Betriebssysteme

Dibaj Babak

Juni 2003

Zusammenfassung

Dieses Dokument ist eine studentische Mitschrift der Veranstaltung Betriebssysteme 1 (WS 03) bei Prof. Dr. Letschert im Studiengang Informatik an der Fachhochschule Gießen-Friedberg. Es fehlen die letzten drei oder vier Vorlesungen, jedoch ist die Mitschrift bis dahin komplett. Da im Wintersemester 03 viele Donnerstage Feiertage waren, ist die Veranstaltung oft ausgefallen, so dass einige Themenfelder etwas kürzer behandelt wurden. Es existieren sowohl von Prof. Dr. Letschert, als auch von Prof. Dr. Jäger sehr gute Skripte zu der Veranstaltung[3, 4].

- **Bezugsquelle:** Diese Mitschrift wurde mit dem Programm lyx erstellt (Grafiken mit xfig). Alle Dateien zum Dokument (.pdf, .lyx und Abbildungen) werden auf der Homepage von Matthias Ansorg bei seinem Open Tools Projekt zur Verfügung gestellt: http://matthias.ansorgs.de.
- **Lizenz:** Diese studentische Mitschrift ist Public Domain, darf also ohne Einschränkungen oder Quellenangaben für jeden beliebigen Zweck benutzt werden, kommerziell und nichtkommerziell. Jedoch enthält sie keinerlei Garantien für Richtigkeit oder Eignung oder sonst irgendetwas, weder explizit noch implizit. Das Risiko der Nutzung dieser studentischen Mitschrift liegt allein beim Nutzer selbst.
- **Korrekturen**: Fehler und Verbesserungswünsche bitte dem Autor mitteilen: Babak.Dibaj@mni.fh-giessen.de.
- **Dozent:** Herr Prof. Dr. Thomas Letschert

Inhaltsverzeichnis

1 Wiederholung 4

2 Einführung 6

2.1 Was ist ein Betriebssystem? 6
2.2 Arten von Betriebsystemen 6
2.3 Historie 6
2.4 Das Betriebssystem als 2 Kollektionen von Routinen 7
2.5 Unterbrechungen/ Interrupts 8

3 BS - Konzept 8

3.1 Eine erste Skizze 8
3.2 Booten (System hochfahren) 9

4 Programm 10
4.1 Adressbindung . . . 11
4.1.1 Adressbindung zur Bindezeit . . . 11
4.1.2 Adressbindung zur Ladezeit . . . 11
4.1.3 Adressbindung zur Laufzeit . . . 11
4.1.4 Logischer Adressraum eines Programms . . . 12
4.2 Statische/ Dynamische Bindung . . . 12
4.2.1 statisch . . . 12
4.2.2 dynamisch . . . 12
4.3 Bibliotheken erzeugen . . . 13
4.3.1 Erzeugen einer statischen Bibliothek . . . 13
4.3.2 Erzeugen einer dynamischen Bibliothek . . . 13
4.4 Bindelader . . . 15
4.5 Plugin Konzept . . . 15
4.5.1 Systemaufrufe . . . 16
5 Prozess 17
5.1 Kontext . . . 17
5.2 Multiprozessor System . . . 19
5.3 Multitasking System . . . 19
5.4 Kontextwechsel . . . 20
5.5 Prozess Zustände . . . 20
6 BS - Kern 21
6.1 Kernstrukturen . . . 21
7 Prozesse in Unix 21
7.1 Programmstart . . . 22
7.2 fork - Aufruf . . . 22
7.2.1 Rückkehrwert von fork . . . 23
7.3 exec - anderen Programmcode ausführen . . . 24
8 Shell 24
8.1 Komplexe Kommandos . . . 24
8.2 Magische Zeile . . . 25

9 Skriptsprachen **26**
9.1 CGI - Programme 26
9.2 PHP 27
9.3 Javascript 27

10 Arbeiten mit Dateien **28**

11 Filterprogramme **31**
11.1 Umlenkung der Ein- / Ausgabe 31
11.2 Pipe- Operator 32
11.3 Umlenkung der Ein-/ Ausgabe mit dup 33
11.4 Systemcall pipe 34

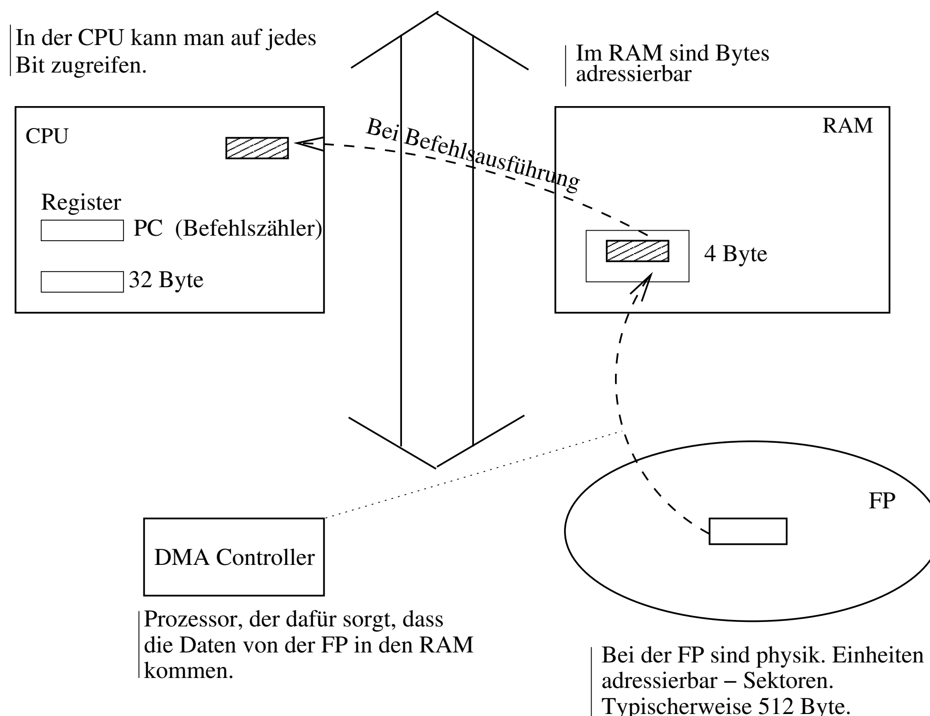
12 Nebenläufige Programme **34**
12.1 Threads 37
12.2 POSIX Threads 37
12.3 Bemerkungen zur dritten Bonusaufgabe 38

1 Wiederholung

Adressierbarkeit und Zugriff auf Speicher und FP

Die CPU kann nicht direkt von der Festplatte lesen, da die CPU die Sektoren der Festplatte nicht adressieren kann - die Einheiten sind zu groß:

Moderne Prozessoren haben 32- oder 64bit Register, diese sind bitweise adressierbar. Da der Hauptspeicher byte-weise adressierbar ist, kann die Cpu von dort Daten in ihre Register laden. Bei der Festplatte sind die physikalisch adressierbaren Einheiten die *Sektoren* - diese sind i.a. 512 Byte groß, also viel zu groß für die Register der Cpu.



DMA - Controller

Damit die Cpu nicht dauerhaft damit beschäftigt ist Daten zeichenweise vom Controller der Festplatte zu holen, wird das DMA- Verfahren eingesetzt. Hierzu braucht das BS Hardware - den *DMA Controller*. Der DMA Controller erledigt die Kopierarbeit von Festplatte in den RAM und signalisiert das Ende per Unterbrechung an die Cpu.

Little- und Big Endian

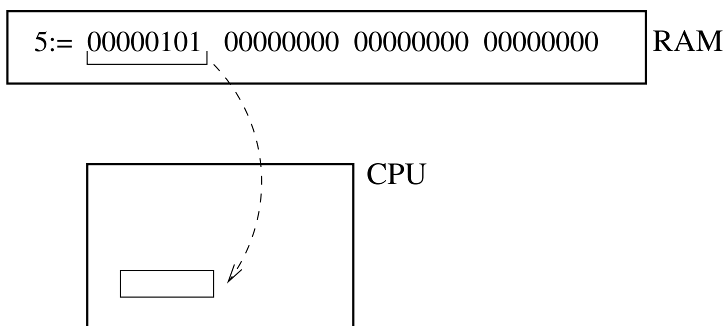
Die Darstellung der Zahl 5 als int sieht in der Bitdarstellung folgendermaßen aus:

Die niederwertigen Bits befinden sich am Ende (*little Endian*)

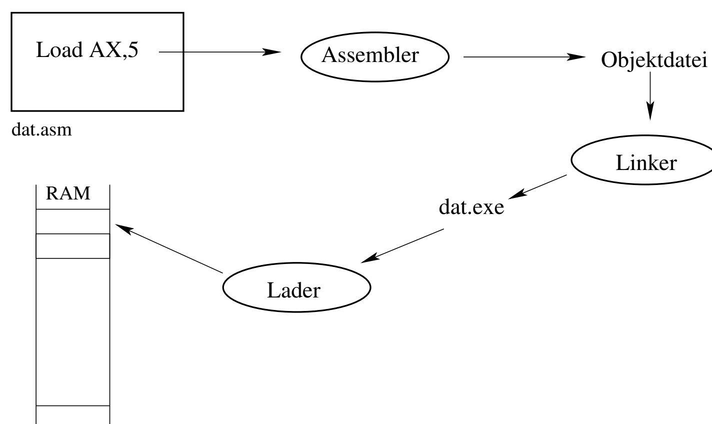
00000000 00000000 00000000 00000101

Die Big Endian Darstellung:

00000101 00000000 00000000 00000000



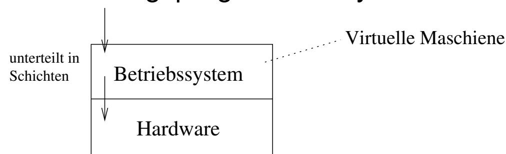
Ladevorgang



2 Einführung

2.1 Was ist ein Betriebssystem?

- Das BS macht die Hardware für 2 Arten von Usern benutzbar:
  - Menschen: Graphische Oberfläche, Konsole[1]
  - Anwendungsprogramme: Systemaufrufe (z.B. von der Platte zu lesen)



- Verwaltet Ressourcen
  - Mittlel, die man braucht (Platte, CPU, RAM, Drucker,...)
- Sonstiges
  - Abrechnungen, Sicherheit,...

2.2 Arten von Betriebsystemen

- Allzweck-Systeme (Windows,...)
- Spezialsysteme
  - Embedded Systems (Maschienen, Handys, Autos,...)
  - Großrechner (Mainframe)

2.3 Historie

Open Shop Die Programmierer hatten Zugang zu riesigen Rechnern und programmierten diese mit Lochkarten direkt.

Closed Shop/ Operatorbetrieb Aufgrund der schlechten Ausnutzung und der enorm hohen Kosten wurden Programmierer und Rechner getrennt. Die Programme wurde bei Operateuren abgegeben.

Batchsysteme Unter Hilfe eines Programmladers und wenigen Systemroutinen sorgten Operatoren durch Einlegen von Bändern für das Weiterlaufen der Programme. Die Arbeit der Operateure wurde teilweise automatisiert.

[1] Die Frage nach der Zugehörigkeit von graphischen Oberflächen zum Betriebsystem ist umstritten.

