet wird.

```
#include <signal.h>
  #define Signal_kill    kill
  #define Signal_SIGTERM SIGTERM
#include <stdio.h>
  #define Stdio_printf printf
#include <sys/types.h>
  #define SysTypes_ProcessID pid_t
#include <unistd.h>
  #define Unistd_fork  fork
  #define Unistd_sleep sleep

int main()
{
  SysTypes_ProcessID childProcessNumber = Unistd_fork();

  if (childProcessNumber < 0) {
    Stdio_printf("fork() failed");
  } else if (childProcessNumber == 0) {
    /* Kindprozeß */
    int i;
    for(i = 0; ; i++) {
      Stdio_printf("%d\n", i);
    }
  } else {
    /* Vaterprozeß */
    Unistd_sleep(3);
    Signal_kill(childProcessNumber, Signal_SIGTERM);
  }
  return 0;
}
```

5.1.1 Signalbehandlung in Prozessen

Wie in Abbildung 5-2 beschrieben, gibt es diverse Signale, die in einem Prozeß oft eine standardisierte Behandlung erhalten. Man kann aber für diverse Signale spezielle Reaktionen definieren. Beispielsweise kann es sinnvoll sein, daß ein Prozeß bei Abbruch nicht einfach terminiert, sondern vorher erst Ressourcen geordnet abschließt also z.B. Information auf Datei ablegt oder ähnliches.

Man erreicht dies dadurch, daß man für Signale Behandlungsroutinen festlegt durch den Systemaufruf `signal`.

```
alteFunktion = signal(signalCode, neueFunktion)
```

bewirkt, daß beim nächsten Auftreten von Signal `signalCode` die Behandlungsfunktion `neueFunktion` aufgerufen wird. `alteFunktion` ist ein Zeiger auf die bisherige Behandlungsfunktion für dieses Signal. `SIG_IGN` und `SIG_DFL` stehen für Standardbehandlungen von Signalen.

Wenn der Systemkern ein Signal erkannt hat und die Kontrolle wieder an den entsprechenden Prozeß im Benutzermodus zurückgehen soll, wird das Signal behandelt:

- Ist für einen Prozeß keine spezifische Behandlungsroutine definiert, ist die Reaktion einfach: Ein einmal zu ignorierendes Signal wird immer wieder ignoriert; die Standardbehandlung bricht meist den Prozeß ab und erzeugt einen Speicherabzug.

- Ist eine Behandlungsroutine definiert, passiert folgendes: Zuerst merkt sich der Systemkern die für das Signal definierte Signalbehandlungsroutine, löscht den Signalvermerk in der Prozeßtabelle und setzt die Routine dort auf den Standardwert. Danach sucht er den aktuellen Kellerpegel für den Benutzermodus und verändert den Keller so, daß bei Rückkehr in den Benutzermodus die Signalbehandlungsroutine angesprungen wird. Nach Beenden dieser Routine setzt das Programm an der Stelle fort, an der zuvor der Sprung in den Systemkern erfolgte.

Wichtig ist, daß nach jedem Signal die benutzerdefinierte Behandlungsroutine wieder eingerichtet werden muß!! Dies kann zu problematischen Effekten führen:

```
#include <signal.h>
  #define Signal_kill   kill
  #define Signal_signal signal
  #define Signal_SIGINT SIGINT
#include <stdlib.h>
  #define Stdlib_exit exit
#include <stdio.h>
  #define Stdio_printf printf
#include <sys/types.h>
  #define SysTypes_ProcessID pid_t
#include <unistd.h>
  #define Unistd_fork    fork
  #define Unistd_getppid getppid
  #define Unistd_sleep   sleep

void signalCatcher (int signalNumber)
{
  Stdio_printf("Interrupt behandelt.\n");
  Signal_signal(Signal_SIGINT, signalCatcher);
}

int main()
{
  if (Unistd_fork()>0) { /* Vaterprozeß */
    Signal_signal(Signal_SIGINT, signalCatcher);
    for (;;) ;  /* Warteschleife */
  } else { /* Kindprozeß */
    SysTypes_ProcessID parentProcessNumber = Unistd_getppid();
    Unistd_sleep(3); /* etwas warten */
    for (;;) { /* Vaterprozeß beschießen */
      if (Signal_kill(parentProcessNumber, Signal_SIGINT)!=0) {
        Stdlib_exit(1);
      }
    }
  }
}
```

Problem: Es könnte sein, daß der Vaterprozeß beim Aufruf von `printf` unterbrochen wird und der Rechnerkern wieder an sein Kind gegeben wird. Damit könnte dies ein Interrupt-Signal schicken, das zum Abbruch des Vaterprozesses führt!

5.1.2 Ende von Prozessen

Ein Prozeß wird beendet, wenn die Funktion `exit` im Programm aufgerufen wird oder wenn die Funktion `main` verlassen wird. Der Parameter von `exit` oder der Rückgabewert von `main` definieren den Endestatus eines Programms. Dieser wird an das aufrufende Programm zurückgemeldet. Als Konvention bedeutet 0, daß alles in Ordnung ist und ≠0, daß ein Fehler aufgetreten ist.

Bei Prozeßende werden alle Ressourcen freigegeben; der Prozeß wird zum "Zombie" und der Eintrag in der Prozeßtabelle merkt sich Statusinformation für diese Prozeßnummer (Ergebniswert, Laufzeit usw.). Alle Kindprozesse werden an den Init-Prozeß weitergegeben; für alle Zombie-Kinder wird ein Signal `SIGCHLD` an Init gesendet (der diese Zombies dann aus der Prozeßtabelle löscht). Dieses Signal wird auch an den eigenen Vaterprozeß geschickt und dann der Systemkern zur Rechnerkernvergabe betreten.

Manchmal ist ein Vaterprozeß nicht daran interessiert, wann ein Kindprozeß beendet ist. Daher wird standardmäßig SIGCHLD ignoriert. Oft wartet ein Vater auf die Beendigung seiner Kinder (mittels `wait`) oder eines bestimmten Kinds (mittels `waitpid`).

`pid_t wait(int *status)` liefert als Resultatwert die Prozeßnummer des beendeten Kinds und hat als Ausgabeparameter dessen Status. Über Makros kann man aus dem Status Teilinformationen extrahieren, beispielsweise mit WEXITSTATUS den Ergebniswert des Prozesses. `wait` legt den Prozeß schlafen, bis ein Kind Zombie-Status erreicht und löscht dann dieses Kind aus der Prozeßtabelle. Beispielcode für `wait` wäre

```
pid_t childProcessId;  int statusCode;  int childExitStatus;
/* warten auf Ende eines Kindprozesses */
childProcessId = Wait_wait(&statusCode);
childExitStatus = Wait_WEXITSTATUS(statusCode);
```

`pid_t wait(pid_t, int* status, int options)` liefert als Resultatwert die Prozeßnummer des beendeten Kinds und hat als Ausgabeparameter dessen Status. Mit der Prozeßnummer wird angegeben, auf welches Kind zu warten ist (-1 heißt auf alle Kinder!) und die Option sagt, ob gewartet wird (`options=0`) oder nur geprüft wird, ob der Kindprozeß bereits beendet ist (`options=WNOHANG`).

5.1.3 Starten anderer Programme

Will man nicht nur ein Duplikat eines vorhandenen Prozesses erzeugen, sondern stattdessen ein anderes Programm starten, kommt man mit `fork` alleine nicht weiter.

Der Systemaufruf exec ersetzt den aktuellen Prozeß durch ein neues Programm. Es gibt diverse Bibliotheksfunktionen, die `exec` kapseln. Gebräuchlich ist

```
execl(dateiname, argv[0], argv[1], ...)
```

Hier gibt `dateiname` den Pfadnamen des auszuführenden Programms an und die restlichen Parameter `argv[i]` Zeichenketten für die Parameter dieses Programms (mit Nullzeiger abgeschlossen). Außer den Parametern wird nichts in den neuen Prozeß übernommen. `exec` kehrt

auch niemals zurück, weil der neue Prozeß den alten ersetzt (damit verschwindet auch die Rücksprungstelle...).

```
#include <unistd.h>
  #define Unistd_execl execl
  #define Unistd_fork  fork

int main()
{
  if (Unistd_fork()>0) { /* Vaterprozeß */
    /* macht nichts */
  } else { /* Kindprozeß */
    /* Prozeß ersetzen durch "ls -l" */
    Unistd_execl("/bin/ls", "ls", "-l", NULL);
  }
}
```

Die Parameter eines Programms in `exec` sind die, die das Programm auch von einer Shell bekäme. Daher sind alle shellspezifischen Dinge wie z.B. E/A-Umleitung nicht möglich. Will man das verwenden, muß man eine Shell starten. Dies ist mit der Funktion

```
system(kommandozeile)
```

möglich. Nachteil dieser Funktion ist, daß sie neben den auszuführenden Befehlen eine Shell startet, Vorteil ist die sehr einfache Benutzung. Beispielsweise führt

```
system("ls -l /bin | tail +2 | wc -l");
```

eine einfache Pipeline aus, die die Zahl der Dateien in "/bin" bestimmt.

