C.1 Dateien

- Kleinste Einheit, in der etwas auf den Hintergrundspeicher geschrieben werden kann.

1 Dateiattribute

- *Name* — Symbolischer Name, vom Benutzer les- und interpretierbar
  - z.B. **`AUTOEXEC.BAT`**
- *Typ* — Für Dateisysteme, die verschiedene Dateitypen unterscheiden
  - z.B. sequentielle Datei, satzorientierte Datei
- *Ortsinformation* — Wo werden die Daten physisch gespeichert?
  - Gerätenummer, Nummern der Plattenblocks

1 Dateiattribute (2)

- *Größe* — Länge der Datei in Größeneinheiten (z.B. Bytes, Blocks, Sätze)
  - steht in engem Zusammenhang mit der Ortsinformation
  - wird zum Prüfen der Dateigrenzen z.B. beim Lesen benötigt
- *Zeitstempel* — Zeit und Datum der Erstellung, letzten Modifikation etc.
  - unterstützt Backup, automatische Dateierzeugung, Benutzerüberwachung etc.
- *Rechte* — Zugriffsrechte bestimmen, wer lesen, schreiben etc. kann
  - z.B. nur für den Eigentümer schreibbar für alle anderen nur lesbar
- *Eigentümer* — Identifikation des Eigentümers
  - Eventuell eng mit den Rechten verknüpft
  - Zuordnung beim Accounting (Abrechnung des Plattenplatzes)

2 Operationen auf Dateien

- Erzeugen *(Create)*
  - ◆ Nötiger Speicherplatz wird angefordert
  - ◆ Katalogeintrag wird erstellt
  - ◆ Initiale Attribute werden gespeichert
- Schreiben *(Write)*
  - ◆ Identifikation der Datei
  - ◆ Daten werden auf Platte transferiert
  - ◆ Eventuelle Anpassung der Attribute, z.B. Länge
- Lesen *(Read)*
  - ◆ Identifikation der Datei
  - ◆ Daten werden von Platte gelesen

2 Operationen auf Dateien (2)

- Positionieren des Schreib-/Lesezeigers *(Seek)*
  - ◆ Identifikation der Datei
  - ◆ In vielen Systemen wird dieser Zeiger implizit bei Schreib- und Leseoperationen positioniert
  - ◆ Ermöglicht explizites Positionieren
- Verkürzen *(Truncate)*
  - ◆ Identifikation der Datei
  - ◆ Ab einer bestimmten Position wird der Inhalt entfernt (evtl. kann nur der Gesamtinhalt gelöscht werden)
  - ◆ Anpassung der betroffenen Attribute
- Löschen *(Delete)*
  - ◆ Identifikation der Datei
  - ◆ Entfernen der Datei aus dem Katalog und Freigabe der Plattenblocks

C.2 Kataloge

- Ein Katalog gruppiert Dateien und evtl. andere Kataloge
  - ◆ Verknüpfung mit der Benennung
    - Katalog enthält Namen und Verweise auf Dateien und andere Kataloge
      z.B. *UNIX, MS-DOS*
  - ◆ Zusätzliche Bedingung
    - Katalog enthält Namen und Verweise auf Dateien, die einer bestimmten Zusatzbedingung gehorchen
      z.B. gleiche Gruppennummer in *CP/M*
      z.B. eigenschaftsorientierte und dynamische Gruppierung in *BeOS-BFS*
- Katalog erlaubt die Benennung von Dateien
  - ◆ Vermittlung zwischen externer und interner Bezeichnung (Dateiname — Plattenblocks)

1 Operationen auf Katalogen

- Auslesen der Einträge *(Read, Read directory)*
  - ◆ Daten des Kataloginhalts werden gelesen und meist eintragsweise zurückgegeben
- Erzeugen und Löschen der Einträge erfolgt implizit mit der zugehörigen Dateioperation
- Erzeugen und Löschen von Katalogen *(Create and Delete Directory)*

2 Attribute von Katalogen

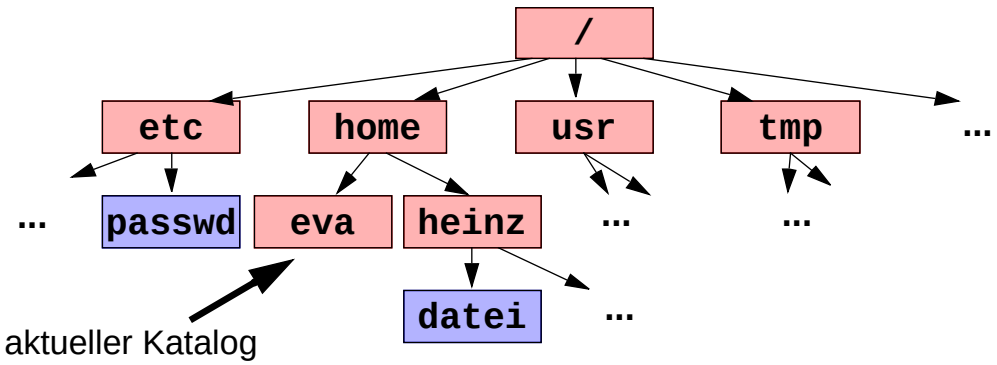
- Die meisten Dateiattribute treffen auch für Kataloge zu
  - ◆ Name, Ortsinformationen, Größe, Zeitstempel, Rechte, Eigentümer

C.3 Beispiel: UNIX (Sun-UFS)