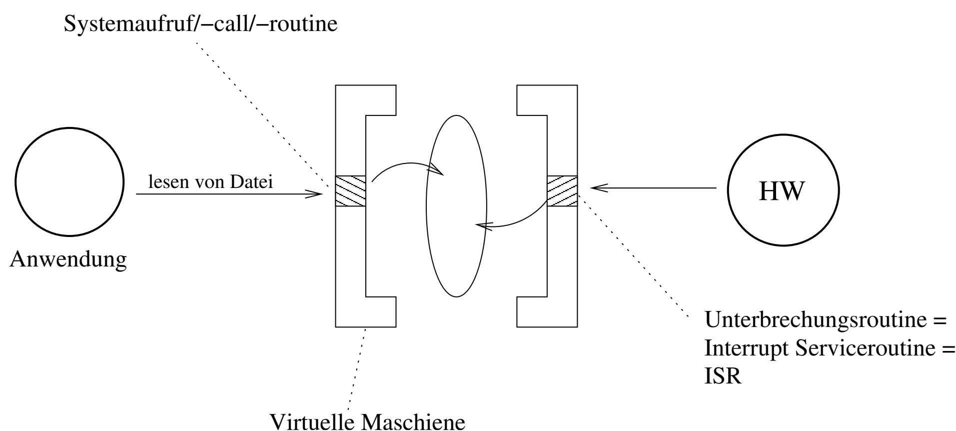
Multiprogramming Eine Idee, wonach die CPU beim Warten des einen Programms auf E/A, in der Zwischenzeit, ein anderes Programm abarbeitet, d.h. viele Programme quasi gleichzeitig.

Die drei beschriebenen Systeme sorgten hauptsächlich für die Optimierung der Verwaltung der Ressourcen. Bei den folgenden Systemen steht der User im Zentrum.

Timesharing Die Benutzer haben nun wieder Zugang zum Rechner und programmieren diese über Terminals. Die Zeit, die der Rechner einem Benutzer zuteilt wird für mehrere Benutzer eingeteilt.⇒ Mehrbenutzerbetrieb, Terminals.
Die Systemzeit wird gleichmäßig zwischen den Anwendungen verteilt, hierzu benötigt das System eine "Uhr", den sog. *Timer.*

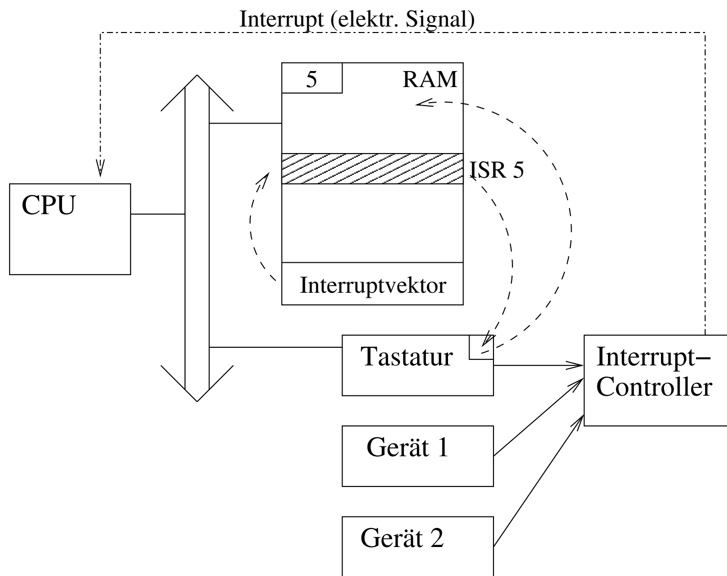
Graphisch-interaktive Systeme Mehrere Anwendungen laufen parallel. Komplexe Befehlseingaben sind nicht mehr nötig. Dem User steht ein Mehrfenstersystem zur Verfügung.
Unterschied zu Timesharing: Bei graphisch- interaktiven Systemen entscheidet der User darüber welche Anwendungen aktiv werden.

2.4 Das Betriebssystem als 2 Kollektionen von Routinen



In dieser vereinfachten Perspektive besteht das Betriebssystem aus 2 Kollektionen von Routinen, die eine ist der Anwedung zugewendet, die andere - der Hardware.

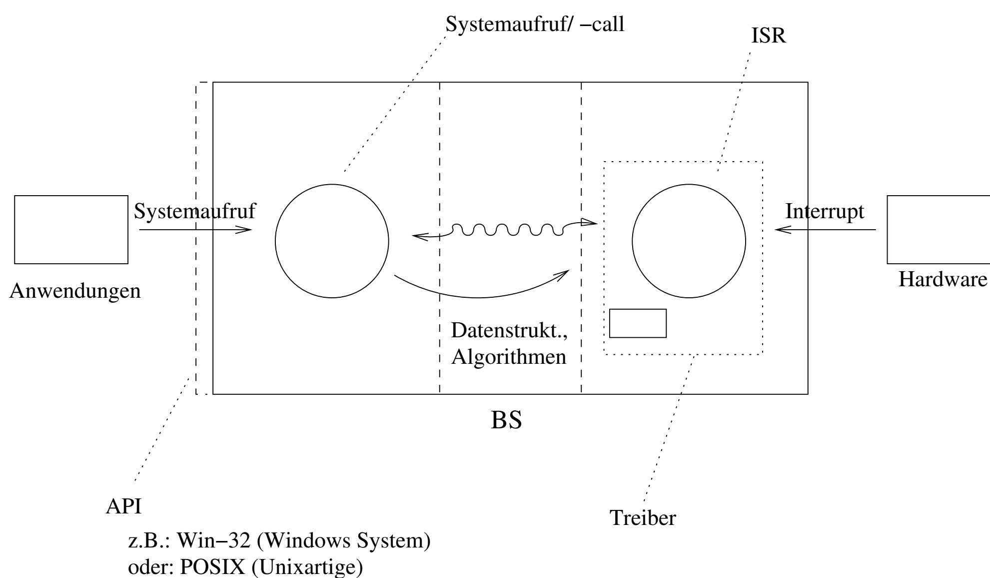
2.5 Unterbrechungen/ Interrupts



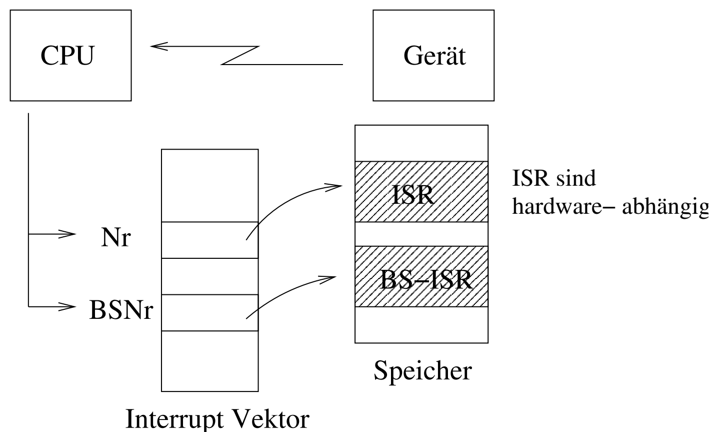
Nach dem Tastendruck auf die Taste 5 wird die CPU vom *Interrupt Controller* auf den Tastendruck durch einen Interrupt (ein elektrisches Signal) aufmerksam gemacht. Nach dem Interrupt 5 schaut die CPU in den *Interruptvektor* und findet dort die Adresse von *ISR-5*. Der Befehl, der dort steht wird in die CPU geladen und ausgeführt. Die *ISR* schaut in den *Tastaturcontroller*, um herauszufinden welche Taste gedrückt wurde, danach wird der ASCII-Wert irgendwo in den RAM geschrieben.

3 BS - Konzept

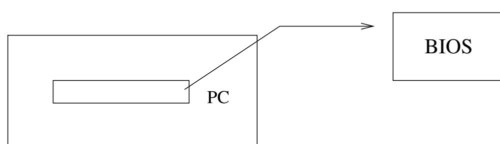
3.1 Eine erste Skizze



Anwendungen nutzen eine API um BS Dienste zu nutzen. Das System stellt eine Menge an Systemaufrufen bereit. Die Hardware signaliesiert das Eintreten bestimmter Ereignisse durch das Auslösen von Interrupts. Die Cpu unterbricht ihre Arbeit und behandelt den Interrupt durch eine *Interrupt Service Routine*.



3.2 Booten (System hochfahren)[2]



Der PC (Program Counter) wird nach dem Einschalten mit dem ersten auszuführenden Befehl geladen - dieser ist festgelegt. Der Code vom *ROM* (Read Only Memory) im BIOS wird ausgeführt - dieser Code mit dem das BS startet heißt BIOS. In der Urzeit der Informatik wurden Rechner und BS von einem Hersteller geliefert. Später, als Rechner und BS von unterschiedlichen Herstellern geliefert wurden - zur Zeit, als das BS auf einer Diskette war, entschloss man sich hardwareabhängige (im BIOS) und hardwareunabhängige Teile des BS aufzuteilen.
Heutzutage reicht das BIOS nicht aus, um die hardwareabhängigen Dinge zu enthalten. Heute gibt es einfache Hardware-check Routinen. Das ROM enthält nicht mehr die hardwareunabhängigen Dinge. **Das ROM enthält nun das Ladeprogramm um das BS von Platte zu laden.**

CPU - Privilegien

Das BS braucht die Fähigkeit der CPU in folgenden Modi zu arbeiten:

- *Benutzermodus*
  - manche Befehle sind nicht erlaubt

[2]boot (enl.): “Stiefel”. Der Begriff “booten” wird abgeleitet von ’bootstrap’: “sich am eigenen Schopf hochziehen”

  - es gibt geschützte Speicherbereiche, die nicht zugreifbar sind.

- *Privilegierter Modus*

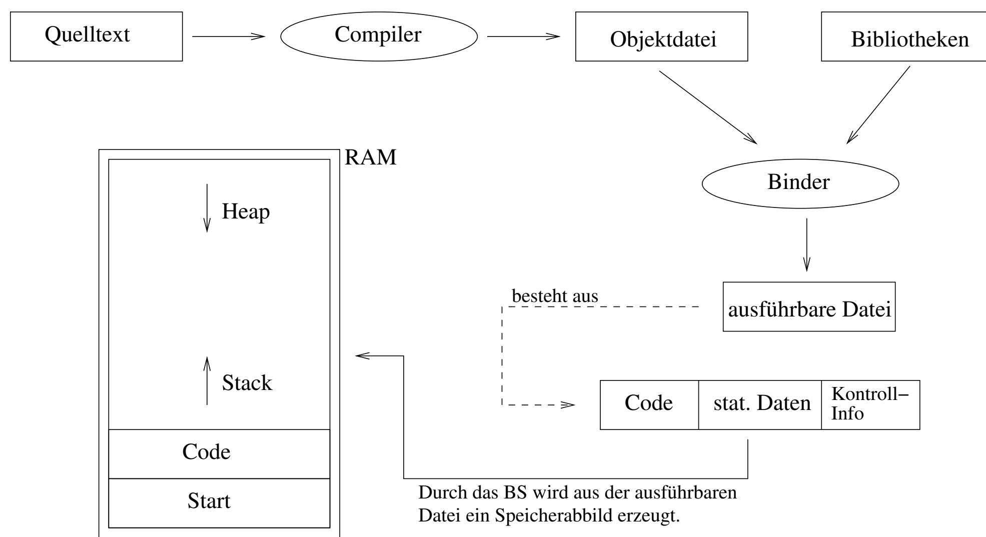
  - Alle Befehle erlaubt.

ISR werden immer im privilegierten Modus ausgeführt, da Interrupts die CPU in den privilegierten Modus schalten.

Systemaufrufe geschehen immer via Interrupt, da man damit automatisch in den privilegierten Modus umschaltet ⇒**Der Einstieg in eine Systemroutine geschieht via ISR.**

Der Interrupt Befehl 'intr' erzeugt beispielsweise einen "Softinterrupt" oder "*Trap*"[3].