Eine Shell basiert auf dem exec-Aufruf. Eine einfache Shell könnte wie folgt aussehen:

```
#include <stdio.h>
  #define Stdio_gets gets
  #define Stdio_puts puts
#include <unistd.h>
  #define Unistd_execlp execlp
  #define Unistd_fork   fork
#include <wait.h>
  #define Wait_wait wait

#define loop for(;;)
#define maxLineSize 100
#define promptString "[hugo]> "

int main()
{
  loop {
    char commandLine[maxLineSize];
    Stdio_puts(promptString);
    Stdio_gets(commandLine);
    if (commandLine[0] == 'q'){ /* Abbruch bei 1. Buchstabe = q */
      break;
    }
    if (Unistd_fork() == 0) {
      Unistd_execlp(commandLine, commandLine, NULL);
    } else {
      Wait_wait(NULL);
    }
```

```
    }
}
```

Diese Shell ist ziemlich dumm, da sie beispielsweise alle Argument als erstes Argument an den aufgerufenen Prozeß übergibt (natürlich klappen auch keine Dinge wie *-Expansion usw.).

6 Dateirepräsentation

Dateisysteme legen Information über längere Zeiträume in benannten Informationsbehältern (Dateien) ab. Das UNIX-Dateisystem unterstützt sowohl einzelne Dateien als auch Gruppen von Dateien (Verzeichnisse). Verzeichnisse können Hierarchien bilden; sie sind jedoch lediglich Dateien mit einem speziellen Format, daher betrachten wir im folgenden nur die Struktur einfacher Dateien.

Generell gilt für UNIX, daß Groß- und Kleinschreibung bei Dateinamen signifikant ist. Die Namen dürfen bis zu 255 Zeichen lang sein und prinzipiell alle Zeichen enthalten.

6.1 Datenblöcke und Inodes

Hauptbestandteil des Dateisystems sind die Datenblöcke, die die Dateiinformation enthalten. Dateien enthalten in UNIX Bytefolgen und haben keine vorgegebene Satzstruktur. Die Datenblöcke haben eine feste Länge (z.B. 512 Byte) und sind für eine Datei in der Regel frei über den physikalischen Speicher verteilt.

Man muß einen Kompromiß eingehen zwischen kleiner Dateiblockgröße (besser bei vielen kleinen Dateien) und großer Dateiblockgröße (besser zu verwalten, höhere Transferrate).

Sämtliche Verwaltungsinformation einer Datei steht in ihrem Indexknoten (i-node), einem speziellen Datenblock. Dieser Inode enthält Information wie Dateilänge, aber auch Besitzer, Änderungsdaten usw. Zusätzlich steht in ihm eine Liste von Nummern der Dateiblöcke, die die Dateiinformation enthalten.

Im einzelnen findet man folgendes in Inodes:

- Anzahl der Verweise

  Ein Dateiinhalt kann in UNIX unter mehreren Namen verfügbar sein. In Verzeichnissen referenziert ein Dateiname einen beliebigen Inode. Damit man die Datei löschen kann, sobald der letzte Verweis verschwunden ist, wird eine Referenzzählung im Inode gemacht.

- Dateityp (Verzeichnis, einfache Datei, Gerät usw.)

- Liste von Datenblocknummern

  Dateien werden nicht in benachbarten Datenblöcken gespeichert, sondern frei verteilt. Der Inode enthält eine endliche Liste von Datenblocknummern (mit Länge *n*). Damit ein Inode aber mit festem Platz auskommt, wird ein Trick angewandt (die Datenblockgröße sei *b,* die Größe einer Datenblocknummer in Bytes sei *k*):

  - Datenblocknummern *1* bis *n-3* zeigen auf richtige *direkte Datenblöcke* der Datei. Damit können Dateien mit Länge<*(n-3)b* direkt adressiert werden.
  - Datenblocknummer *n-2* zeigt auf einen *einfach indirekten Indexblock*, der seinerseits nur Verweise auf direkte Datenblöcke enthält. Daher können also weitere *(b/k)b* Bytes erreicht werden.

- Datenblocknummer *n-1* zeigt auf einen *zweifach indirekten Block*, der seinerseits nur Verweise auf einfach indirekte Indexblöcke enthält. Damit werden also weitere $(b/k)^2 b$ Bytes erreicht.
- Datenblocknummer *n* zeigt auf einen *dreifach indirekten Block* für weitere $(b/k)^3 b$ Bytes.

Für n=15, k=8 und b=2048 ergeben sich maximale Längen von 24K, 536K, 131.608K und 33.686.040K, bei höherer Blockgröße steigt dieser Wert kubisch an.

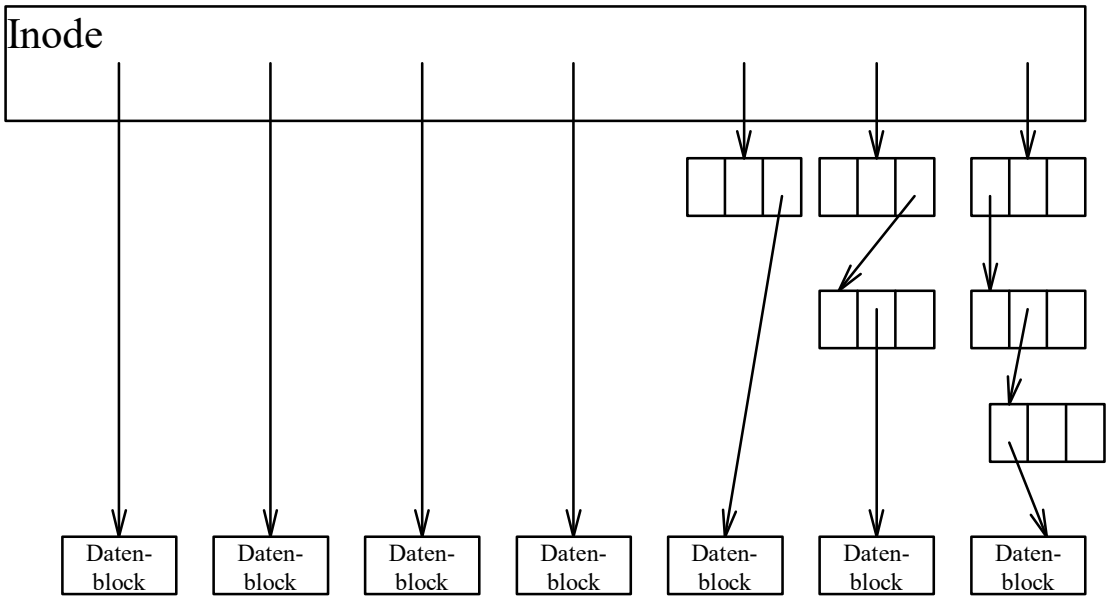


Bild 6-1: Verweisstruktur zwischen Inode und Datenblöcken

– Eigentümer, Gruppe und Zugriffsrechte

  Jedes Element im Dateisystem hat einen Eigentümer und eine Benutzergruppe, der es zugehört. Der Eigentümer kann auf "eigene" Dateien Zugriffsrechte vergeben für sich selbst, für Benutzer aus seiner Gruppe und für alle anderen.

  Benutzer und Gruppe werden jeweils über eine eindeutige Nummer im System identifiziert, die im Inode verwendet wird. Die Zugriffsrechte auf eine Datei werden kodiert in einem Bitvektor. Jeweils drei Bits kodieren die Lese(r)-/Schreib(w)-/Ausführungsrechte(x) für Eigentümer(u), Gruppe des Eigentümers(g) und alle anderen(o).

  Auf Dateien haben die Rechte die erwartete Bedeutung, auf einem Verzeichnis bedeutet "ausführbar", daß es in einem Pfad auf eine andere Datei traversiert werden darf.

– Zeitmarken:

  - modification: letzter Schreibzugriff auf Inhalt
  - access: letzter Lesezugriff auf Inhalt
  - change: letzter Schreibzugriff auf Inode (z.B. bei Hinzufügen eines Links);

Zeitangaben in UNIX sind die Anzahl der Sekunden seit Beginn der UNIX-Epoche (der 1.1.1970 um 0:00 Uhr). Meist werden dafür 32 Bit-Ganzzahlen genutzt; der Typ ist jedoch parametrisiert (`time_t`).

Diese Information kann mit "ls" gelesen und mit "touch" geändert werden.