- Datei
  - ◆ einfache, unstrukturierte Folge von Bytes
  - ◆ beliebiger Inhalt; für das Betriebssystem ist der Inhalt transparent
  - ◆ dynamisch erweiterbar
  - ◆ Zugriffsrechte: lesbar, schreibbar, ausführbar
- Katalog
  - ◆ baumförmig strukturiert
    - Knoten des Baums sind Kataloge
    - Blätter des Baums sind Verweise auf Dateien *(Links)*
  - ◆ jedem UNIX Prozeß ist zu jeder Zeit ein aktueller Katalog *(Current working directory)* zugeordnet
  - ◆ Zugriffsrechte: lesbar, schreibbar, durchsuchbar, „nur“ erweiterbar

1 Pfadnamen

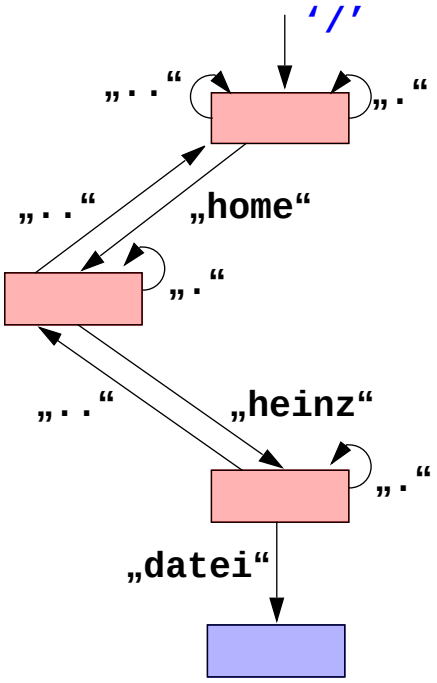
- Baumstruktur



- Pfade
  - ◆ z.B. „**`/home/heinz/datei`**“, „**`/tmp`**“, „**`../heinz/datei`**“
  - ◆ „**`/`**“ ist Trennsymbol *(Slash)*; beginnender „**`/`**“ bezeichnet Wurzelkatalog; sonst Beginn implizit mit dem aktuellem Katalog

1 Pfadnamen (2)

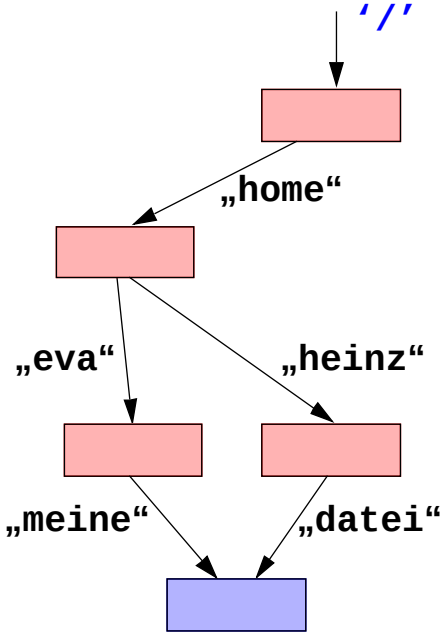
- Eigentliche Baumstruktur



- benannt sind nicht Dateien und Kataloge, sondern die Verbindungen zwischen ihnen
  - Kataloge und Dateien können auf verschiedenen Pfaden erreichbar sein z.B. **`../heinz/datei`** und **`/home/heinz/datei`**
  - Jeder Katalog enthält einen Verweis auf sich selbst („`.`“) und einen Verweis auf den darüberliegenden Katalog im Baum („`..`“)

1 Pfadnamen (3)

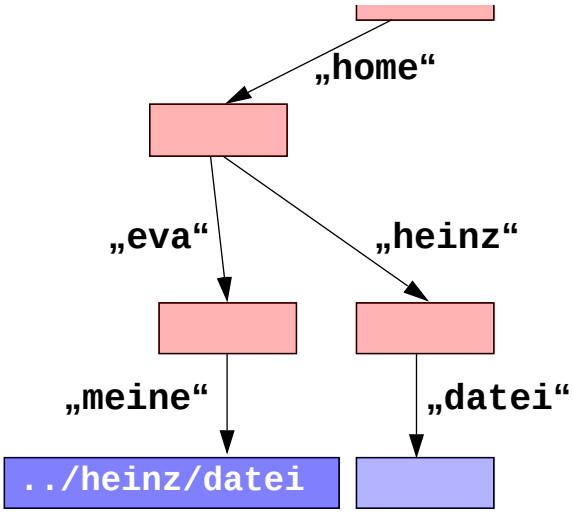
- Links *(Hard links)*
  - Dateien können mehrere auf sich zeigende Verweise besitzen, sogenannte Hard links (nicht jedoch Kataloge)



  - Die Datei hat zwei Einträge in verschiedenen Katalogen, die völlig gleichwertig sind:
    `/home/eva/meine`
    `/home/heinz/datei`
  - Datei wird erst gelöscht, wenn letzter Link gekappt wird.

1 Pfadnamen (4)

- Symbolische Namen *(Symbolic links)*
  - Verweise auf einen anderen Pfadnamen (sowohl auf Dateien als auch Kataloge)
  - Symbolischer Name bleibt auch bestehen, wenn Datei oder Katalog nicht mehr existiert



  - Symbolischer Name enthält einen neuen Pfadnamen, der vom FS interpretiert wird.