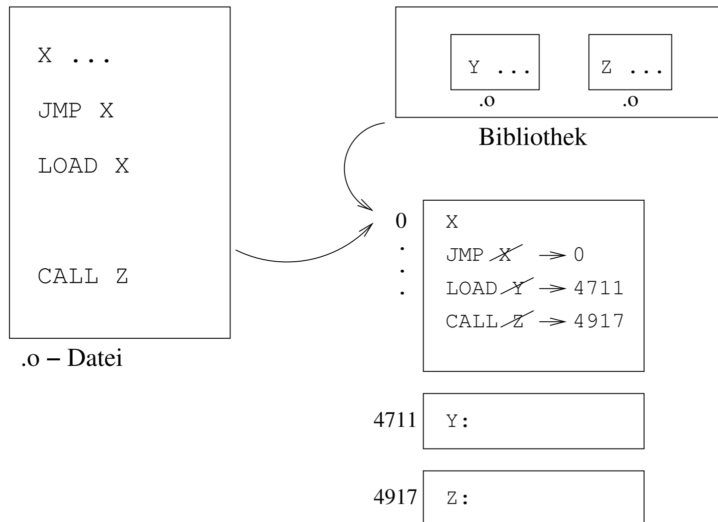
4 Programm



Eine ausführbare Datei ist nicht identisch mit dem Programm im Speicher. Das Programm im Speicher ist generell größer, da es einen Stack und einen Heap hat.

[3]Trap: Ein software-generierter Interrupt, entweder erzeugt durch einen Fehler (Division durch Null, unerlaubter Speicherbereichszugriff) oder explizit durch die Aufforderung einer Anwendung an das BS eine BS-Routine zu starten.

4.1 Adressbindung



Durch die Adressauflösung werden die symbolischen Adressen X, Y, Z in nummerische Werte umgewandelt. Nun liegt die ausführbare Datei (.exe) auf der Platte.

Zur Ausführung der Datei wird sie in den RAM geladen - dies geschieht nicht willkürlich, sondern die Adressen werden umgerechnet.

4.1.1 Adressbindung zur Bindezeit

Logischer Adressraum ⇒ realer Speicher: 1:1 abgebildet, ab einer bestimmten Adresse.

Es handelt sich hier um eine primitive Form der Adressbindung (DOS). Falls beispielsweise in der .exe Datei die nummerische Adresse von 0 beginnt, dann wird diese an die Speicherstelle 0 geladen. Beim .com- Format von DOS werden die Codesequenz und die statischen Daten direkt in den Speicher übernommen.

Großer Nachteil: Das Programm kann nicht auf mehrere Speicherbereiche verteilt werden.

4.1.2 Adressbindung zur Ladezeit

Logischer Adressraum ⇒ realer Speicher: Beliebige Position im Speicher + Zerlegung.

Die symbolischen Adressen (die nummerischen Werte sind ja nach wie vor solche) werden in real existierende physikalische Adressen umgerechnet.

Das Programm kann in mehrere, nicht aufeinanderfolgende Segmente verteilt werden.

4.1.3 Adressbindung zur Laufzeit

Logischer Adressraum ⇒ realer Speicher: Auslagern/ Einlagern von Teilen eines Programms.

Dies ist die heute übliche Form der Adressbindung. Code/ Daten (-Stücke) eines Programms “wandern” im Speicher, dies funktioniert nur mit Hardwareunterstüzung, da sonst die Laufzeit enorm darunter leiden würde.

Codedaten können temporär auf Platte ausgelagert werden. Später können diese irgendwo (anders) wieder eingelaggert werden. Es geht darum Speicher zu sparen und mit einem beschränkten Angebot an Speicher viele Progs gleichzeitig laufen zu lassen.

4.1.4 Logischer Adressraum eines Programms

Der logische Adressraum eines Programms ist üblicherweise zwischnen 2 bis 4GB groß. Dieser ist per Definition zusammenhängend. Moderne BS vergeben mindestens 2GB an logischem Adressraum. In einem nächsten Abbildungsprozess werden durch verschiedene Strategien der logische Adressraum in den realen Speicher abgebildet.

4.2 Statische/ Dynamische Bindung

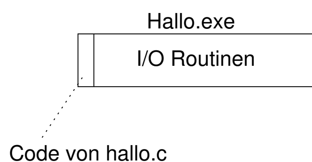
Welche Art von ausführbaren Dateien erzeugt der Binder?

4.2.1 statisch

Eine ausführbare Datei, die vollständig ist - aller Code, den es braucht ist vollständig enthalten. Dazu:

```
gcc -o hallo -static hallo.c
```

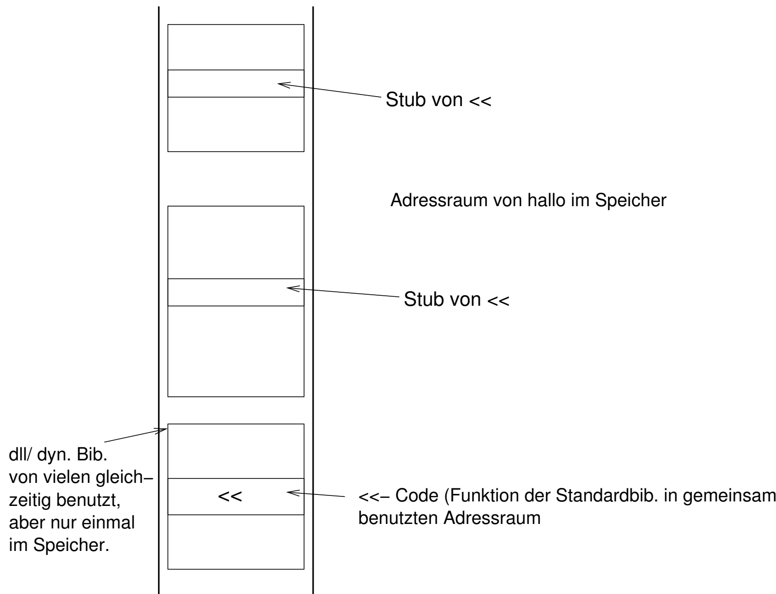
⇒ dies erzeugt eine riesige .exe Datei.



4.2.2 dynamisch

```
gcc -o hallo hallo.c
```

⇒ erzeugt eine zur statisch gebundenen Datei relativ kleine Datei.

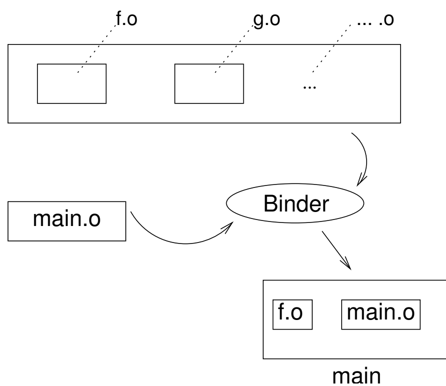


Eine *dll* (Dynamik Link Library) ist auch ein ausführbares Format. Jedoch hat sie im Gegensatz zur .exe viele Einstiegspunkte. Eine dll kann zur Laufzeit gemeinsam von mehreren Programmen genutzt werden.

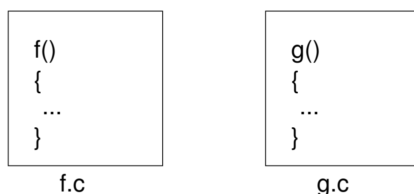
4.3 Bibliotheken erzeugen

4.3.1 Erzeugen einer statischen Bibliothek

- Kollektion (Archiv) von Objekt Dateien. Eine Bibliothek wird mit dem Archivprogramm „ar” erzeugt.

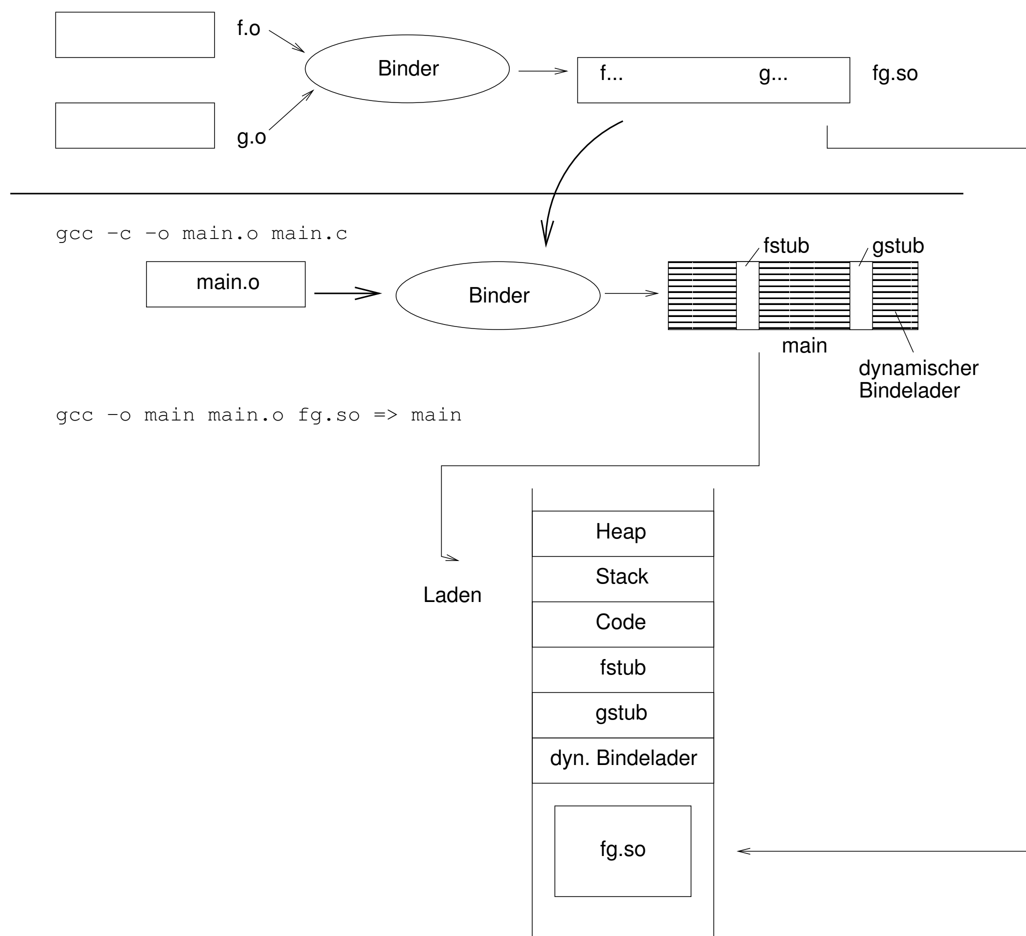


4.3.2 Erzeugen einer dynamischen Bibliothek



```
gcc -c -fpic f.c ⇒ f.o
gcc -c -fpic g.c ⇒ g.o
gcc -shared -fpic -o fg.so f.o g.o
```

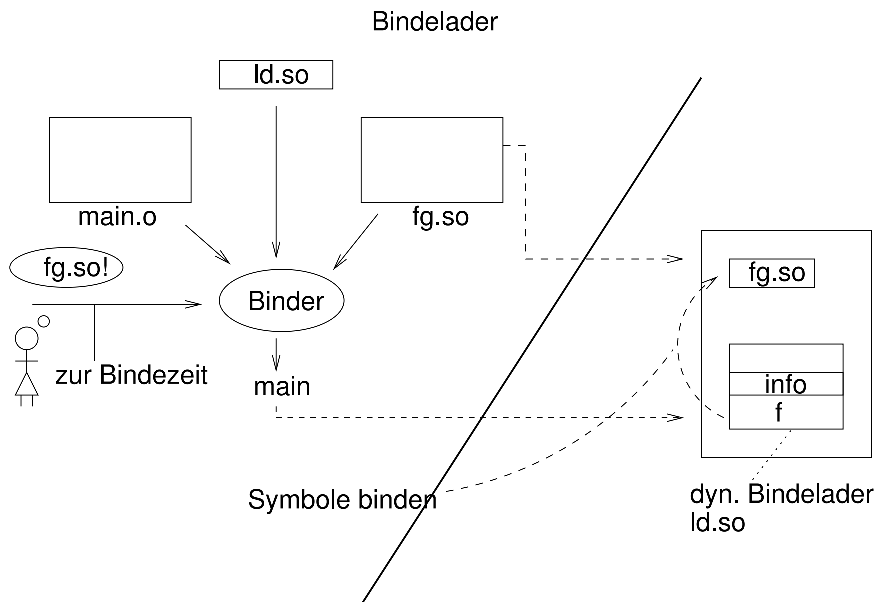
Das „pic” steht für Position Independant Code. „-shared” heißt: erzeuge shared exe (dll).
.so ist die Standardendung für dlls auf Unix. (s.o = shared Objekt).
Im allgemeinen wird User- Code nicht dynamisch gebunden, sondern die Standard-Bibliotheken des Systems.