Beispielsweise bedeutet bei ls -gl

```
-rwxr-x--- 5 tensi users 576 Jun 4 22:22 index.html
```

folgendes: Die Datei index.html im aktuellen Verzeichnis gehört dem Benutzer tensi und ist der Gruppe users zugeordnet. Es gibt 5 Dateinamen, die denselben Inhalt referenzieren (hard links); die Datei ist 576 Bytes lang, am 4.6.99 um 22:22 Uhr wurde der Inhalt zuletzt verändert und tensi kann lesen/schreiben/ausführen, users können lesen und ausführen und alle anderen dürfen nichts mit dieser Datei.

Mittels ls kann man auch die Inodenummer feststellen (via ls -i).

6.2 Dateisysteme

In UNIX sind alle Dateien in einem Verzeichnisbaum enthalten (keine Laufwerke). Unterschiedliche Dateisysteme (beispielsweise auf verschiedenen Platten) werden in den Baum eingehängt (via mount-Kommando). Die Position im Baum ist frei wählbar, die Wurzel des eingehängten Dateisystems wird mit einem inneren Knoten des Unix-Verzeichnisbaums identifiziert

Beispielsweise hängt

```
mount /dev/sda0 /usr
```

das Gerät `/dev/sda0` (eine SCSI-Platte) in den Baum als Verzeichnis `/usr` und dessen Unterverzeichnisse ein.

Beim Systemstart wird aus der Datei `/etc/fstab` gelesen, welche Einhängeoperationen durchzuführen sind. Mounten ist aber jederzeit möglich.

Unix unterstützt unterschiedliche Typen von Dateisystemen gleichzeitig (auch z.B. das FAT-System von DOS) und bildet sie auf sein internes Modell mit Inodes ab (gegebenenfalls mit gewissen Annahmen über Inode-Attribute). Ebenso werden unterstützt

- verteilte Dateisysteme (bei denen Dateien auf anderen Rechnern liegen) und
- virtuelle Dateisysteme (bei denen flüchtige Information abgebildet wird auf eine Datei, z.B. der Inhalt des Hauptspeichers).

6.3 Hard links, soft links

Wie bereits erwähnt gibt es aufgrund der Inode-Referenzierung möglicherweise mehrere Namen für eine UNIX-Datei.

Die direkte Referenzierung bezeichnet man als *hard link*. Mehrere Verzeichniseinträge haben dieselbe Inode-Nummer. Dies kann aber nur innerhalb eines Dateisystems funktionieren und nicht über ein Dateisystem hinweg, weil die Inodenummern gerätespezifisch sind.

Bei einem hard link sind alle Informationen im Inode und der Dateiinhalt identisch für alle referenzierenden Dateien. Änderungen an Inode oder Dateiinhalt betreffen sämtliche Referenzen.

Erstellt wird ein hard link mittels des Kommandos ln (für link) beispielsweise durch

```
ln alteDatei neueDatei
```

wird eine weitere Referenz "neueDatei" auf "alteDatei" erzeugt. Beide Referenzen sind absolut gleichberechtigt. Die Zahl der Referenzen auf einen Inode wird bei einer langen Verzeichnisdarstellung (ls -l) aufgeführt.

Soft links (*symbolische Links*) referenzieren eine andere Datei per Namen. Sie sind besondere Elemente in einem Dateisystem (ein spezieller Inode-Typ); der Inode enthält den Pfadnamen des anderen Partners. Symbolische Links erben auch die Attribute des Ziels (z.B. Zugriffsrechte).

Wesentlicher Vorteil eines soft links ist, daß er über Dateisystemgrenzen hinweg gehen kann und daß er auf ein beliebiges Element geht (möglichweise wieder auf einen symbolischen Link). Nachteilig ist, daß er möglicherweise ins Leere zeigen kann.

Erstellt wird ein soft link mittels des Kommandos ln mit Option -s beispielsweise durch

```
ln -s alteDatei neueDatei
```

wird eine symbolische Referenz "neueDatei" auf "alteDatei" erzeugt. Nur "alteDatei" zeigt auf den Inode des Dateiinhalts. Die Zahl der Referenzen auf den Inode darf nicht zugenommen haben..

6.4 Dateideskriptoren

Für den UNIX-Systemkern werden alle offenen Dateien eines Prozesses über eine positive Ganzzahl identifiziert: den Dateideskriptor. Auch die Standardeingabe, Standardausgabe und Standardfehlerausgabe sind mit Dateideskriptoren verbunden (konventionell 0, 1 und 2).

Für einen Prozeß ist es ohne Bedeutung, welche physischen Dateien hinter den Deskriptoren verborgen sind. Die Deskriptoren abstrahieren Kanäle, die Byteströme liefern oder entgegennehmen.

Generell sollte man davon ausgehen, daß Ein-/Ausgabeoperationen auf Dateien über Dateideskriptoren ungepuffert ablaufen. Dies muß nicht für die systemkerninterne Verwaltung gelten (in der Regel puffert der Systemkern die Information, bevor sie auf Hintergrundspeicher geschrieben wird), aber zumindest ist jeder Aufruf von E/A mit diesen Funktionen ein Systemaufruf.

Folgende Operationen sind via Dateideskriptoren definiert:

- Öffnen einer Datei (open):

  ```
  int open (const char *pathName, int openFlags)
  ```

  Der Pfadname gibt an, welche Datei zu öffnen ist, der zweite Parameter ist eine Menge von Optionen: Transportrichtung (Lesen, Schreiben, beides), Anhängen beim Schreiben oder nicht usw. Alle diese Optionen sind durch symbolische Konstanten repräsentiert, die per ODER-Operation verknüpft werden.

  `open` gibt als Ergebnis einen Dateideskriptor zurück (oder einen negativen Fehlercode).

- Erzeugen einer Datei (create):
  ```
  int creat (const char *pathName, mode_t mode)
  ```

  Der Pfadname gibt an, welche Datei neu zu erzeugen ist, im zweiten Parameter werden die Zugriffsrechte angegeben (der 9 Bit-Vektor als Zahl kodiert).

  `creat` gibt als Ergebnis einen Dateideskriptor zurück (oder einen negativen Fehlercode).

  Mittlerweile kann auch `open` zur Erzeugung von Dateien verwendet werden (über entsprechende Optionen). `creat` ist damit redundant.

- Lesen von einer Datei (read):
  ```
  ssize_t read (int fileDesc, void *buffer, size_t size)
  ```

  Der erste Parameter gibt den Deskriptor der Datei an, von der `size` Zeichen gelesen werden sollen in den Puffer `buffer`. Als Ergebnis liefert `read` die Anzahl der wirklich gelesenen Zeichen zurück (≥0) oder eine negative Zahl bei einem Fehler (`size_t` ist ein vorzeichenloser Typ, der eine Anzahl angeben kann; `ssize_t` der entsprechende Typ mit Vorzeichen).

- Schreiben auf eine Datei (write):
  ```
  ssize_t write (int fileDesc, void *buffer, size_t size)
  ```

  Der erste Parameter gibt den Deskriptor der Datei an, auf die `size` Zeichen geschrieben werden sollen aus dem Puffer `buffer`. Als Ergebnis liefert `write` die Anzahl der wirklich geschriebenen Zeichen zurück (≥0) oder eine negative Zahl bei einem Fehler.

- Schließen einer Datei (close):
  ```
  int close (int fileDesc)
  ```

  Der Parameter gibt den Deskriptor der Datei an, die geschlossen werden soll. Bei korrekter Verarbeitung wird 0 zurückgeliefert, bei Fehler eine negative Zahl.

  Unter UNIX werden bei Prozeßende alle Dateien automatisch geschlossen.