2 Eigentümer und Rechte

- Eigentümer
  - Jeder Benutzer wird durch eindeutige Nummer (UID) repräsentiert
  - Ein Benutzer kann einer oder mehreren Benutzergruppen angehören, die durch eine eindeutige Nummer (GID) repräsentiert werden
  - Eine Datei oder ein Katalog ist genau einem Benutzer und einer Gruppe zugeordnet
- Rechte auf Dateien
  - Lesen, Schreiben, Ausführen (nur vom Eigentümer veränderbar)
  - einzeln für den Eigentümer, für Angehörige der Gruppe und für alle anderen einstellbar
- Rechte auf Kataloge
  - Lesen, Schreiben (Löschen und Anlegen von Dateien etc.), Durchsuchen
  - Schreibrecht ist einschränkbar auf eigene Dateien

3 Dateien

Basisoperationen

◆ Öffnen einer Datei

```
int open(const char *path, int oflag, [mode_t mode] );
```

- Rückgabewert ist ein Filedescriptor, mit dem alle weiteren Dateioperationen durchgeführt werden müssen.
- Filedescriptor ist nur prozeßlokal gültig.

◆ Sequentielles Lesen und Schreiben

```
int read( int fd, char *buf, int nbytes );
```

Gibt die Anzahl gelesener Zeichen zurück

```
int write( int fd, char *buf, int nbytes );
```

Gibt die Anzahl geschriebener Zeichen zurück

3 Dateien (2)

Basisoperationen (2)

◆ Schließen der Datei

```
int close( int fd );
```

Fehlermeldungen

◆ Anzeige durch Rückgabe von **`-1`**

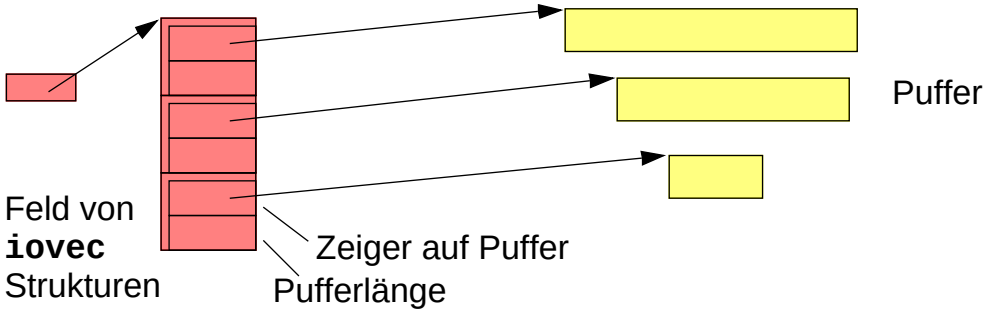
◆ Variable **`errno`** enthält Fehlercode

◆ Funktion **`perror("")`** druckt Fehlermeldung auf die Standard-Ausgabe

3 Dateien (2)

- Weitere Operationen
  - Lesen und Schreiben in Pufferlisten

```
int readv( int fd, struct iovec *iov, int iovcnt );
int writev( int fd, struct iovec *iov, int iovcnt );
```



  - Positionieren des Schreib-, Lesezeigers

```
off_t lseek( int fd, off_t offset, int whence );
```


3 Dateien (3)

- Attribute einstellen
  - Länge

```
int truncate( char *path, off_t length );
int ftruncate( int fd, off_t length );
```

  - Zugriffs- und Modifikationszeiten

```
int utimes( char *path, struct timeval *tvp );
```

  - Implizite Maskierung von Rechten

```
mode_t umask( mode_t mask );
```

  - Eigentümer und Gruppenzugehörigkeit

```
int chown( char *path, uid_t owner, gid_t group );
int lchown( char *path, uid_t owner, gid_t group );
int fchown( int fd, uid_t owner, gid_t group );
```

3 Dateien (4)

◆ Zugriffsrechte

```
int chmod( const char *path, mode_t mode );
int fchmod( int fd, mode_t mode );
```

◆ Alle Attribute abfragen

```
int stat( const char *path, struct stat *buf );
```

Alle Attribute von **path** ermitteln (folgt symbolischen Links)

```
int lstat( const char *path, sturct stat *buf );
```

Wie **stat**, folgt aber symbolischen Links nicht

```
int fstat( int fd, struct stat *buf );
```

Wie **stat**, aber auf offene Datei

4 Kataloge

■ Kataloge verwalten

◆ Erzeugen

```
int mkdir( const char *path, mode_t mode );
```

◆ Löschen

```
int rmdir( const char *path );
```

◆ Hard link erzeugen

```
int link( const char *existing, const char *new );
```

◆ Symbolischen Namen erzeugen

```
int symlink( const char *path, const char *new );
```

◆ Verweis/Datei löschen

```
int unlink( const char *path );
```

4 Kataloge (2)

- Kataloge auslesen
  - Öffnen, Lesen und Schließen wie eine normale Datei
  - Interpretation der gelesenen Zeichen ist jedoch systemabhängig, daher wurde eine systemunabhängige Schnittstelle zum Lesen definiert:

```
int getdents(  int fildes, struct dirent *buf,
               size_t nbyte );
```

  - Zum einfacheren Umgang mit Katalogen gibt es in der Regel Bibliotheksfunktionen:

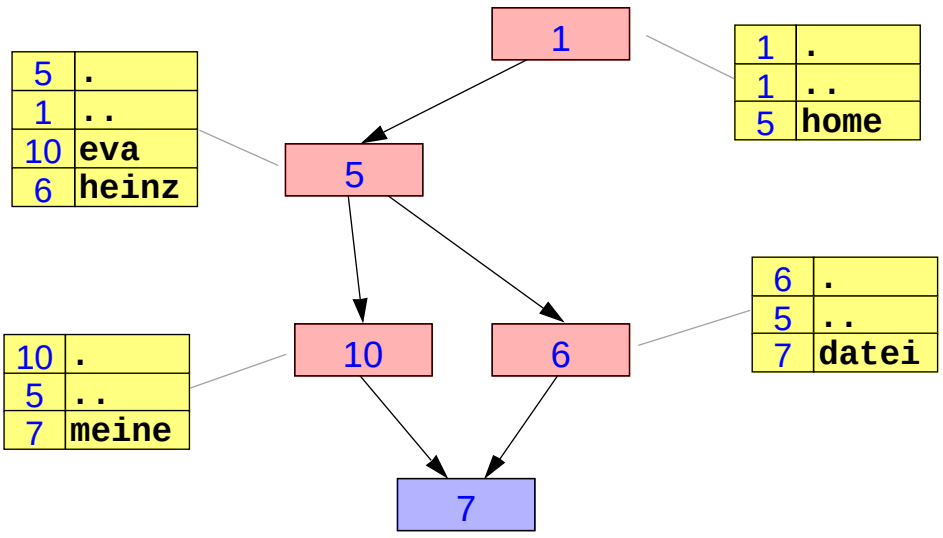
```
DIR *opendir( const char *path );
struct dirent *readdir( DIR *dirp );
int closedir( DIR *dirp );
```

- Symbolische Namen auslesen

```
int readlink( const char *path, void *buf, size_t bufsiz );
```


5 Inodes

- Attribute einer Datei und Ortsinformationen über ihren Inhalt werden in sogenannten Inodes gehalten
  - Inodes werden pro Partition numeriert *(Inode number)*
- Kataloge enthalten lediglich Paare von Namen und Inode-Nummern

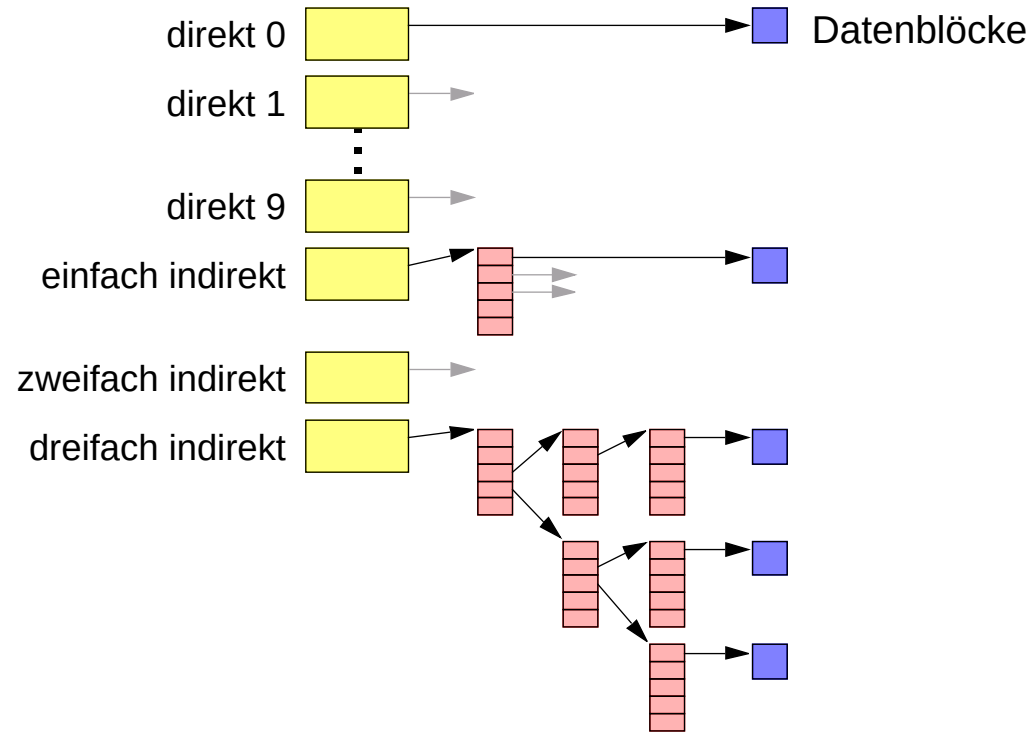


5 Inodes (2)

- Inhalt eines Inodes
  - Inodenummer
  - Dateityp: Katalog, normale Datei, Spezialdatei (z.B. Gerät)
  - Eigentümer und Gruppe
  - Zugriffsrechte
  - Zugriffszeiten: letzte Änderung *(mtime)*, letzer Zugriff *(atime)*, letzte Änderung des Inodes *(ctime)*
  - Anzahl der Hard links auf den Inode
  - Dateigröße (in Bytes)
  - Adressen der Datenblöcke des Datei- oder Kataloginhalts (zehn direkt Adressen und drei indirekte)