Nach dem ersten Aufruf von f() oder g() im Code sucht der Bindelader die fg.so auf Platte und lädt diese nach. Falls die fg.so nicht gefunden wird, stürzt das Programm ab. ⇒Standardlösung für Unix: Es gibt einen Bib-Suchpfad: Liste von Verzeichnissen in denen .so- Dateien gesucht werden. Die Pfadinfos sind Bestandteil jedes Programms. Suchpfad definieren:

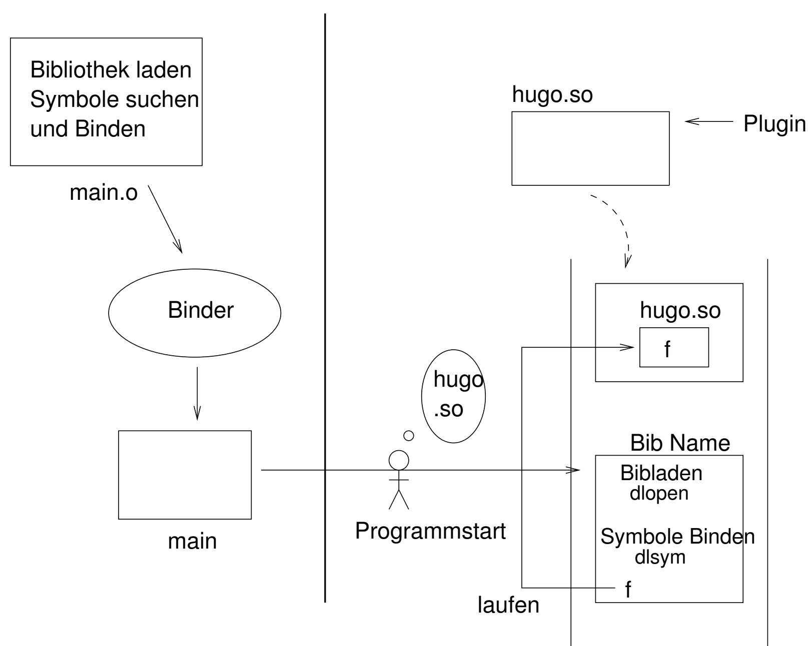
```
export LD_LIBRARY_PATH='LD_LIBRARY_PATH':/home/hg4711/bs1/libexp
```

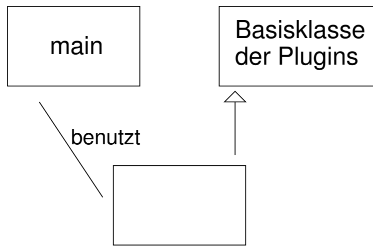
4.4 Bindelader



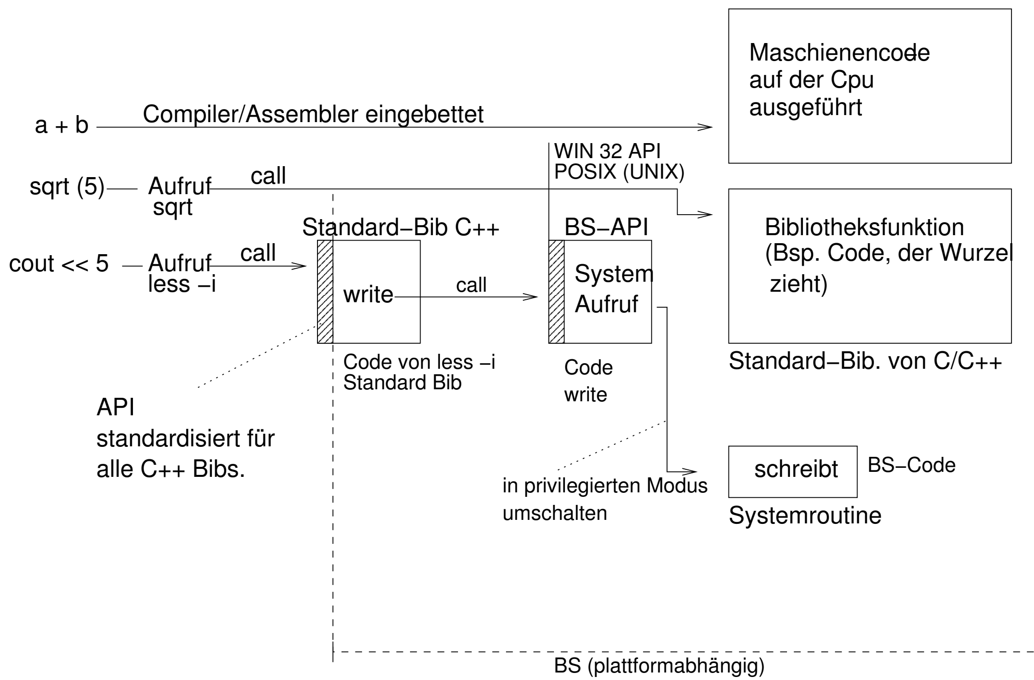
Der dynamische Bindelader wird nach dem Laden eines dynamisch gebundenen Programms aufgerufen, um diese mit schon geladenen oder noch zu ladenden dlls zu binden.

4.5 Plugin Konzept



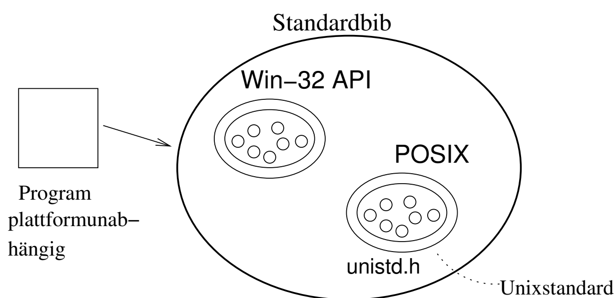


4.5.1 Systemaufrufe



Aufteilung der Schreibfunktionalität auf

- Bibfunktion C++ Standard-Bib (less -i)
- Bibfunktion OS (write)
- Systemcall (via Interrupt)



```
#include <unistd.h>
...
open
read
write
...
```

Das kleinstmögliche Programm (ohne Bibliotheken)

- kann kein C-Programm sein, da die main vom “Startup- Code”, der dazugebunden wird, aufgerufen wird.
- muss mindestens sich selbst beenden ⇒ Systemcall exit. Der Aufruf muss “zu Fuß” programmiert werden - teilweise in Assembler, da ein Interrupt ausgelöst werden muss. Die Parameterübergabe erfolgt via Register (statt Stack wie in C üblich).

5 Prozess

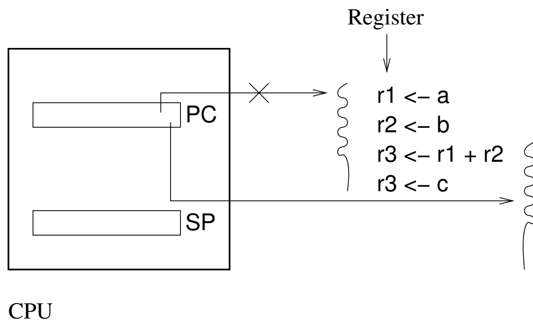
Ein Prozess im allgemeinen ist “etwas das passiert” - eine Abfolge von Aktivitäten. Zu einem Prozess gehört typischerweise ein Prozessor (oder Prozessoren), der den Prozess ausführt. Der Prozessor führt oft mehrere Prozesse aus, es wird zwischen den Aktivitäten hin- und hergeschaltet. Ein Prozessor kann jedoch zur exakt selben Zeit nicht zwei unterschiedliche Dinge berechnen.
Strategie: Ein Prozess wird ”langweilig” (kein Input da,...) ⇒bearbeite einen anderen.

5.1 Kontext

Beim hin- und herwechseln von einem Prozess zum anderen bleibt der Kontext des ersten Prozesses erhalten. Beim Wechsel wird der Kontext aufgehoben.
Unter einem Prozesswechsel versteht man den *Contextswitch*[4] (Kontextswitch).

Ein Prozess ist ein Programm in Ausführung
Hierzu benötigt das Betriebssystem Algorithmen und Datenstrukturen damit die Cpu nach einem Wechsel des Prozesses in dessen Kontext weitermachen kann.

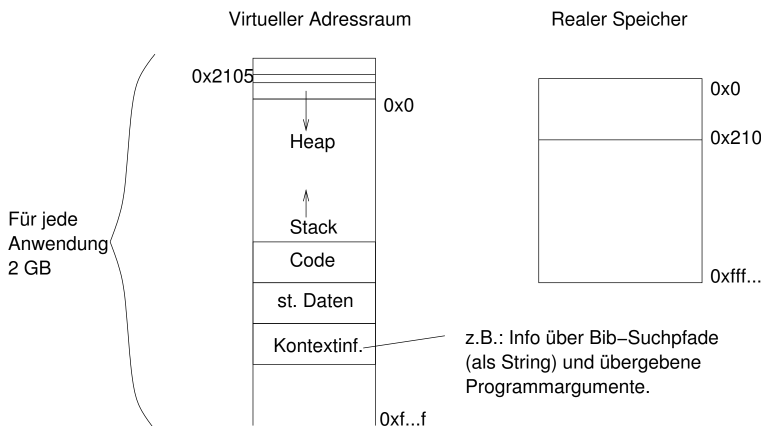
[4]Kontext bedeutet hier u.a. konkret: Der PC (Programm Counter), der auf den Code des alten Prozesses zeigt, wird auf den nächsten auszuführenden Befehl des neuen Prozesses gesetzt. Die Registerwerte, die zum alten Prozess gehören müssen “gerettet” werden.



Informationen über den Kontext Kontext wird vom Prozess Management (BS) verwaltet.

- Adresse nächster Befehl
- Alle benutzten Registerwerte,SP,... werden gespeichert.
- Die Information über den Adressraum wird ebenfalls gespeichert.
- ...

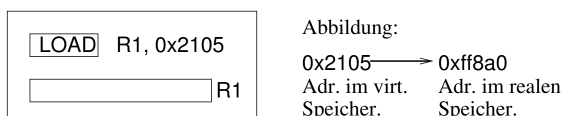
Funktion des Stackpointers: Zu jedem Programm gehört ein Stack - lokale Variablen, Parameterwerte von Funktionen und Rücksprungadressen - eine Funktion, wird nach Beendung einer andern Funktion durch die Rückspr.adresse wieder aktiviert - dazu dient der Stackpointer.



Damit das Konzept des virtuellen Adressraums funktioniert, wobei 2GB für jede Anwendung verteilt wird, muss dieser virtuelle Adressraum in den realen HSP abgebildet werden. Welche Information steckt in dieser Abbildung? -Welche virtuelle Adresse in welche reale Adresse abgebildet wird.

Beispiel:

```
LOAD R1,0x2105 //virtuelle Adresse ⇒0xff890 //reale Adresse
```



Jeder Prozess hat seine eigene Implementierung seines virtuellen Adressraum.

Bem.: Moderne Prozessoren unterstützen das Prozess-Konzept, ansonsten jedoch existiert auf der Prozessorebene kein Prozess.