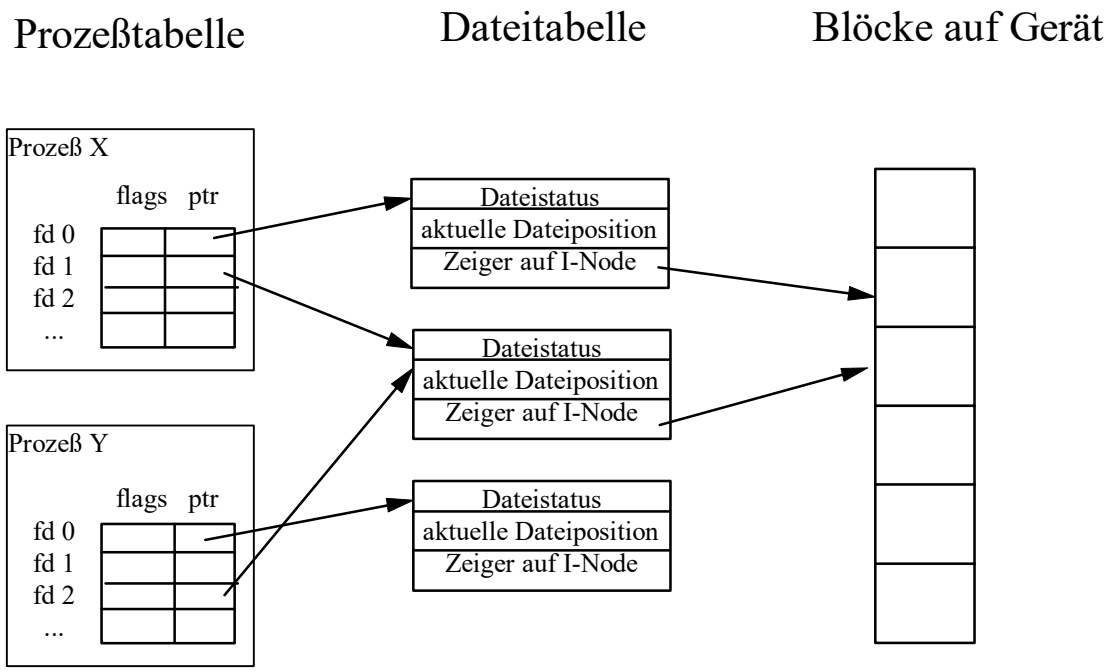


Bild 6-2: Interne Tabellen für Abbildung von Dateideskriptoren auf Dateien

Beispiel:

```
#include <fcntl.h>
  #define FCntl_creat       creat
  #define FCntl_open        open
  #define FCntl_openForRead O_RDONLY
#include <stdio.h>
  #define Stdio_fprintf fprintf
  #define Stdio_sprintf sprintf
  #define Stdio_stderr  stderr
#include <sys/types.h>
  #define Systypes_SizeType       size_t
  #define Systypes_SignedSizeType ssize_t
#include <unistd.h>
  #define Unistd_close close
  #define Unistd_exit  exit
  #define Unistd_read  read
  #define Unistd_write write

#define buffersize 128
#define loop for(;;)
typedef int FileDescType;

void errorExit(char *format, char* filename)
{
  char formatstring[200];
  Stdio_sprintf(formatstring, "%%s: %s\\n", format);
  Stdio_fprintf(Stdio_stderr, formatString, progName, filename);
  Unistd_exit(1);
}

int main(int argc, char *argv[])
  /* cp-Funktion aus UNIX */
{
```

```
  FileDescType infile, outfile;
  char *progName, buffer[buffersize];
  progName = argv[0];
  infile = FCntl_open(argv[1], FCntl_openForRead);
  if (infile<0) errorExit("can't open %s", argv[1]);
  outfile = FCntl_creat(argv[2], 0600); /* Bitvektor rw------- */
  if (outfile<0) errorExit("can't create %s ", argv[2]);
  loop {
    int bytesRead, bytesWritten;
    bytesRead = Unistd_read(infile, buffer, buffersize);
    if (bytesRead<0) errorExit("read error on %s", argv[1]);
    if (bytesRead==0) break;
    bytesWritten = Unistd_write(outfile, buffer, bytesRead);
    if (bytesWritten<bytesRead)
      errorExit("write error on %s", argv[1]);
  }
}
```

6.5 Pipes

Pipes sind ein sehr einfacher Mechanismus zur Prozeßkommunikation unter UNIX. Eine Pipe wird durch den Systemaufruf **`int`** `pipe(int fileDesc[2])` erzeugt.

Pipes sind Kanäle, wobei sowohl das Lese- als auch das Schreibende über einen Dateideskriptor im Prozeß verfügbar ist. Folgende Regeln gelten:

- Eine Pipe unterstützt nur einen unidirektionalen Datenfluß.
- Pipes können nur zwischen Prozessen benutzt werden mit gemeinsamen Vorfahrprozeß.

`pipe` liefert zurück, ob eine Pipe erzeugt werden konnte (<0 heißt: erfolglos) und gibt zwei Dateideskriptoren in `fileDesc` zurück: `fileDesc[0]` enthält das Lese-Ende der Pipe, `fileDesc[1]` das Schreib-Ende.

Eine Pipe in einem Prozeß ist nutzlos. Sinnvoll ist es, daß Vater und Kindprozeß Information über eine Pipe austauschen. Da alle Daten (also auch die Dateideskriptoren) bei einem `fork` dupliziert werden, ist das möglich.

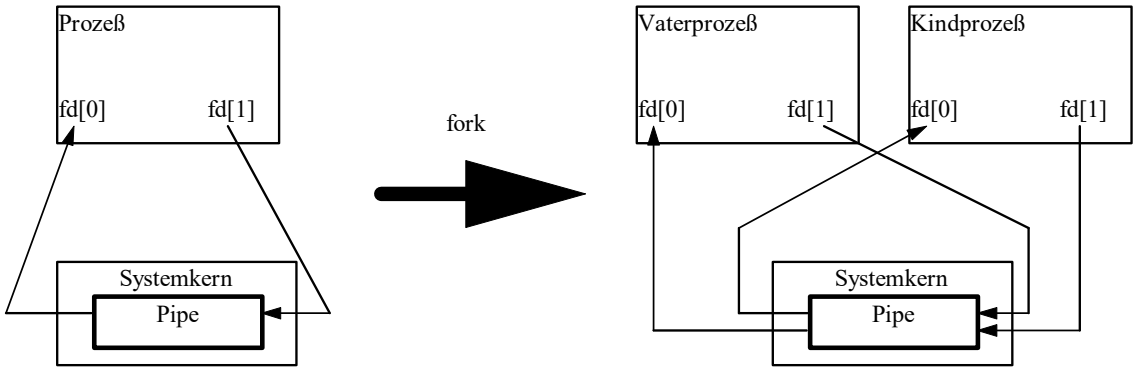


Bild 6-3: Deskriptoren für Pipe in abgeleiteten Prozessen

Beispiel:

```
#include <fcntl.h>
  #define FCntl_close  close
  #define FCntl_read   read
  #define FCntl_STDOUT STDOUT_FILENO
  #define FCntl_write  write
#include <unistd.h>
  #define Unistd_fork fork
  #define Unistd_pipe pipe

typedef int FileDescType;

int main()
{
  FileDescType filedesc[2];  char buffer[80];  int count;
  Unistd_pipe(filedesc);
  if(Unistd_fork() == 0) { /* Kindprozeß */
    FCntl_close(filedesc[1]);  /* nur lesen */
    count = FCntl_read(filedesc[0], buffer, 200);
    FCntl_write(FCntl_STDOUT, buffer, count);
  } else { /* Vaterprozeß */
    FCntl_close(filedesc[0]); /* nur schreiben */
    FCntl_write(filedesc[1], "Kuckuck ruft's aus dem Wald!", 28);
  }
}
```

Es ist guter Stil, wenn ein Programm die Dateideskriptoren schließt, die es nicht benötigt.

Wenn ein Ende einer Pipe geschlossen ist, passiert folgendes:

- Der Leser einer Pipe bekommt eine EndOfFile-Information (d.h. `read` liefert Länge 0), wenn alle Information aus der Pipe gelesen wurde.

- Der Schreiber einer Pipe bekommt das Signal SIGPIPE. Wird das Signal ignoriert oder der Signalbehandler kehrt zurück, erhält `write` den Fehlercode EPIPE. (Typische Fehlermeldung in der Shell: "pipe broken".)

Für bidirektionale Kommunikation braucht man zwei Pipes für Vater und Kindprozeß.

Beispielprogramm für Synchronisation zweier Prozesse über zwei Pipes:

```
#include <fcntl.h>
  #define FCntl_close  close
  #define FCntl_read   read
  #define FCntl_STDOUT STDOUT_FILENO
  #define FCntl_write  write
#include <unistd.h>
  #define Unistd_exit exit
  #define Unistd_fork fork
  #define Unistd_pipe pipe
typedef int FileDescType;

/* ends of a pipe */
#define readEnd  0
#define writeEnd 1

int main()
{
  /* Vater und Kindprozeß tun etwas Kompliziertes und müssen aber
     aufeinander warten, bevor sie weitermachen können. Dies wird
     über zwei Pipes realisiert. */
  FileDescType pipeUp[2], pipeDown[2];  char buffer[1];
  if (FCntl_pipe(pipeUp)<0 || FCntl_pipe(pipeDown)<0) {
    Unistd_exit(1);
  }
  if (Unistd_fork()==0) {
    FCntl_close(pipeUp[readEnd]);
    FCntl_close(pipeDown[writeEnd]);
    ... /* komplizierte Aktion im Kind*/
    FCntl_write(pipeUp[writeEnd], "c", 1);    /* Vater anstoßen */
    FCntl_read(pipeDown[readEnd], buffer, 1); /* warten auf Vatr*/
    ... /* noch eine komplizierte Aktion im Kind*/
} else {
    FCntl_close(pipeUp[writeEnd]);
    FCntl_close(pipeDown[readEnd]);
    ... /* komplizierte Aktion im Vater */
    FCntl_read(pipeUp[readEnd], buffer, 1);   /* warten auf Kind */
    FCntl_write(pipeDown[writeEnd], "p", 1);  /* Kind anstoßen */
    ... /* noch eine komplizierte Aktion im Vater */
  }
}
```

7 Unterbrechungsanforderungen, Ausnahmen

Prinzipiell wird der Befehlszyklus eines Prozessors ständig wiederholt: ein Befehl wird geholt, dekodiert und ausgeführt.