5 Inodes (3)

- Adressierung der Datenblöcke



6 Spezialdateien

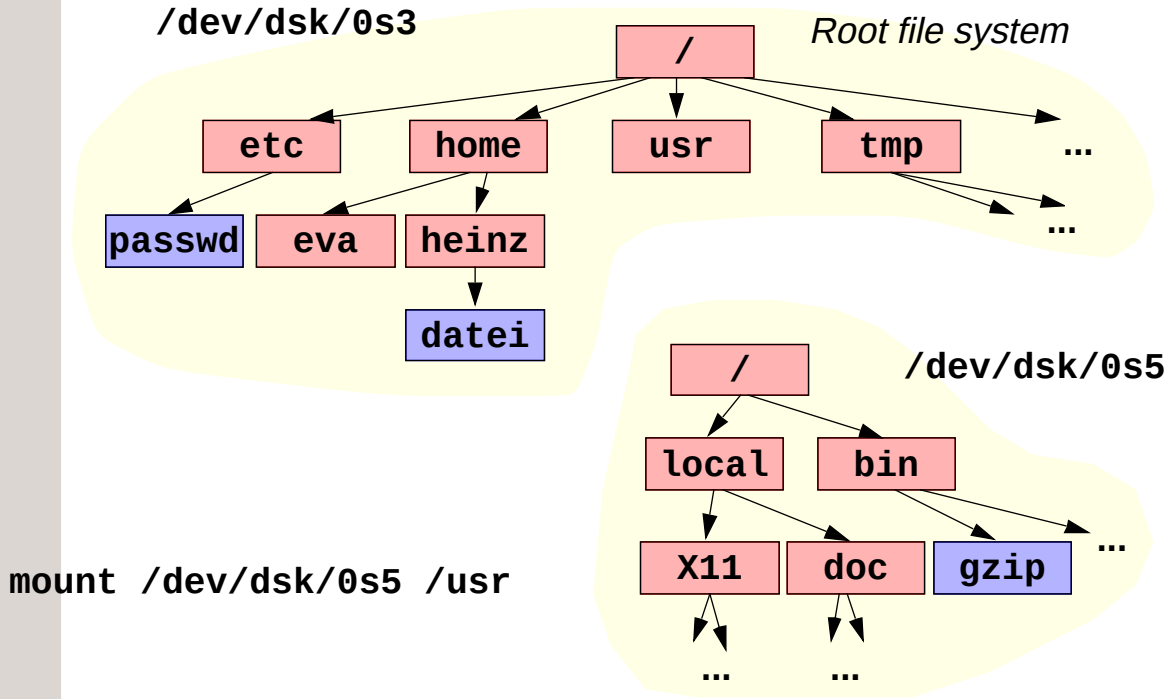
- Periphere Geräte werden als Spezialdateien repräsentiert
  - Geräte können wie Dateien mit Lese- und Schreiboperationen angesprochen werden
  - Öffnen der Spezialdateien schafft eine (evtl. exklusive) Verbindung zum Gerät, die durch einen Treiber hergestellt wird
- Blockorientierte Spezialdateien
  - Plattenlaufwerke, Bandlaufwerke, Floppy Disks, CD-ROMs
- Zeichenorientierte Spezialdateien
  - Serielle Schnittstellen, Drucker, Audiokanäle etc.
  - blockorientierte Geräte haben meist auch eine zusätzliche zeichenorientierte Repräsentation

7 Montieren des Dateibaums

- Der UNIX-Dateibaum kann aus mehreren Partitionen zusammenmontiert werden
  - Partition wird Dateisystem genannt *(File system)*
  - wird durch blockorientierte Spezialdatei repräsentiert (z.B. **`/dev/dsk/0s3`**)
  - Das Montieren wird *Mounten* genannt
  - Ausgezeichnetes Dateisystem ist das *Root file system*, dessen Wurzelkatalog gleichzeitig Wurzelkatalog des Gesamtsystems ist
  - Andere Dateisysteme können mit dem Befehl **`mount`** in das bestehende System hineinmontiert werden

7 Montieren des Dateibaums (2)

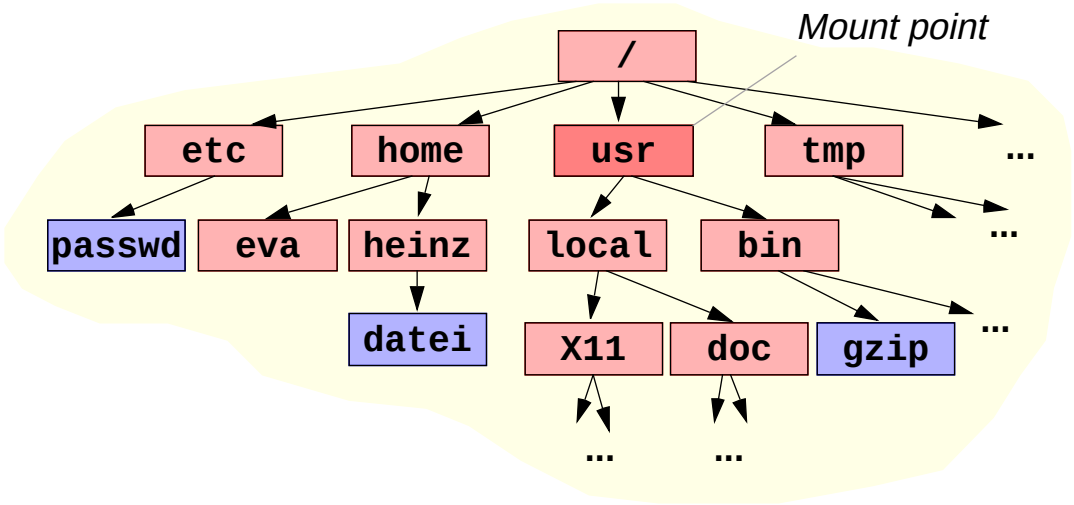
- Beispiel




```
mount /dev/dsk/0s5 /usr
```


7 Montieren des Dateibaums (3)

- Beispiel nach Ausführung des Montierbefehls



C.4 Beispiel: Windows 95 (VFAT)