Einige allgemeine Fragen

Ein Prozess ”wartet”, bedeutet, dass die CPU einen anderen Prozess ausführt. Ein Prozess hört auf zu warten, wenn das Ereignis auf das er wartet eintritt. Die Cpu und dann das Bs bemerkt dieses Eintretten.

Das Bs reaktiviert diesen Prozess bedeutet, dass das Bs die Kontextinfos such, die Cpu wird mit den entsprechenden Registerwerten geladen. Die letzte Aktion des BS ist - den Befehlszähler auf den nächsten Befehl des unterbrochenen Prozesses setzen.

5.2 Multiprozessor System

- Mittlere Systeme

  Typische Werte für die Anzahl der Prozessoren: 2 - 8

- Symmetrische (SMP-Systeme)

  Jeder Prozessor hat im Prinzip die gleiche Aufgabe. 32 - 128 Prozessoren (Große Datenbank-Server). SMP- Systeme sind die heutzutage üblichen Systeme.

- Asymmetrische Systeme

  Unterschiedliche Aufgaben für die Cpu. >128 Prozessoren

5.3 Multitasking System

Task $\sim$ Prozess

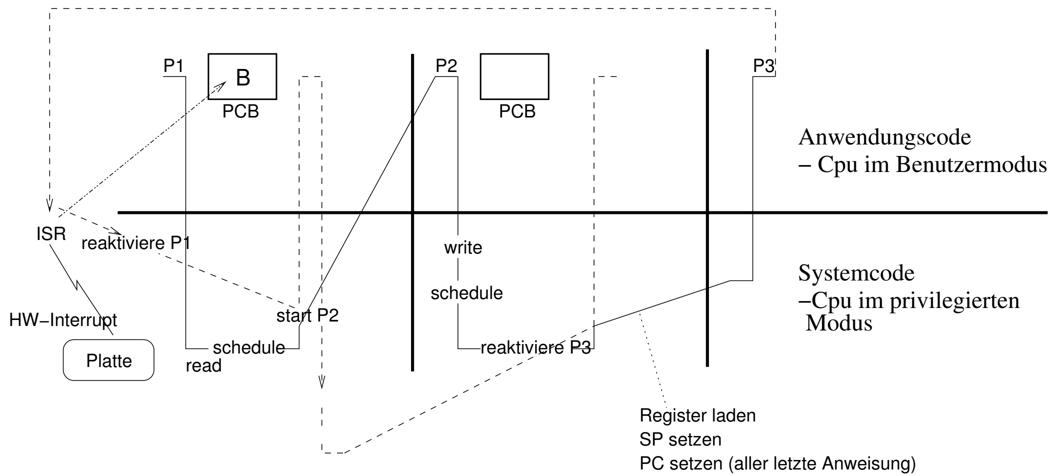
- *Kooperierende MT* (non-preemtive - nicht eingreifend)

  Prozesse müssen Kontrolle freiwillig abgeben.

- *MT* (preemtive - eingreifend) Das System kann den Prozessen die CPU entziehen, indem es in die Prozessorzeit eingreift. Hierbei erzeugt ein *Timer* einen Interrupt und das System prüft welcher Prozess nun die CPU bekommt.

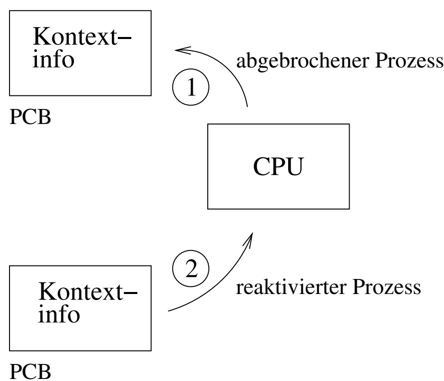
Bem.: Timesharing Systeme sind Spezialisierungen von Multitasking Systemen.

Die Konzepte Prozess/ BS/ Anwendung sind orthogonal[5]



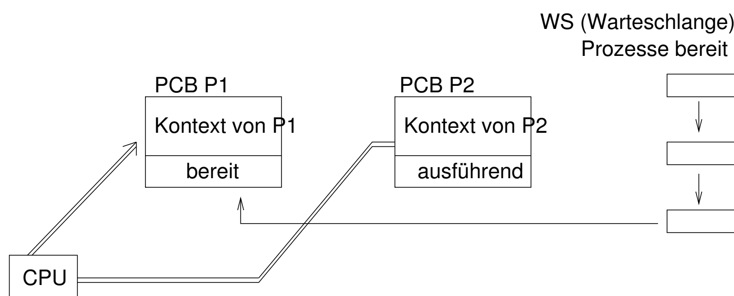
Nach dem Hardware-Interrupt von der Platte, findet ein Zustandsübergang bei P1 statt: Die Information „bereit" (B) wird in den *Prozess Kontroll Block* (PCB) eingetragen.

5.4 Kontextwechsel



(1) ist hardware unterstützt (Interrupt Mechanismus)

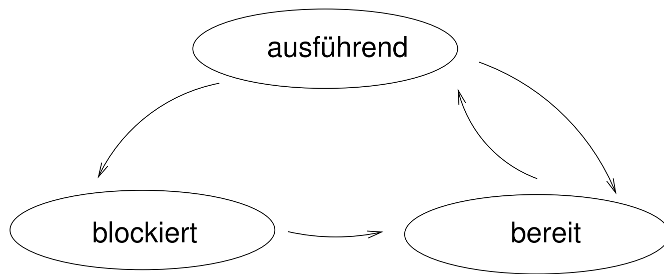
(2) Code des BS



5.5 Prozess Zustände

Bem.: Ein Zustandsübergangsdiagramm ist ein Objektdiagramm.

[5] rechtwinklig, unabhängig



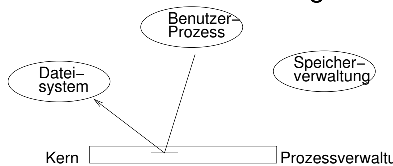
6 BS - Kern

- Innerstes des BS
  - Alles was zum BS gehört aber nicht als Prozess ausgeführt wird.

6.1 Kernstrukturen

- Microkern

  Kern: Prozessverwaltung + Datenaustausch von Prozessen

- Monolithischer Kern

  Kern:

  Enthält die Prozessverwaltung, Speicherverwaltung, Dateisystem,... . Man kommt hierbei mit einem Systemcall an mehrere Funktionen.

- Modularer Kern

  Ein modularer Kern ist auch monolithisch. Die Funktionalität wird hier in ein laufendes BS geladen (als Modul, „BS- Plugin"). Das Konzept entspricht der Plugintechnik. ⇒ Das BS muss einen dynamischen Bindelader enthalten.

  (Dies ist zur Zeit stand der Technik - modulare Kerne)

7 Prozesse in Unix

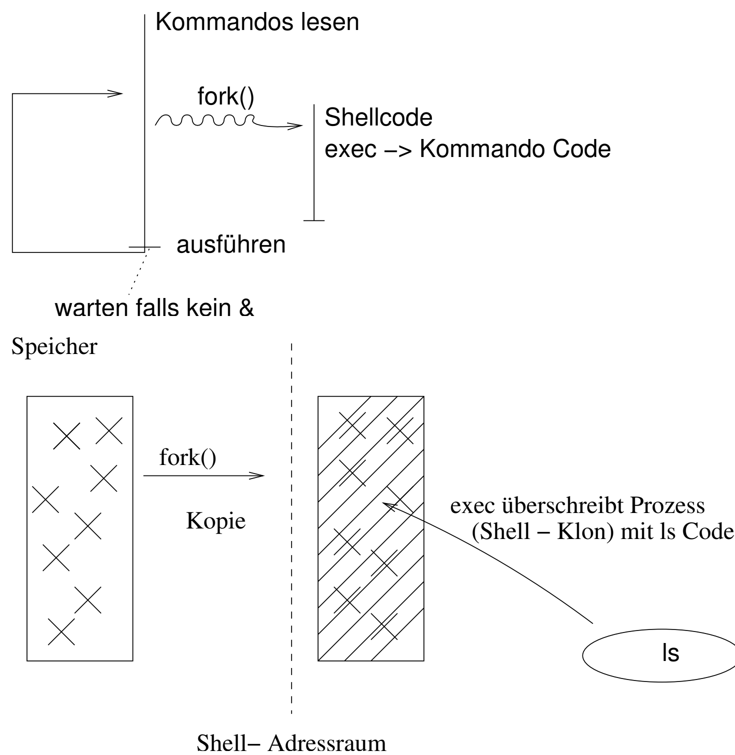
- fork

  erzeugt einen neuen Prozess als Klon.

- exec

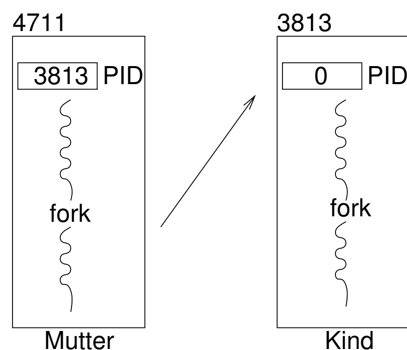
  versorgt einen Prozess mit neuem Code.

7.1 Programmstart



7.2 fork - Aufruf

pid = fork()



Der Erzeugerprozess hat nach dem fork die Adresse des Kindes als PID. Der Kindprozess hat PID 0.

Beispiel:

(1)

#include <unistd.h>

```
int main()
{
  cout <<"Hallo1\n";
  fork();
  cout <<"Hallo2\n";
}
```

Ausgabe:

- Hallo1
- Hallo2
- Hallo2

(2)

```
int main()
{
  int x = 0;
  fork;
  ++x;
  cout << x << endl;
}
```

Ausgabe:

- 1
- 1

7.2.1 Rückkehrwert von fork

```
...
int main()
{
  int pid;
  pid = fork();
  if (pid < 0) {...Fehler...};
  if (pid == 0) {cout <<"Ich bin Kind \n"};
  if (pid > 0) {cout <<"Ich bin Mutter von" << pid << endl;}
  cout << "und wer bin ich?\n";
}
/* moegliche Ausgaben:
---
Ich bin Kind
Ich bin Mutter von 4711
Und wer bin ich?
Und wer bin ich?
---
Ich bin Kind
Ich bin Mutter von 4711
Und wer bin ich?
Und wer bin ich?
---
Ich bin Mutter von 4711
Und wer bin ich?
Ich bin Kind
Und wer bin ich?
Mögliche Ausgaben:
---
```

```
...
jedoch niemals:
Und wer bin ich? ... */
```

Das Programmverhalten ist hier nicht exakt festgelegt - dies nennt man *Nicht-Determinismus*.

"wait" auf Prozessende Die man- Pages dazu anschauen.

7.3 exec - anderen Programmcode ausführen

```
int main()
{
  pid = fork();
  if (pid == 0)  // Kind
  {
    exec ("Programmdatei",...) //Programmdatei soll hier
                                 //"blub" ausgeben
    cout << "hallo";
  }
}
/*
Ausgabe:
blub (und nur blub!) cout<< "hallo"; nur dann, wenn exec schief geht.*/
```

fork + exec ⇒ Neuer Prozess mit neuem Code.