In einem Prozessor können jedoch Ausnahmesituationen auftreten (*exceptions*). Diese können auf unterschiedliche Weise entstehen:

6. durch einen Fehler bei der Abarbeitung eines Befehls (z.B. Teilen durch 0):
   Es ist eine Ausnahme, die synchron und im Prozessor auftritt; sie ist auch reproduzierbar. Sinnvollerweise sollte die Abarbeitung des Programms nicht einfach weiterlaufen, sondern dieser Sonderfall muß behandelt werden.

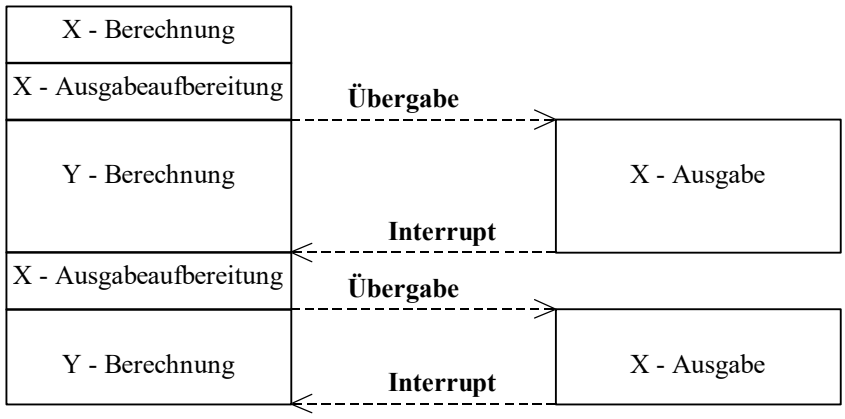
7. durch einen Befehl, der explizit eine Ausnahme auslöst (Trap-Befehl):
   Dies ist eine Ausnahme, die synchron und prozessorintern ist. Trap-Befehle dienen in der Regel als definierte Einsprünge in Routinen des Betriebssystems (ähnlich zu Unterprogrammsprüngen).

8. durch ein prozessorexternes, aber befehlssynchrones Ereignis (z.B. einen fehlerhaften Buszyklus):
   Hier handelt es sich um Ausnahmen, die an definierter Stelle im Befehlszyklus auftreten, aber von außerhalb kommen (Beispiel: Zugriff auf eine nicht vorhandene Speicherstelle). Auch dieser Sonderfall muß behandelt werden.

9. durch ein asynchrones Signal (z.B. von E/A-Einheiten, Rücksetzen des Systems):
   Für externe Ausnahmen gibt es Anschlüsse am Prozessor. Diese Ausnahmen treten irgendwann auf.

Asynchrone Signale von außen (letzter Fall) nennt man *Interrupts*, die anderen *Traps*.

Nicht immer ist eine Ausnahme ein Fehler! Wenn verschiedene Teile eines DV-Systems parallel arbeiten können (beispielsweise Prozessor und Peripherie), ist es unsinnig, daß der Prozessor auf das Ende einer E/A-Operation wartet. Meist wird die E/A-Operation relativ lange dauern; der Prozessor kann in dieser Zeit eine andere Aufgabe bearbeiten und wird dann über einen Interrupt informiert, daß die Operation fertig ist.



•••

Bild 1: Interrupt bei Parallelverarbeitung

Man könnte das Problem auch dadurch lösen, daß der Prozessor sporadisch nachfragt, ob die E/A-Operation schon fertig ist, aber das passiert meist entweder zu oft oder zu selten.

Ausnahmen werden durch spezielle Unterprogramme abgehandelt: das aktuelle Programm wird unterbrochen, die Ausnahme bearbeitet und danach das unterbrochene Programm - meist - fortgesetzt.

7.1.1.1 Bearbeitung von Ausnahmen

Traps lassen sich komplett prozessorintern abhandeln, Interrupts erfordern zusätzlichen technischen Aufwand:

- Da Interrupts asynchron passieren, kann nicht einfach sofort das Mikroprogramm beendet werden (sonst wird nicht reproduzierbarer interner Prozessorzustand erreicht). Vielmehr wird der aktuelle bearbeitete Befehl beendet und dann der Interrupt bearbeitet. Eine Unterbrechung eines Befehls ist nur möglich, wenn ein Prozessor seinen internen Status auf den Wert vor Befehlsauführung restaurieren kann.

- Signalisiert werden Interrupts an einem definierten Prozessoranschluß. Um auch Pulse als Unterbrechungssignale zuzulassen, wird dieser Anschluß mit einem Flipflop gepuffer, dem **Interrupt-Request-FF**. Dieses Flipflop wird nach der Ausführung eines jeden Maschinenbefehls getestet und gegebenenfalls eine Interruptroutine aufgerufen.

Für jede Ausnahme gibt es eine eigene Routine. Die Anfangsadresse ergibt sich häufig über eine Sprungtabelle (indiziert mit der Nummer der Ausnahme). Vor Ansprung der Routine wird der Prozessorkontext auf dem Prozessorkeller gesichert; das ist Statusregister und Befehlszähler. Dann wird wie bei einem Unterprogramm der Befehlszähler überschrieben.

Am Ende der Ausnahmebehandlung wird Statusregister und der Befehlszähler restauriert und damit die Verarbeitung des unterbrochenen Programms fortgesetzt. Wenn es sich um eine Unterbrechung gehandelt hat, dann wird das Ende der Bearbeitung durch ein Signal an einem externen Anschluß des Prozessors signalisiert (INTA = interrupt acknowledge).

7.1.1.2 Veränderung im Mikroprogramm

Das Mikroprogramm für die Befehlsholphase wird mit einer Interruptabfrage (und gegebenenfalls mit Behandlungen von Traps) erweitert:

```
REPEAT
      SAR := BZ;
      SR := SP[SAR];
      << hier könnte ein Trap passieren! >>
      BZ := BZ +1;
      BR := SR;
      Befehlsausführung << hier könnte ein Trap passieren! >>
      IF IRQ = 1 THEN
            SP[ST] := BZ;
            ST := ST -1;
            IRQ := 0;
            BZ := Anfangsadresse der Interruptroutine
      END
UNTIL FALSE
```

Im fettgedruckten Teil des Mikroprogramms wird der Befehlszähler gerettet und die Interruptroutine angesprungen. Wichtig ist, daß bereits hier das Interrupt-Request-Flipflop zurückgesetzt wird, weil sonst die Interruptroutine gleich nach dem ersten Befehl selbst wieder unterbrochen würde.

Bei modernen Prozessoren (mit Trace, Ausnahmeebenen sowie System- und Benutzermodus) gibt noch es folgende Spezialitäten:

- Der Statusregisterinhalt wird bei einer Ausnahme zunächst prozessorintern gemerkt und der Prozessor schaltet in den Systemmodus. Dabei werden spezielle Statusbits zurückgesetzt (beispielsweise Einzelschrittbits) und Befehlszähler und gemerktes Statusregister werden auf dem Systemkeller gespeichert.

- Je nach Art der Ausnahme wird die Prioritätsstufe in einem Teil des Statusregisters abgelegt. Nur Ausnahmen einer höheren Prioritätsstufe unterbrechen die aktuelle Ausnahmebearbeitung.

7.1.1.3 Anfangsadresse der Routine zur Ausnahmebehandlung

Es wurde noch nicht beschrieben, wie die Anfangsadressen der Ausnahmeroutinen festgelegt werden. Denkbar sind zwei Ansätze:

10. Aus der internen Nummer der Ausnahme wird hardwaremäßig mit einem Dekoder eine Anfangsadresse erzeugt werden, die vom Dekoderausgang in den BZ übernommen wird.

11. Mit der internen Nummer der Ausnahme wird eine Liste von Adressen indiziert. Diese Liste fängt bei einer durch ein Basisregister gegebenen Adresse an. Eintrag i ist die Anfangsadresse der Routine für Ausnahme i.

    Man nennt eine solche Tabelle *Ausnahmensprungtabelle* (oft auch *Interrupttabelle*).

Die erste Methode verkürzt den Zugriff auf den ersten Befehl der Ausnahmenbehandlung erheblich, die zweite Methode der Adreßzeiger ist erheblich flexibler, weil die Ausnahmeroutinen frei angeordnet werden können und insbesondere lassen sich Ausnahmebehandlungen temporär oder permanent auf andere Routinen umleiten.