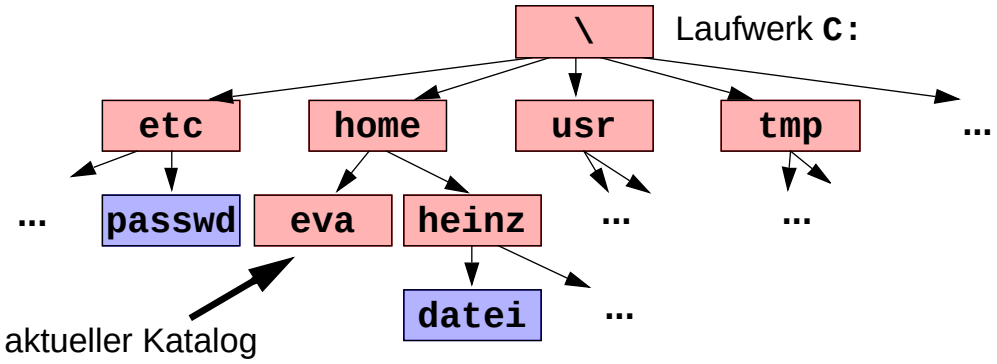
- VFAT = Virtual (?) file allocation table
  - ◆ VFAT: MS-DOS–kompatibles Dateisystem mit Erweiterungen
- Datei
  - ◆ einfache, unstrukturierte Folge von Bytes
  - ◆ beliebiger Inhalt; für das Betriebssystem ist der Inhalt transparent
  - ◆ dynamisch erweiterbar
  - ◆ Zugriffsrechte: lesbar, schreib- und lesebar

C.4 Beispiel: Windows 95 (VFAT) (2)

- Katalog
  - ◆ baumförmig strukturiert
    - Knoten des Baums sind Kataloge
    - Blätter des Baums sind Dateien
  - ◆ jedem Windows Programm ist zu jeder Zeit ein aktuelles Laufwerk und ein aktueller Katalog pro Laufwerk zugeordnet
  - ◆ Zugriffsrechte: lesbar, schreib- und lesbar
- Partitionen heißen Laufwerke
  - ◆ Sie werden durch einen Buchstaben dargestellt (z.B. **`C:`**)

1 Pfadnamen

- Baumstruktur



- Pfade
  - z.B. „**C:\home\heinz\datei**“, „**\tmp**“, „**C:..\heinz\datei**“
  - „\“ ist Trennsymbol *(Backslash)*; beginnender „\“ bezeichnet Wurzelkatalog; sonst Beginn implizit mit dem aktuellem Katalog
  - beginnt der Pfad ohne Laufwerksbuchstabe wird das aktuelle Laufwerk verwendet

1 Pfadnamen (2)

- Namenskonvention
  - Kompatibilitätsmodus: 8 Zeichen Name, 3 Zeichen Erweiterung (z.B. **AUTOEXEC.BAT**)
  - Sonst: 255 Zeichen inklusive Sonderzeichen (z.B. „**Eigene Programme**“)
- Kataloge
  - Jeder Katalog enthält einen Verweis auf sich selbst („**.**“) und einen Verweis auf den darüberliegenden Katalog im Baum („**..**“) (Ausnahme Wurzelkatalog)
  - keine Hard links oder symbolischen Namen

2 Rechte

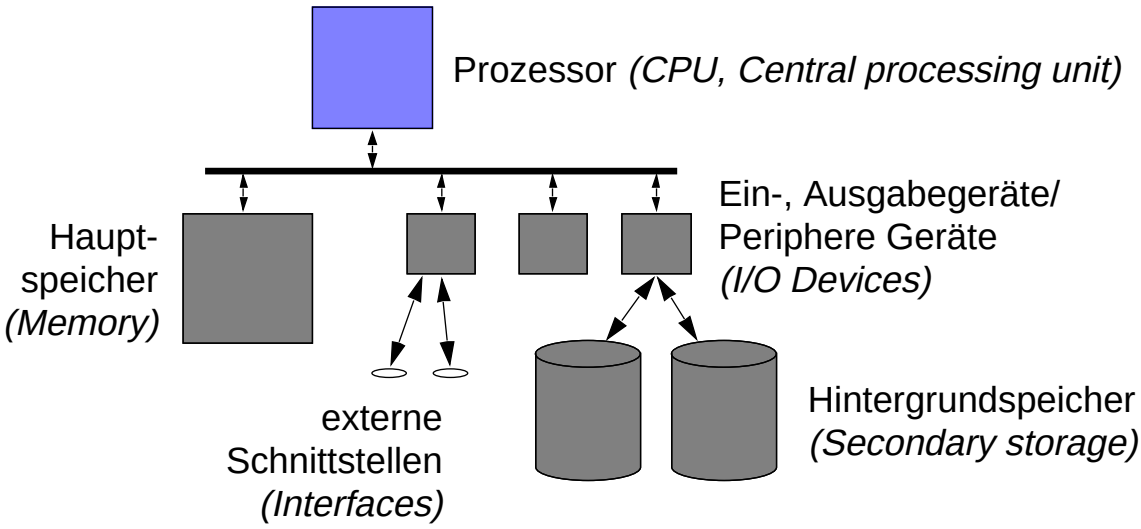
- Rechte pro Datei und Katalog
  - ◆ schreib- und lesbar — nur lesbar *(read only)*
- Keine Benutzeridentifikation
  - ◆ Rechte garantieren keinen Schutz, da veränderbar

3 Dateien

- Attribute
  - ◆ Name, Dateilänge
  - ◆ Attribute: versteckt *(hidden)*, archiviert *(Archive)*, Systemdatei *(System)*
  - ◆ Rechte
  - ◆ Ortsinformation: Nummer des ersten Plattenblocks
  - ◆ Zeitstempel: Erzeugung, letzter Schreib- und Lesezugriff