8 Shell

- viele Shells
  - bash (meist loginshell)
  - ...
- kann mehr als Programme ausführen - ist eine Programmiersprache ("Skriptsprache", interpretierte Sprache)

8.1 Komplexe Kommandos

```
for f in *.c do mv $f Verz; done
# f ist die Variable, $f steht für den Inhalt von f
#   Das Kommando verschiebt alle c-Dateien nach Verzeichnis Verz
for ... do ... done
```

Shell Skript = Kommandoprozedur

Man kann die Shell-Kommandos in eine Datei schreiben:

```
# k.sh
cd /home/hg4711
ls -l
rm *.o
```

Das Shell- Skript wird ausgeführt:

```
source k.sh # die Shell führt den Inhalt von k.sh aus
```

oder

```
. k.sh
```

Der . entspricht dem source.

```
bash k.sh          # fork, exec bash mit Input k.sh
                   # k.sh wird von Sub-Shell ausgeführt
chmod u+x k.sh     # k.sh ausfuerbar machen
                   # auch chmod 700 k.sh
./k.sh             # als Kommando ausfueren
                   # dies entspricht dem Aufruf "bash k.sh"
```

8.2 Magische Zeile

```
#! /usr/bin/bash   # chmod u+x k.sh
cd home/hg4711/    # ./k.sh (entspricht /usr/bin/bash k.sh
rm *.o
```

Falls die *Magische Zeile* "#!..." nicht in der Datei enthalten ist, wird es automatisch eingefügt - also automatisch die bash verwendet.

```
#! /usr/bin/perl   # chmod u+x p.pl
 {...PearlProg     # ./p.l (entspricht /usr/bin/perl p.pl)
```

Eigener Interpreter:

```
#!/home/hg4711/nina
 {...NinaProg     # chmod u+x q.nina
                  #./q.nina (entspricht home/hg4711/nina q.nina)
# in /home/hg4711 => nina.c
...
int main (int argc, char * argv[])
{
  ifstream progdat (argv[1]);
  while (not eof argv[1])
  {
    string s;
    progdat >> s;
    interpretiere (s);
}
```

9 Skriptsprachen

Die Shell ist eine Skriptsprache und ist durch einen Interpreter implementiert. Die bash ist eine spezielle Shell.

Bem.: Skriptsprachen werden häufig zur Textverarbeitung eingesetzt (Analyse - Kommandos erkennen).

Andere Skriptsprachen: Perl, php, python, lisp, tcl/tk,...

Einsatz von Skriptsprachen:

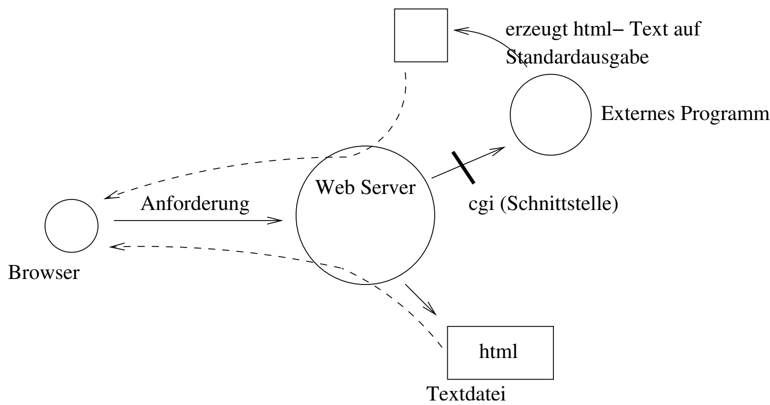
- Internet: cgi- Programme
- Administrative Aufgaben (Systemverwaltung / Verwaltung allgemein)
- generell Aufgaben mit viel Textverarbeitung

9.1 CGI - Programme

CGI (Common Gateway Interface)

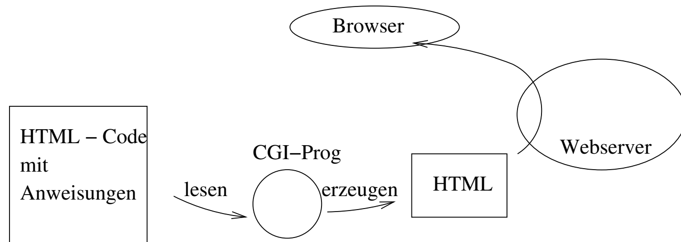
- ist meist in Perl geschrieben, kann jedoch beliebiges ausführbares Programm sein. (z.B.: als C++ Programm, jedoch falls dieses dynamisch gebunden ist tretten Probleme auf)
- Hauptaufgabe: Erzeugen von HTML Seiten.
- Format:

```
Einige Header-Zeilen
Leerzeile
HTML Text in bel. Zeilen
...
```



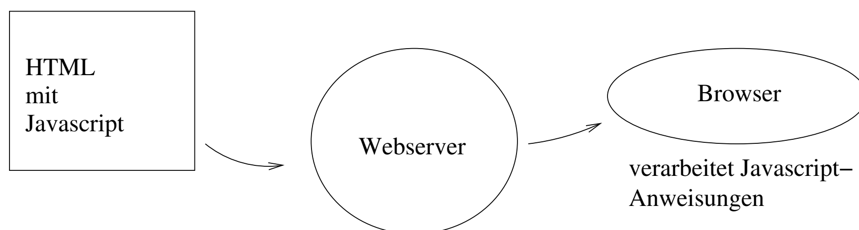
Die Schnittstelle bestimmt das Format der Anforderung das an das externe Programm übergeben wird.

9.2 PHP



Meist ist der php- Interpreter in den Webserver integriert (kann jedoch wie im Bild als CGI-Programm realisiert werden).

9.3 Javascript



10 Arbeiten mit Dateien

```
ifstream dat (”dat.txt”)
```

Hier wird der Konstruktor von ifstream aufgerufen:

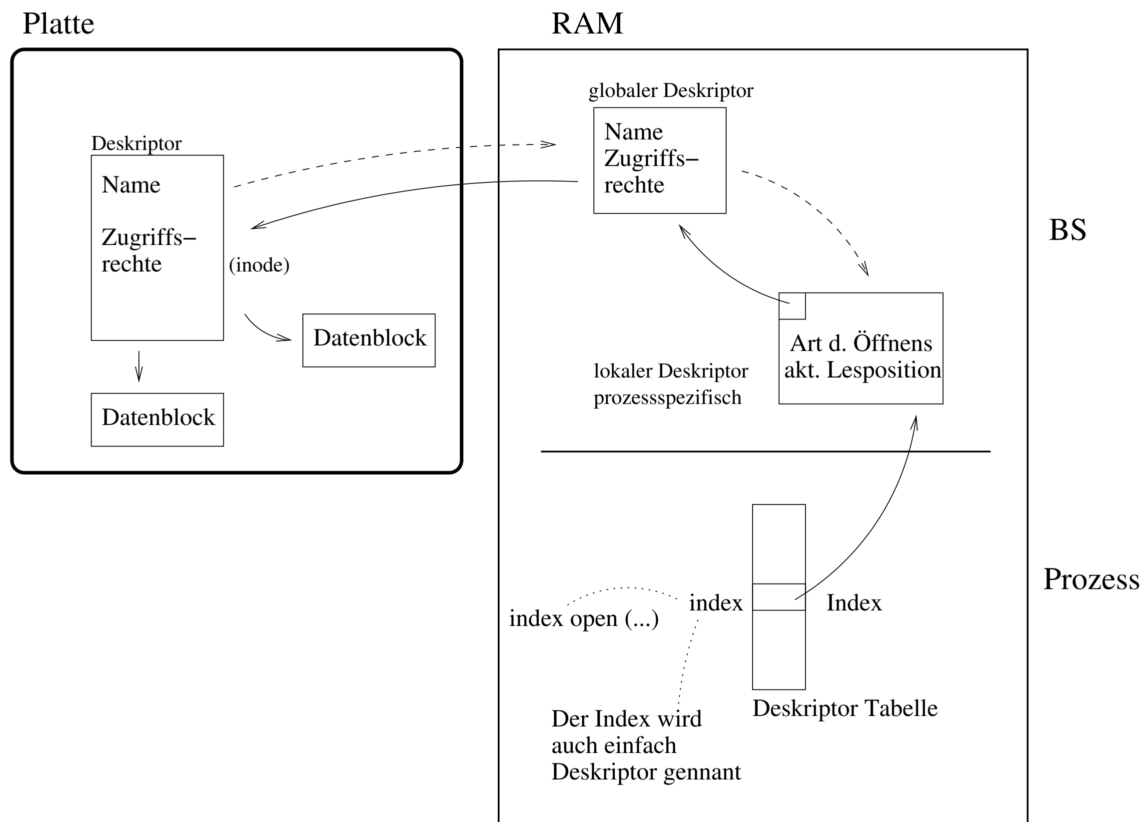
```
ifstream (cstring) {...}
```

Nach dem POSIX Bib.- Systemaufruf von open wird ein Interrupt ausgelöst.

```
open () {...intr...}
```

Durch den Interrupt wird der Systemcall zum Öffnen der Datei aufgerufen. (Adressraum BS)

Die Datei ist identifiziert durch ihren Deskriptor.



Öffnen bedeutet, dass der Deskriptor auf der Platte gesucht wird - falls es einen Deskriptor gibt, werden die darin enthaltenen Informationen in den RAM kopiert.

Der open - Systemcall liefert einen index - dieser wird im Datenfeld von der Variablen ifstream dat gespeichert.

```
class ifstream: public...
{
  ...
  private:
    inf fd; // fd: File Descriptor
}
```

Betrachte:

```
dat << ”Hallo”; // operator << (dat, ”Hallo”)
```

Aufruf:

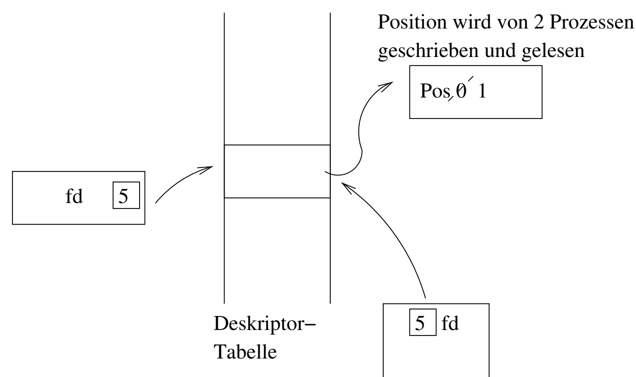
```
write (dat, fd, ”Hallo”);
```

Implementierung:

```
write (){...}
// => POSIX Bib. Systemaufruf => Interrupt => Systemcall - schreiben
```

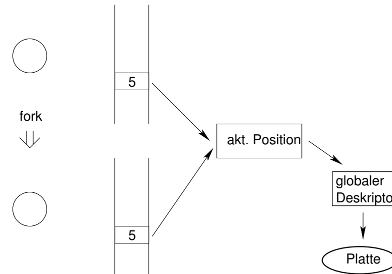
Deskriptortabellen werden vererbt

```
int main()
{
  ofstream dat (”blub.txt”)
  fork();
  dat >> x;
  // erstes Zeichen lesen aus Pos 0 -> 1
}
```



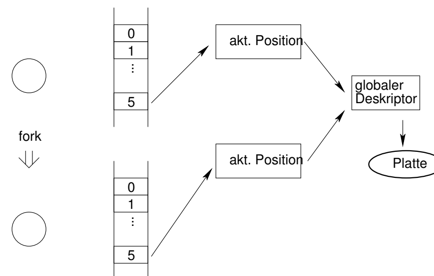

```
// durch fork entstandener Prozess:
int main()
{
  ofstream dat (”blub.txt”);
  fork();
  dat >> x;
  // jetzt wird das zweite Zeichen gelesen!
}
```

```
int main()
{
  ifstream dat ("dat.txt");
  fork();
  while(!dat.eof)
  {
    string s;
    dat >> s;
    cout << s;
  }
}
```



Nach dem Öffnen ist die aktuelle Position = 0. Der erste Prozess liest den ersten String mit Länge 4. ⇒Position = 4. Mutterprozess gibt aus. ⇒ Kindprozess liest nun ab 4 weiter und nicht wieder bei 0. Die Datei dat.txt wird von 2 Prozessen 1mal ausgegeben.

```
int main()
{
ifstream ... ;
// wie oben
}
```



Nun wird die Datei dat.txt von 2 Prozessen 2x ausgegeben, da die aktuelle Positionsangabe sich für die Prozesse unabhängig voneinander bewegt.