Adresse	**Inhalt**
Tabellenanfang+0*wb	Anfangsadresse für $Ausnahme_0$
Tabellenanfang +1*wb	Anfangsadresse für $Ausnahme_1$
...	**...**
Tabellenanfang +n*wb	Anfangsadresse für $Ausnahme_n$
...	**...**
Anfangsadresse für $Ausnahme_j$	1. Befehl für $Ausnahme_j$
...	**...**
Anfangsadresse für $Ausnahme_k$	1. Befehl für $Ausnahme_k$
...	**...**

Normalerweise ist die Ausnahmensprungtabelle in einem Bereich abgelegt, der vom normalen Benutzer nicht geschrieben werden kann (ebenso wie die dadurch erreichten Adreßbereiche...).

7.1.1.4 Anbindung von Unterbrechungsquellen an Prozessor

Ein Prozessor kann unterschiedlich viele Anschlüsse für Interruptinformation vorsehen. Es gibt drei Mechanismen:

12. *Single-Level-Interrupt:* Es gibt nur einen Interruptanschluß. Die Quellen werden über ein Oder-Gatter verbunden und der Prozessor erfragt durch Polling die Quelle.

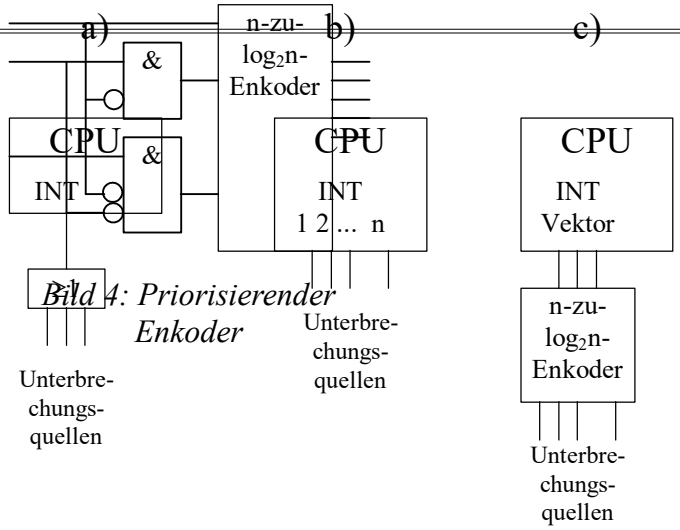


Bild 3: Single-Level- (a), Multilevel- (b), Vektorinterrupt (c)

13. *Multi-Level-Interrupt:* Es gibt für mehrere Quellen jeweils einen eigenen Anschluß. Intern wird ein Interrupt über einen n-zu-log$_2$n-Enkoder auf eine Binärzahl abgebildet.

14. *Vektor-Interrupt:* Es gibt log$_2$n Anschlüsse für Interrupts, wobei diese als Binärzahl interpretiert werden. Wenn die i-te Quelle einen Interrupt auslöst, liegt das Bitmuster der Zahl i an den Anschlüssen an. Man benötigt also extern einen n-zu-log$_2$n-Enkoder.

Die entsprechende Binärzahl beim Multilevel- oder Vektorinterrupt läßt sich direkt als interne Nummer des Interrupts verwenden. Gegebenenfalls muß noch eine Konstante addiert werden, wenn Traps in der internen Numerierung vor den Interrupts kommen.

7.1.1.5 Varianten der Unterbrechungsbehandlung

Im folgenden werden nützliche Varianten der Unterbrechungsbehandlung vorgestellt.

Gleichzeitige Unterbrechungen

Sofern mehrere Unterbrechungsquellen registriert werden können (beim Multilevel- oder Vektorinterrupt) werden sie möglicherweise nicht korrekt behandelt. Wird lediglich ein normaler Enkoder verwendet, liefert er falsche Ergebnisse, wenn mehrere Eingangsleitungen auf 1 liegen.

Dies ist aber prinzipiell zulässig. Daher wird der Enkoder so erweitert, daß er priorisiert. Technisch wird das folgendermaßen realisiert:

Ausnahmen werden nach folgender Reihenfolge priorisiert:

- RESET (Rücksetzsignal);
- Busfehler: bei Ausbleiben der Quittierung beim Lesen im Speicher;
- Einzelschrittmodus (gekennzeichnet durch gesetzte Tracebits im SR);
- Befehlskodefehler;
- Teilen durch 0;

- explizite Traps (benutzt als Schnittstelle zum Betriebssystem):
  Sie haben eine Nummer als Parameter, die als Offset in die Sprungtabelle benutzt wird (d.h. es gibt mehrere Einträge für Traps);
- Interrupts: für Anforderungen externer Geräte:
  Dabei sind langsame Geräte höher priorisiert als schnelle, um einen Ausschluß langsamer Geräte zu vermeiden.

Unterbrechungssperre

Wenn man obiges Mikroprogramm implementieren würde, wäre jedes Programm an jeder beliebigen Stelle unterbrechbar.

Es gibt aber in Systemprogrammen Abschnitte, die nicht unterbrochen werden dürfen (z.B. ein Prozeßwechsel).

Man muß also einen Mechanismus vorsehen, der Unterbrechungen ausschaltet. Hierzu wird ein internes Flipflop INTEA („Interrupt enable Flipflop“) verwendet, das eine *Unterbrechungssperre* realisiert. Dieses Flipflop wird über zwei Maschinenbefehle verändert: EI setzt das Flipflop (enable interrupt) und DI setzt es zurück (disable interrupt).

Im Mikroprogramm wird lediglich die Prüfung nach der Unterbrechungsanforderung verändert auf

IF IRQ = 1 AND INTEA = 1 THEN

Das IRQ-Flipflop wird sofort wirksam, sobald die Unterbrechungssperre gelöscht ist. Unterbrechungen gehen also nicht verloren!

Das Zusammenwirken von Steuerwerk mit den Flipflops IRQ und INTEA sieht prinzipiell folgendermaßen aus:

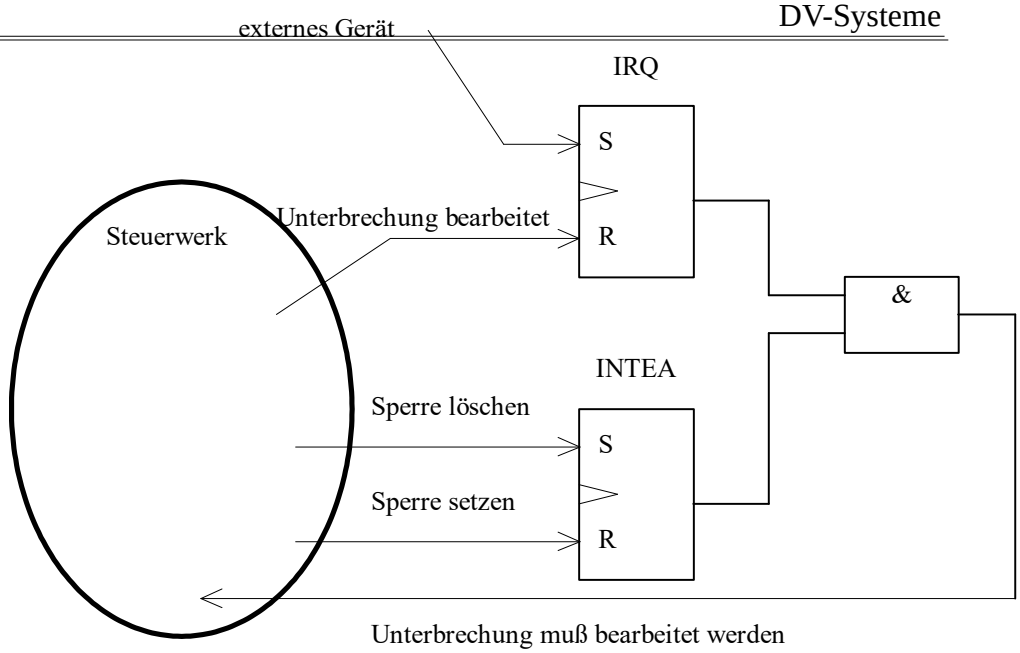


Bild 5: Mechanismus der Unterbrechungssperre

Es ist klar, daß man mit den Befehlen EI, insbesondere aber mit DI ein System aus dem Tritt bringen kann. Daher sind sie auf Systemen mit verschiedenen Ablaufmodi (also Benutzer- und Systemmodus) privilegierte Befehle.

Schachtelung von Unterbrechungen

Eine differenziertere Behandlung von Unterbrechungen als bisher ist möglich: Man kann eine niedrig priorisierte Unterbrechung durch eine höher priorisierte unterbrechen.