D Prozesse und Nebenläufigkeit

- Einordnung



D.1 Prozessor

- Register
  - ◆ Prozessor besitzt Steuer- und Vielzweckregister
  - ◆ Steuerregister:
    - Programmzähler *(Instruction pointer)*
    - Stapelregister *(Stack pointer)*
    - Statusregister
    - etc.
- Programmzähler enthält Speicherstelle der nächsten Instruktion
  - ◆ Instruktion wird geladen und
  - ◆ ausgeführt
  - ◆ Programmzähler wird inkrementiert
  - ◆ dieser Vorgang wird ständig wiederholt

D.1 Prozessor (2)

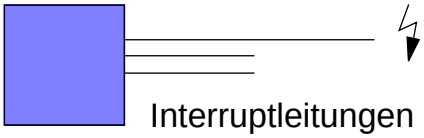
- Beispiel für Instruktionen

```
...
0010 5510000000  movl DS:$10, %ebx
0015 5614000000  movl DS:$14, %eax
001a 8a          addl %eax, %ebx
001b 5a18000000  movl %ebx, DS:$18
...
```

- Prozessor arbeitet in einem bestimmten Modus
  - ◆ Benutzermodus: eingeschränkter Befehlssatz
  - ◆ privilegierter Modus: erlaubt Ausführung privilegierter Befehle
    - Konfigurationsänderungen des Prozessors
    - Moduswechsel
    - spezielle Ein-, Ausgabebefehle

D.1 Prozessor (3)

- Unterbrechungen *(Interrupts)*



Interruptleitungen

Signalisieren der Unterbrechung
(Interrupt request; IRQ)

- Prozessor unterbricht laufende Bearbeitung und führt eine definierte Befehlsfolge aus (vom privilegierten Modus aus konfigurierbar)
- vorher werden alle Register einschließlich Programmzähler gesichert (z.B. auf dem Stack)
- nach einer Unterbrechung kann der ursprüngliche Zustand wiederhergestellt werden
- Unterbrechungen werden im privilegierten Modus bearbeitet

D.1 Prozessor (4)

- Systemaufrufe *(Traps; User interrupts)*
  - Wie kommt man kontrolliert vom Benutzermodus in den privilegierten Modus?
  - spezielle Befehle zum Eintritt in den privilegierten Modus
  - Prozessor schaltet in privilegierten Modus und führt definierte Befehlsfolge aus (vom privilegierten Modus aus konfigurierbar)
  - solche Befehle werden dazu genutzt die Betriebssystemschnittstelle zu implementieren *(Supervisor calls)*
  - Parameter werden nach einer Konvention übergeben (z.B. auf dem Stack)

D.2 Prozesse

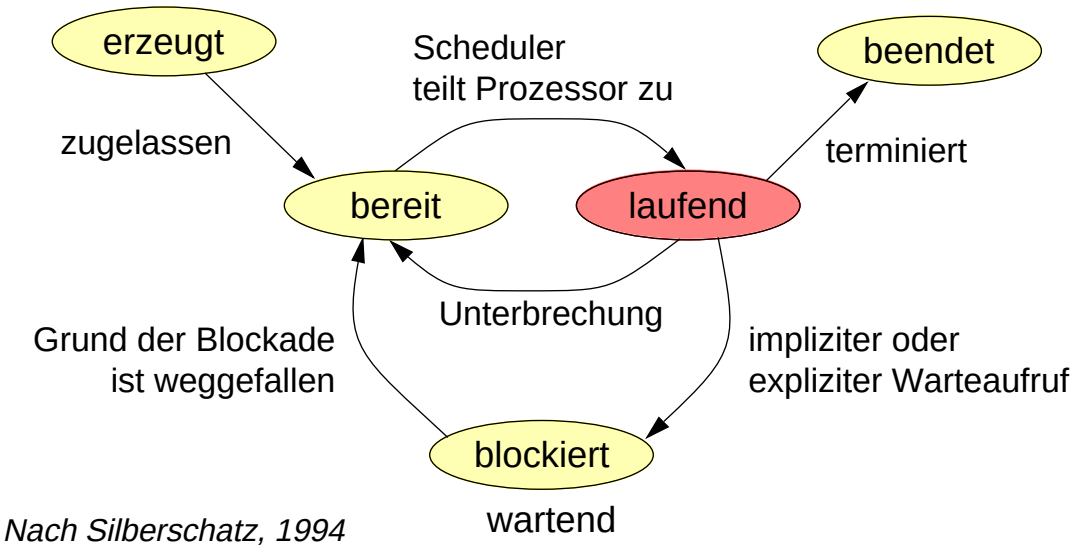
- Stapelsysteme *(Batch systems)*
  - ◆ ein Programm läuft auf dem Prozessor von Anfang bis Ende
- Heutige Systeme *(Time sharing systems)*
  - ◆ mehrere Programme laufen gleichzeitig
  - ◆ Prozessorzeit muß den Programmen zugeteilt werden
  - ◆ Programme laufen nebenläufig
- Terminologie
  - ◆ **Programm:** Folge von Anweisungen
    (hinterlegt beispielsweise als Datei auf dem Hintergrundspeicher)
  - ◆ **Prozeß:** Programm, das sich in Ausführung befindet, und seine Daten
    (ein Programm kann sich mehrfach in Ausführung befinden)