Standarddeskriptoren

0 Standardeingabedatei ∼ Terminal

1 Standardausgabedatei ∼ Terminal

2 Standardfehlerdatei ∼ Terminal

In C++:

- cin ∼ Stardeingabe

  `cin >> x;` ⇒ `read (0, &x, length(x));`

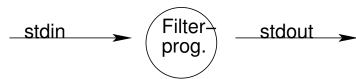
- cout ∼Standardausgabe

  `cout << x;` ⇒ `write (1, &x, length (x));`

- cin ∼ Standardfehlerausgabe

  `cerr << x;` ⇒ `write (2, &x, length (x));`

11 Filterprogramme



* Lesen von Std.- Eingabe
* Schreiben auf Std.- Ausgabe

als C++ Programm:

```
int main()
{
  ...;
  cin x;
  cout << f(x);
}
```

Unix- Tools sind sehr oft Filterprogramme. z.B.:

sort

```
sort
... Text eintippen ...
^d  #Ctrl.-D simuliert das Ende der Std.- Eingabe
```

grep

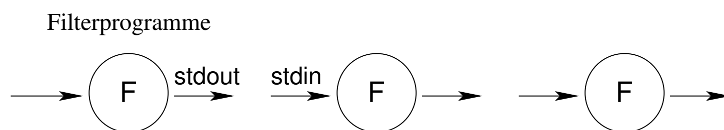
```
grep Hugo
... Text eintippen ...
^d
⇒ Ausgabe aller Zeilen, die "Hugo" enthalten.
```

11.1 Umlenkung der Ein- / Ausgabe

```
sort > dat.txt
... Text eintippen ...
^d
⇒ dat.txt enthaelt nun die Ausgabe/ den sortierten Text

grep Hugo <passwd.txt> hugos.txt
⇒ In der Datei passwd.txt wird nach "Hugo" gesucht,
die Ausgabe in hugos.txt geschrieben.

ls -l *.c > alleCdateien.txt 2> err.txt
⇒ *.c- Dateien aus aktuellem Verzeichnis in alleCdateien.txt
schreiben, falls Fehlermeldungen (keine c-Dateien im Verz.) in
err.txt schreiben.

ls -l *.c > alleCdateien.txt 2>&1
⇒ ... alle Fehlermeldungen werden in die gleiche Datei wie die
Standardausgabe geschrieben.
```

```
grep Charlotte <Emil.txt> Hugo.txt 2>&1
# Das grep hat entweder 1 oder 2 Parameter.

grep Charlotte Emil.txt > Hugo.txt 2>&1
⇒ Emil.txt ist die Datei in der gesucht wird- als zweiten Parameter,
man kann sich das < (wie oben) sparen.

sort <was.txt> wohin.txt # sort ohne Parameter
sort was.txt > wohin.txt # sort mit Parameter
```

11.2 Pipe- Operator



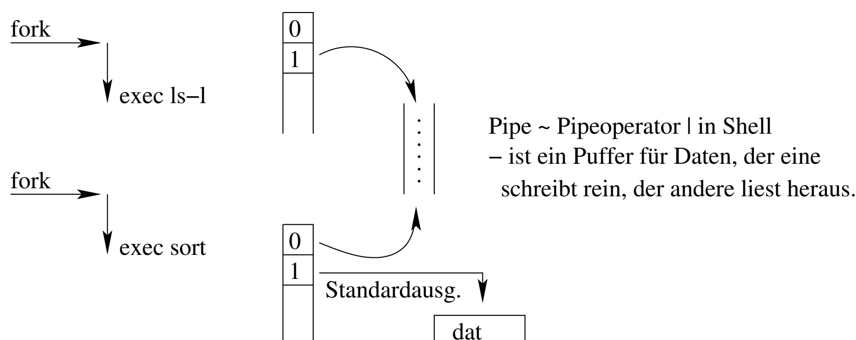
Bsp.:

```
ls -l | grep *.c | sort > Cdat.txt

grep Paramter * | sort >> Fehler.txt
⇒ Nach dem falsch- geschriebenen Wort Paramter suchen und
sortiert mit Dateinamen- Angabe in Fehler.txt schreiben.
```

Konzept

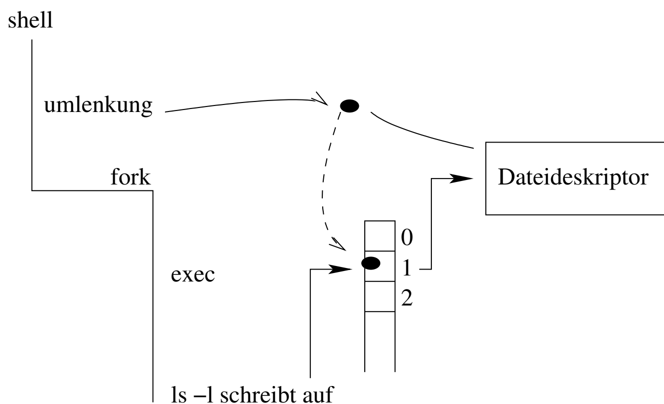
Filterprogramme + Eing.- / Ausgabeumlenkung + Pipe + mächtige Basistools (grep, sort, find, ls, ...) ⇒ Mächtige Kommandos.

Wie arbeitet die Shell Kommandos der Form ”ls -l | sort > dat” ab?



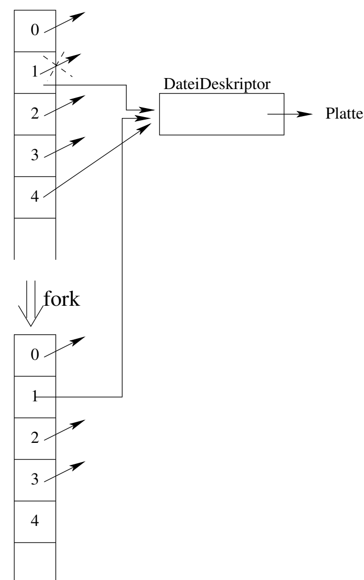
Die Deskriptortabelle wird durch fork geerbt. Das sort weiß von der Umlenkung nichts, es muss *vor* dem fork geschehen. Die ls - Standardausgabe muss auf etwas zeigen, worauf die Standardeingabe von sort zeigt.

```
ls -l > t.txt
```



11.3 Umlenkung der Ein-/ Ausgabe mit dup

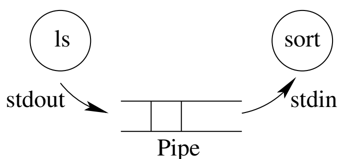
```
fd = open ("t.txt,...)
dup2 (fd,1);
/* Hiernach wird die Std.- Ausgabe (1)
geschlossen und es wird auf
den Dateideskriptor verwiesen.
*/
cout << "Hallo"; // schreibt auf t.txt
...
r = fork();
if (r == 0)
  exec (”ls”);
  // ls schreibt Ausgabe auf t.txt
...
}
```



Pipes

Ein Puffer ist ein Datenstruktur, in die der eine herein-, der andere herauslesen kann.

```
ls -l | sort
```



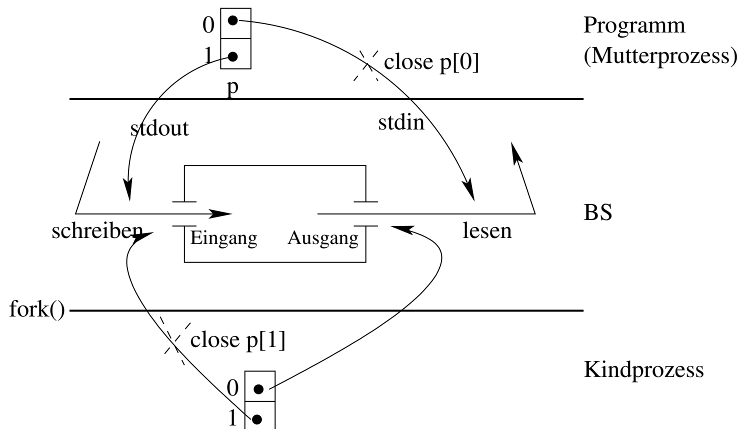
| Pipeoperator

Mit dem | Pipeoperator werden 2 Prozesse über eine Pipe (Datenstruktur des BS) verknüpft.

1. Pipe erzeugen
2. 2x umlenken der Ein-/ Ausgabe

11.4 Systemcall pipe

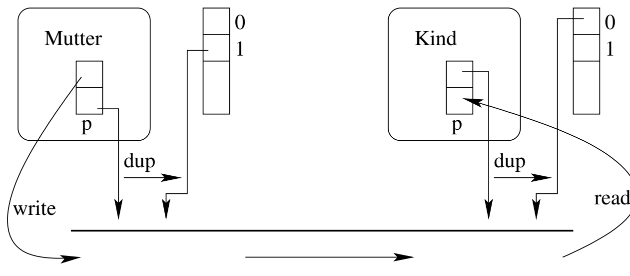
```
int p[2];
pipe (p);
```



Mutter sendet Info an Kind:

Mutter:
close (p[0]);
write (p[1], Daten);

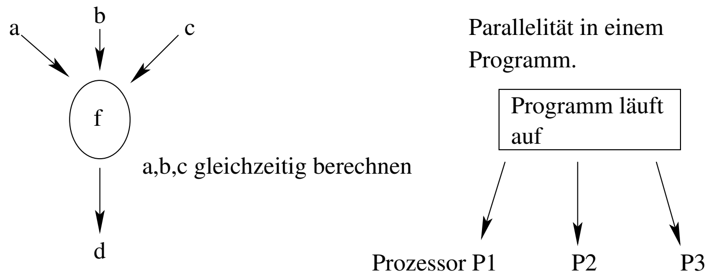
Kind:
close (p[1]);
x = read (p[0]);



12 Nebenläufige Programme

Nebenläufige Programme sind Anwendungsprogramme, die aus mehreren Prozessen bestehen. Die Pipe ist hierbei ein mögliches Kommunikationsmedium der Prozesse.

Wozu nebenläufige Programme?

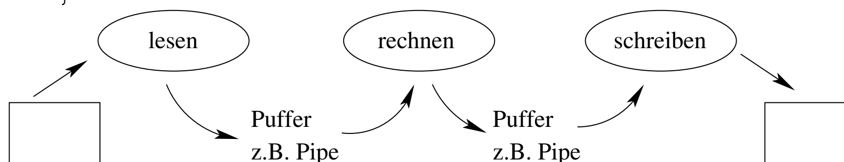


a, b, c können hier gleichzeitig berechnet werden, vorausgesetzt ist eine echte Parallelität.

- Mehrprozessorsystem ausnutzen.
- Durch bessere Ausnutzung Systemdurchsatz erhöhen.

  CPU/ I/O- Hardware Ausnutzung durch gleichzeitige Beschäftigung, also nebelläufige Programme auch in 1 Prozessorsystemen sinnvoll.

```
while (1)

{
  lesen
  rechnen ---> CPU wartet
  ausgeben
}
```

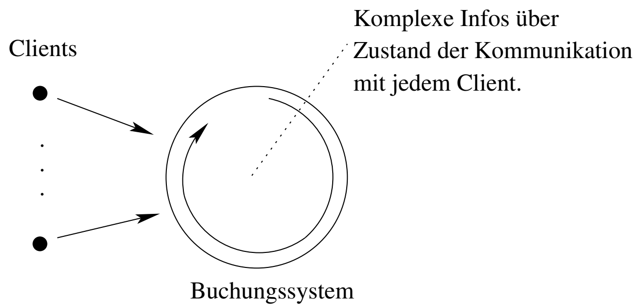



```
Durch die Nutzung von Puffern führt man hier Parallelität auf
der Ebene "meines Programms" ein. ⇒ Die Prozessverwaltung des
BS erhöht Durchsatz meines Programms.
```

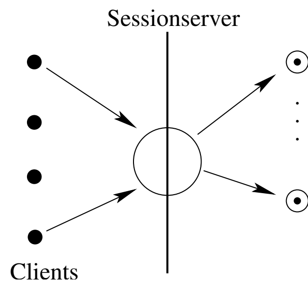
- Nebenläufigkeit macht Programme einfach

  "einfacher": Programmstruktur ist klarer.