Man braucht pro Unterbrechungsquelle zwei Flipflops:

- das **Anforderungs-Flipflop** (AFF) wird von der Quelle asynchron gesetzt; es wird zurückgesetzt, sobald die Anforderung bewilligt ist
- das **Behandlungs-Flipflop** (BFF) zeigt an, daß die Interruptroutine für diese Quelle gestartet wurde, aber noch nicht beendet ist; es wird gesetzt beim Akzeptieren der Unterbrechung und rückgesetzt beim Verlassen der Interruptroutine

IRQ darf jetzt nur dann gesetzt werden, wenn es ein gesetztes AFF(i) gibt, sodaß für jedes gesetzte BFF(j) gilt: i<j.

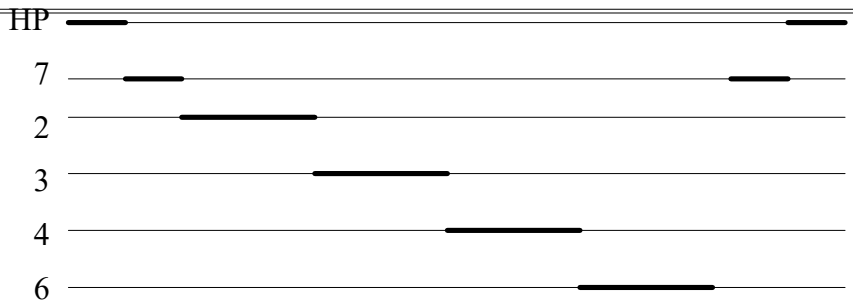
Eine Schaltung dazu ist simpel (Hausaufgabe).

Als Beispiel betrachten wir folgende Belegung:

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
AFF:	0	0	1	1	0	1	0	0
BFF:	0	1	0	0	0	0	1	0

Folgendes gilt:

- nicht akzeptierte Anforderungen: 3, 4, 6



- derzeit in Bearbeitung: 2
- in der Bearbeitung unterbrochen: 7 (von 2)

  Die Interrupts 3, 4 und 6 müssen nach 2 eingetroffen sein, denn sonst hätten sie ihrerseits die Behandlung von 7 unterbrochen (und die entsprechenden BFF's wären gesetzt)

Wenn keine weiteren Unterbrechungsanforderungen mehr eintreffen, erhalten wir folgenden Ablauf der Interruptroutinen

Auffällig ist, daß am Ende von Interrupt 2 nicht der noch offene Interrupt 7 ausgeführt wird, sondern sämtliche höher priorisierten Interrupts abgearbeitet werden.

Maskierung von Interrupts

Neben der totalen Interruptsperre und den hardwaremäßig definierten Interruptprioritäten gibt es bei einigen Systemen noch eine weitere Methode, Interrupts vorübergehend gezielt zu ignorieren.

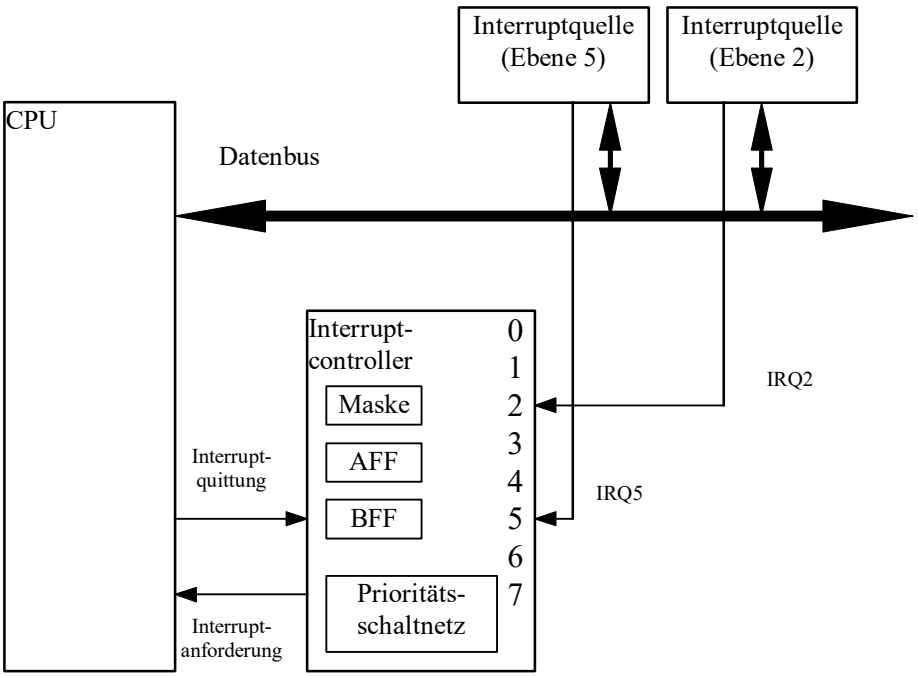
Zu diesem Zweck ist ein Maschinenbefehl „Maskiere Interrupts" vorgesehen, der im Operandenfeld eine sog. **Interruptmaske** definiert. In der Maske ist für jeden Interrupt ein Bit vorgesehen. Durch Setzen des Maskenbits kann ein Interrupt wegmaskiert werden. Er wird solange ignoriert, bis das Maskenbit wieder rückgesetzt wird. Intern wird die aktuelle Interruptmaske in einem Register gespeichert.

Je nachdem, ob die Maskenbits am Eingang der AFF's wirksam werden (dann verhindern sie ein Setzen der AFF's) oder am Ausgang (dann verhindern sie daß der Wert in die Prioritätsschaltung eingeht), bleibt ein maskierter Interrupt stehen, oder er geht verloren.

Interruptcontroller

Gängige Prozessoren bieten nicht alle besprochenen Möglichkeiten an, können aber durch einen speziellen Baustein, einen **Interruptcontroller** mit allen besprochenen Möglichkeiten der Prioritäten und Maskierung ergänzt werden.

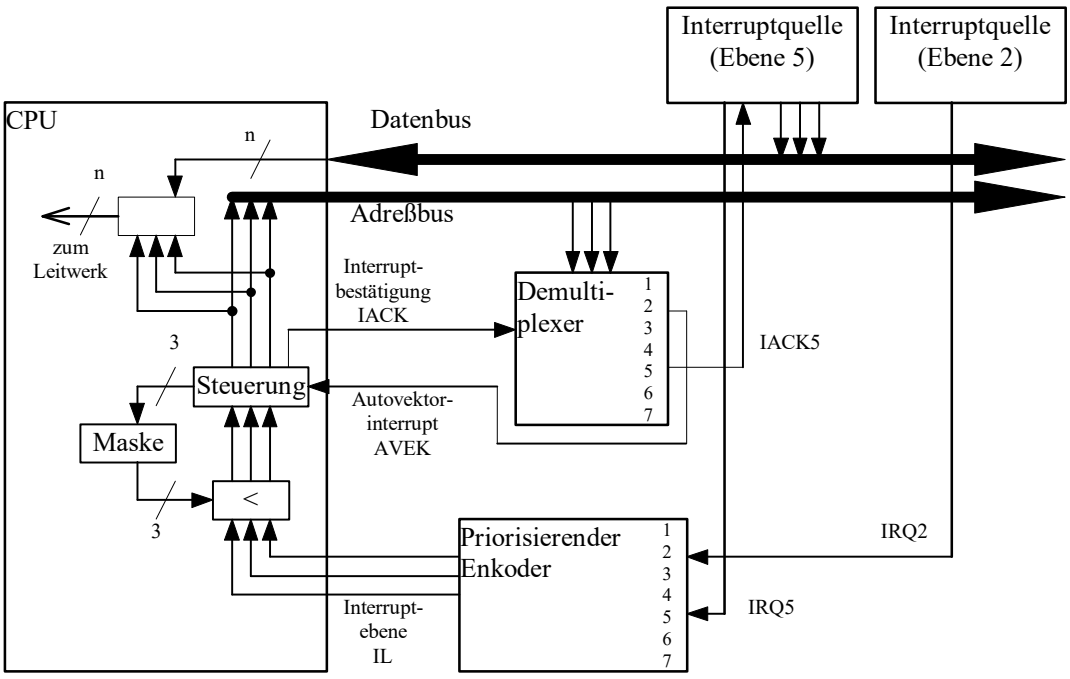
Interruptcontroller sind für eine feste Zahl von Geräten ausgelegt (z.B. 8), können aber kaskadiert werden, um beliebig viele Geräte anzuschließen oder auch mehrstufige Prioritätsschemata zu realisieren.



Bei PCs werden zwei 8bit-Interruptcontroller kaskadiert (durch Anschluß der zweiten Interruptanforderung an einen Eingang des ersten).