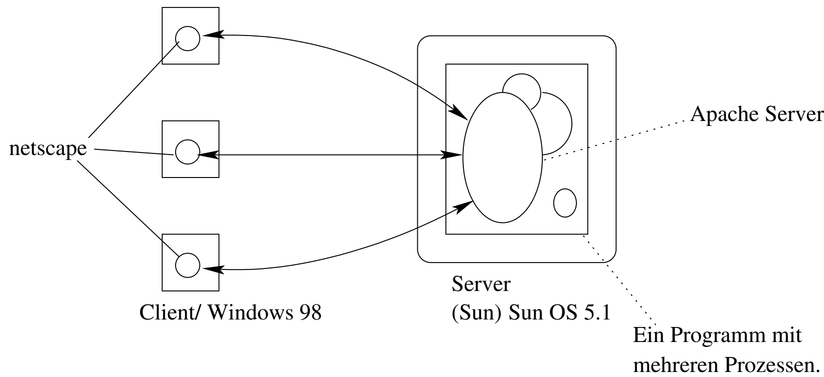
  Beispiel:



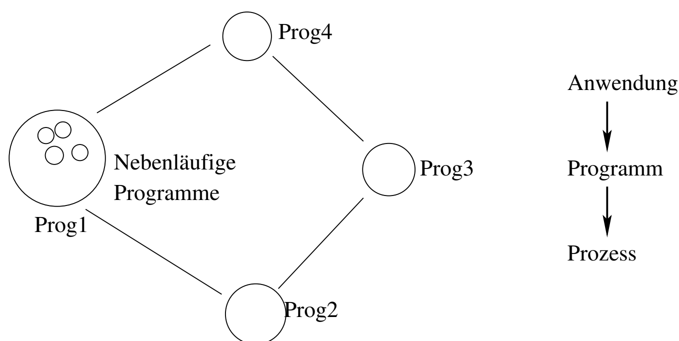
BESSER:



Nebenläufige Programme werden häufig mit verteilten Anwendungen verwechselt. Eine verteilte Anwendung besteht aus mehreren Programmen, die auf unterschiedlichen Instanzen von BS laufen:



Anwendung:



Prozesse sind schwergewichtig. Sie sind zur Abwicklung von Benutzerprogrammen gedacht. Prozesse grenzen die Programme gegenseitig ab, sie haben einen eigenen Adressraum und teilen ihre Variablen nicht mit anderen.

⇒

Problem der Kommunikation untereinander.

- Pipe
- Dateien

Beide Lösungen sind relativ komplex und langsam, aber stellen die einzige Möglichkeit dar, in der Prozesse in Unix miteinander kommunizieren können. Das Problem hierbei ist der langsame Kontextwechsel - langsam vor allem deswegen, da der virtuelle Adressraum wechselt.

⇒

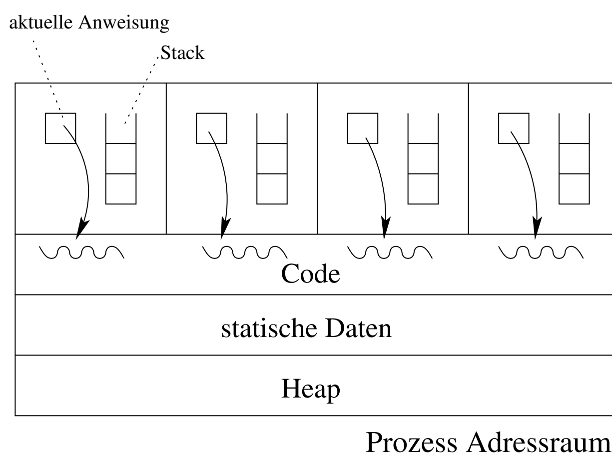
Prozesse zur Realisation nebenläufiger Anwendungen sind zu schwerfällig.

Lösung:

Subprozesse ohne eigenen Adressraum.

12.1 Threads

Threads (ursprünglich "Thread of Control") sind Prozesse ohne eigenen Adressraum, sie haben einen eigenen Handlungsablauf - *Kontrollfaden.* Der Prozess definiert im wesentlichen den Adressraum. Dort gibt es den Heap, stat. Daten, Code und für jeden Thread eine aktuelle Anweisung und einen eigenen Stack.



12.2 POSIX Threads

Posix definiert eine C-Schnittstelle zur Erzeugung von Threads

Thread Erzeugung:

Man muss eine Routine (C-Fkt.) definieren

```
void * routine ()
{
  C-Funktion ist Code des
  neuen Threads.
}
```

und mit

```
pthread_create(tid, routine,...);
```

aktivieren. Nach ”pthread_create” ist die Routine bereit aber nicht zwingend ausführend.

```
pthread_join (tid);
```

entspricht dem ”wait” bei Prozessen und sorgt hier dafür, dass der main Thread solange wartet, bis der Routine Thread ausgeführt ist, d.h. der Routinethread wird garantiert angestoßen, sonst würde main einfach zu Ende gehen. Nach ”pthread_join” wird main blockiert.

12.3 Bemerkungen zur dritten Bonusaufgabe

Sieb des Erathosenes

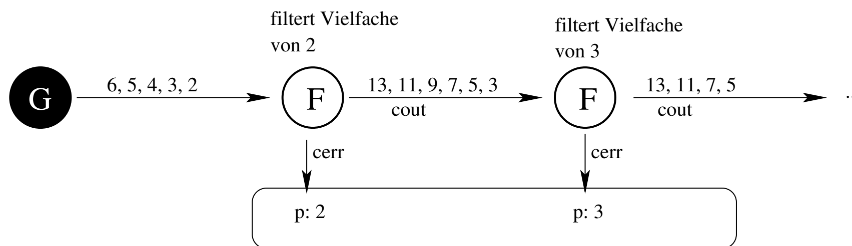
Verfahren:

1. Zahlen notieren
2. 1 zunächst ignorieren
3. Erste Zahl die man findet (= 2) ist Primzahl. Alle Vielfachen der Zahl werden gestrichen.
4. Nächste Zahl ist Primzahl, dann Vielfache streichen usw.

Zunächst hat man einen Prozess, der alle Zahlen beginnen bei 2 generiert.

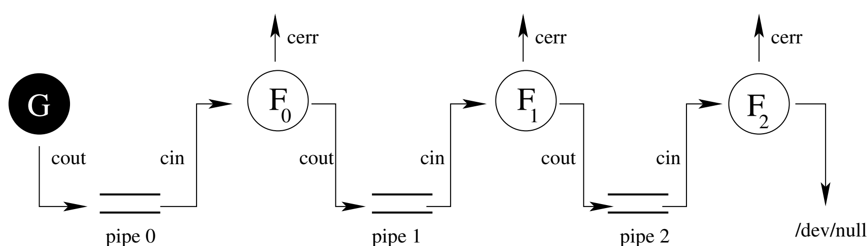
```
// Generator
int i = 2;
while (i < NG)
{
  cout << i << endl;
  ++i;
}
```

```
// Filter
cin >> p; cout <<"primz: " << p << endl;
while (...) // Endlosschleife
{
  cin >> x;
  if (x nicht Vielfach. von p)
    cout << x;
}
```



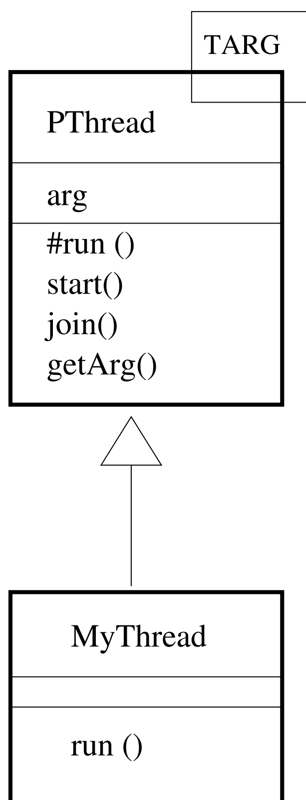
Die gefundenen Primzahlen werden auf die Standard-Fehlerausgabe (cerr) geschrieben. Nicht gefilterte Zahlen werden über die Standardausgabe (cout) an den nächsten Filter gegeben. Pro Filter erhält man eine Primzahl.

```
// NF = Anzahl der Filter = Anzahl der Primzahlen
// NG = bis hierhin erzeugt G Zahlen
// pipes erzeugen => System Call pipe (int [2]]
class Pipe
{
  public:
    Pipe() {pipe (p);}
    int ausgabe (){...}
  private:
    int p [2];
};
Pipe pipe [NF];
/*
Prozess erzeugen mit fork und exec
i in Endlosschleife
if (i == 0)
  exec (G);
else
  exec (F);
```



Das wesentliche bei dieser Aufgabe ist, dass die Prozesse in die richtige Pipe schreiben und von der richtigen Pipe lesen. Sonst ist es nur ein Umlenken der Std.- Eingabe und Std.-Ausgabe. Die Mutter schließt alle Deskriptoren der Pipes. Falls der Generator und die Filter als Threads realisiert werden:

- schneller :-)
- Kommunikation einfach :-)
- Pipe muss/ sollte (wegen Aufgabenstellung) selbst programmiert werden. :-(
  Was macht die Pipe?
  - puffert Daten
  - synchronisiert:
    * leer ⇒ Leser wird blockiert
    * voll ⇒ Schreiber wird blockiert



- TARG ist der Typ des Arguments. (Template)
- Durch den ”fetten Kasten” werden aktive Klassen dargestellt. Ihre Objekte sind aktiv (Sie haben einen eigenen Kontrollfluss)

Index

Adressauflösung, 11
ar, 13
Asymmetrische Systeme, 19

Batchsysteme, 6
Bindelader, 15

CGI, 26
Closed Shop, 6

Deskriptor, 28
Deskriptortabellen, 29
dll, 13
DMA, 4
dup, 33
Dynamische Bindung, 12

Endian, 5
exec, 21, 24

fork, 21, 23

grep, 31

Interrupt Service Routine, 9
Interrupts, 8
Interruptvektor, 8

Javascript, 27

Kern, 21
Kernstrukturen, 21
Kontext, 17
Kontextwechsel, 20
Kontrollfaden, 37

Logischer Adressraum, 12

Magische Zeile, 25
Mittlere Systeme, 19
Modularer Kern, 21
Monolithischer Kern, 21
Multiprogramming, 7
Multitasking, 19

Nebenläufigkeit, 34
non-preemprtive, 19

Open Shop, 6
Operatorbetrieb, 6

PHP, 27
Pipe, 32
pipe, 34
Pipeoperator, 33
Plugin, 15
Position Independent Code, 14
POSIX, 37
Preemptive Multitasking, 19
Prozess, 17
Prozesszustände, 20

Register, 4

Sektoren, 4
Shell, 24
sort, 31
Standarddeskriptoren, 30
Statische Bindung, 12
Symmetrische Systeme, 19

Threads, 37
Timesharing, 7

Umlenkung, 31

Virtueller Adressraum, 18

wait, 24

Literatur

[1] Silberschatz, Galvin, Gagne: *"Applied Operating System Concepts "*

[2] Andrew S. Tanenbaum: "*Moderne Betriebssysteme*" (2. überarbeitete Auflage) Prentice Hall

[3] Prof. Dr. Thomas Letschert: "*Betriebssysteme I*" http://homepages.fh-giessen.de/~hg51/BS-I/

[4] Prof. Dr. Michael Jäger: "*Betriebssysteme I*" http://homepages.fh-giessen.de/~hg52/lv/bs1/