7.1.1.6 Beispiel mit kodierten Unterbrechungsanforderungen

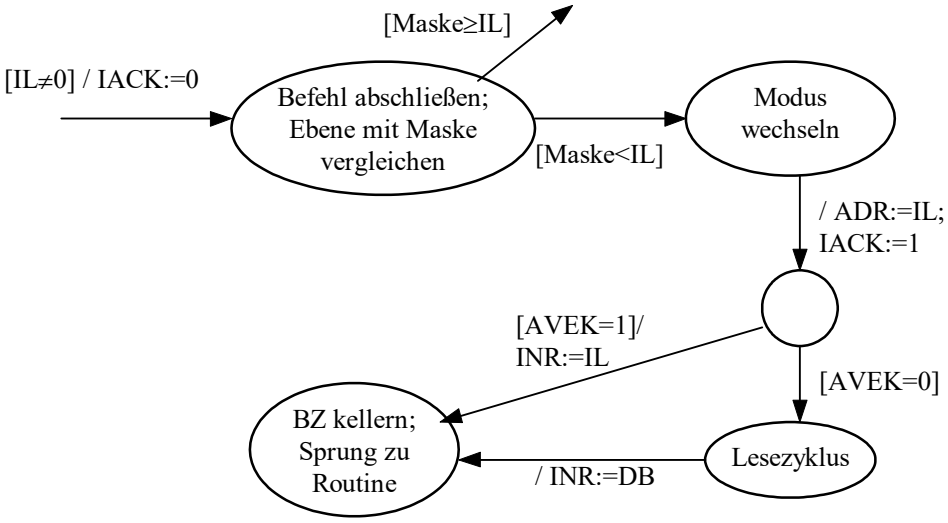
Betrachten wir eine relativ komplexe Schaltung mit zwei Interruptquellen. Der Prozessor hat Eingänge zur Angabe der Interruptebene (Vektorinterrupt) und ein Signal für einen sogenannten Autovektorinterrupt. Diese Leitung besagt, daß die angelegte Interruptebene direkt als Interruptnummer interpretiert werden soll.



Wenn ein Interrupt anliegt (Interruptebene≠0) passiert folgendes:

- Die aktuelle Befehlsausführung wird beendet.
- Die Interruptebene wird mit der aktuellen Interruptebene verglichen. Wenn der Interrupt niedriger priorisiert ist, passiert nichts.
- Ansonsten wird der Interrupt akzeptiert. Die neue Ebene wird in die Maske geschrieben und der Prozessor schaltet in den Systemmodus.
- Der Prozessor legt die Interruptebene auf den Adressbus und quittiert den Interrupt an Leitung „Interruptbestätigung“. Diese Information wird über einen Demultiplexer nach Ebene verteilt. War es ein Autovektorinterrupt (hier bei Ebene 2), wird das direkt an den Eingang „Autovektorinterrupt“ weitergeleitet. Damit wird prozessorintern die Interruptebene direkt als Interruptnummer interpretiert.
- Wenn es kein Autovektorinterrupt ist, wird die Interruptbestätigung an die jeweilige Interruptquelle weitergegeben. Diese gibt auf dem Datenbus die gewünschte Interruptnummer an. Diese wird vom Prozessor gelesen und als Interruptnummer interpretiert.

Noch genauer wird der Ablauf durch einen Zustandsautomaten beschrieben:



7.1.1.7 Ausnahmen auf dem Modellcomputer

Beim MC können durch die Tasten F1 bis F8 acht verschiedene Interrupts ausgelöst werden. An der Speicheradresse 0 liegt ein Sprungteppich, der die Sprünge auf die Interruptroutinen enthält. Die Startadressen der Interruptroutinen können beliebig gewählt werden.

Der Programmierer ist selbst dafür verantwortlich, daß die Tabelle nicht versehentlich überschrieben wird! Insbesondere ist der Anfang des eigentlichen Programms mit .ENTRY zu markieren. Lesen Sie bitte auch die entsprechenden Informationen im MC-Handbuch nach.

Beim MC müssen die Sprünge auf die Interruptroutinen explizit programmiert werden. Als Startadressen für die Interruptroutinen werden also hardwaremäßig die festen Adressen 0,1,2,...7 verwendet, d.h.beim Akzeptieren eines Interrupts wird

BZ := INT

gesetzt.

Prioritäten von Unterbrechungen sind auf dem Modellcomputer möglich. Beim MC entspricht das Register **IntRequest** den A-Flipflops und das Register **IntProcess** den B-Flipflops. Beide Register sind jeweils 8 Bit breit.

Um Maskierung auf dem Modellcomputer zu ermöglichen, hat der MC ein 8 Bit breites Interruptmaskenregister **IM**. Mit dem Befehl LDIM kann es mit einem beliebigen Wert geladen werden. Dabei entspricht eine gesetzte Stelle (=1) einem aktivierten Interrupt, mit 0 wird ein Interrupt wegmaskiert. Standardmäßige Vorbelegung ist die Null, d.h. , wenn keine Interruptmaske geladen wird, reagiert der MC nicht auf die Interrupttasten. Nur F10 ist immer aktiv: damit kann man jederzeit das laufende Programm unterbrechen und ins Hauptmenü zurückkehren.

Beispielprogramm für den Modellcomputer

Um die Möglichkeiten des Modellcomputers auszuprobieren, wird in der Vorlesung ein Programm diskutiert mit folgenden Eigenschaften:

- Alle Interrupts sind aktiviert.
- Das Hauptprogramm und alle Interrupts rufen ein Unterprogramm auf, das circa jede Sekunde ein Zeichen ausgibt (für das Hauptprogramm 'H' und für jeden Interrupt die Ziffer des Interrupts).
- Interrupt 6 blockiert alle anderen, Interrupt 7 die Interrupts 3 und 4.

```
                .TITLE "Interrupttest"
; Dieses Programm gibt circa jede Sekunde ein Zeichen aus
; für das Hauptprogramm 'H' und für jeden Interrupt die Ziffer des Interrupts
;
; Interrupts sind priorisiert nach Nummer (je niedriger desto höher priorisiert)
; Spezialitäten:
;   o Interrupt 6 blockiert alle anderen
;   o Interrupt 7 blockiert 3 und 4

                .OFFSET 0

; Sprungtabelle für Interrupts

                JMP     INTER0
                JMP     INTER1
                JMP     INTER2
                JMP     INTER3
                JMP     INTER4
                JMP     INTER5
                JMP     INTER6
                JMP     INTER7

;---------------------------------------------------------------------
; putout (ch)
; mehrfache Ausgabe des Zeichens in AC in Sekundenabstand
;
; --- KONSTANTEN
COUNT:          DW      10          ; Anzahl der Ausgaben
WAITVAL: DW     20000       ; Konstante, die Anzahl der leeren
                            ; Schleifendurchläufe angibt, die circa eine
                            ; Sekunde brauchen
; ---
PUTOUT:         PUSHR               ; Register kellern
                PUSHA               ; ch kellern
                LDI     [COUNT]     ; FOR IR := 1 TO Wiederholungszahl DO
FOR_1:          LDA     [WAITVAL]  ;   FOR AC := 1 TO Warteschleifenzahl DO
FOR_2:          DECA
                JNZ     FOR_2       ;   END
                POPA
                PUSHA               ;   putc(ch);
                PUTC                ;
                DECI
                JNZ     FOR_1       ; END
                POPA
                POPR                ; Register von Keller holen
                RET

;---------------------------------------------------------------------
                .ENTRY
MAIN:           LDS     STACKTOP
                LDIM    0FFH        ; alle Interrupts freigegeben
                EI                  ; -- " --
                LDA     'H'
                CALL    PUTOUT      ; putout('H')
                STOP

INTER0:         PUSHA
                LDA     '0'
                CALL    PUTOUT      ; putout('0')
                POPA
                RETI

INTER1:         PUSHA
                LDA     '1'
                CALL    PUTOUT      ; putout('1')
                POPA
                RETI

INTER2:         PUSHA
                LDA     '2'
                CALL    PUTOUT      ; putout('2')
                POPA
                RETI

INTER3:         PUSHA
                LDA     '3'
                CALL    PUTOUT      ; putout('3')
                POPA
                RETI

INTER4:         PUSHA
                LDA     '4'
                CALL    PUTOUT      ; putout('4')
```

```
                POPA
                RETI

INTER5:         PUSHA
                LDA     '5'
                CALL    PUTOUT      ; putout('5')
                POPA
                RETI

INTER6:         PUSHA
                DI
                LDA     '6'
                CALL    PUTOUT      ; putout('6')
                EI
                POPA
                RETI

MASK:           DW      0
INTER7:         PUSHA
                STIM    [MASK]      ; Interruptmaske kellern
                LDA     [MASK]      ; -- " --
                PUSHA               ; -- " --
                LDIM    0E7H        ; Interrupts 3 und 4 blockieren
                LDA     '7'
                CALL    PUTOUT      ; putout('7')
                POPA                ; Interruptmaske restaurieren
                STA     [MASK]      ; -- " --
                LDIM    [MASK]      ; -- " --
                POPA
                RETI

                $=2048
STACKTOP:       DW      0
```