1 Prozeßzustände

- Ein Prozeß befindet sich üblicherweise in einem der folgenden Zustände:
  - ◆ **Erzeugt** *(New)*
    Prozeß wurde erzeugt; Prozeß besitzt noch nicht alle Betriebsmittel zum Laufen
  - ◆ **Bereit** *(Ready)*
    Prozeß besitzt alle nötigen Betriebsmittel und ist bereit zum Laufen
  - ◆ **Laufend** *(Running)*
    Prozeß wird vom realen Prozessor ausgeführt
  - ◆ **Blockiert** *(Blocked/Waiting)*
    Prozeß wartet auf ein Ereignis (z.B. Fertigstellung einer Ein- oder Ausgabeoperation, Zuteilung eines Betriebsmittels, Empfang einer Nachricht); zum Warten wird er blockiert
  - ◆ **Beendet** *(Terminated)*
    Prozeß ist beendet; einige Betriebsmittel sind jedoch noch nicht freigegeben oder Prozeß muß aus anderen Gründen im System verbleiben

1 Prozeßzustände (2)

- Zustandsdiagramm

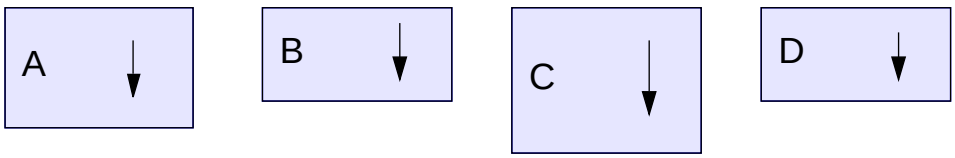


Nach Silberschatz, 1994

- ◆ Scheduler ist der Teil des Betriebssystems, der die Zuteilung des realen Prozessors vornimmt.

2 Prozeßwechsel

- Konzeptionelles Modell

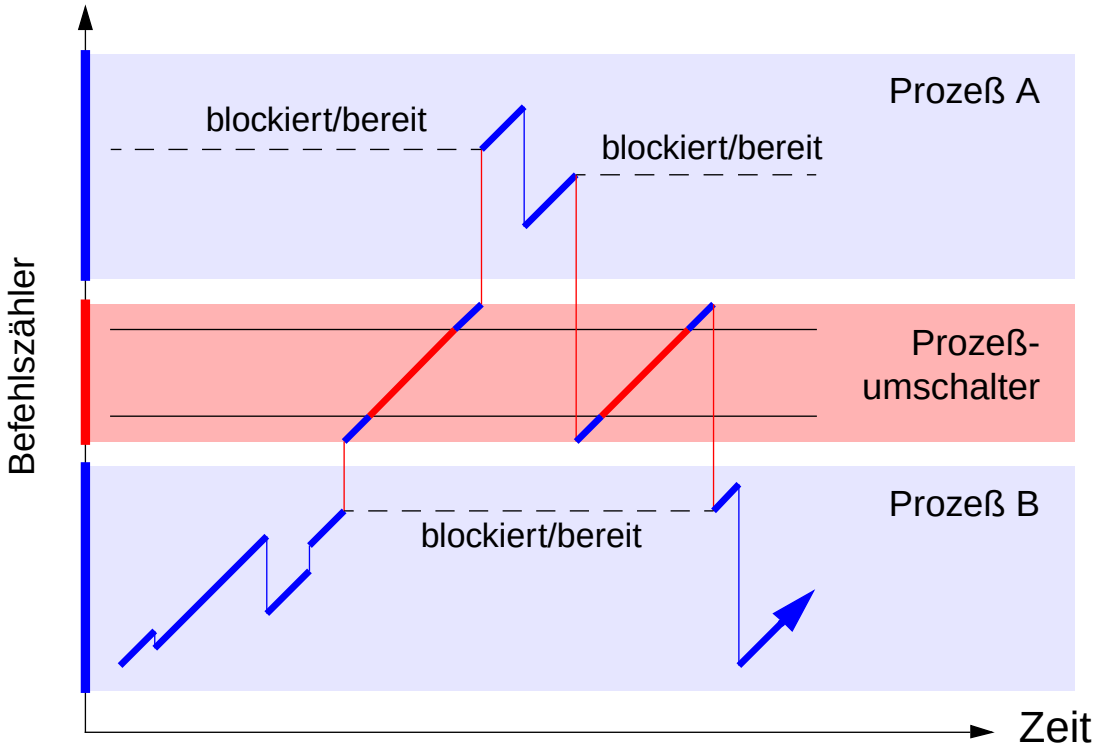


vier Prozesse mit eigenständigen Befehlszählern

- Umschaltung *(Context switch)*
  - ◆ Sichern der Register des laufenden Prozesses inkl. Programmzähler (Kontext),
  - ◆ Auswahl des neuen Prozesses,
  - ◆ Ablaufumgebung des neuen Prozesses herstellen (z.B. Speicherabbildung, etc.),
  - ◆ Gesicherte Register laden und
  - ◆ Prozessor aufsetzen.

2 Prozeßwechsel (2)

- Umschaltung



2 Prozeßwechsel (3)

- Prozeßkontrollblock *(Process control block; PCB)*
  - ◆ Datenstruktur, die alle nötigen Daten für einen Prozeß hält. Beispielsweise in UNIX:
    - Prozeßnummer *(PID)*
    - verbrauchte Rechenzeit
    - Erzeugungszeitpunkt
    - Kontext (Register etc.)
    - Speicherabbildung
    - Eigentümer *(UID, GID)*
    - Wurzelkatalog, aktueller Katalog
    - offene Dateien
    - ...

2 Prozeßwechsel (4)

- Prozeßwechsel unter Kontrolle des Betriebssystems
  - ◆ Mögliche Eingriffspunkte:
    - Systemaufrufe
    - Unterbrechungen
  - ◆ Wechsel nach/in Systemaufrufen
    - Warten auf Ereignisse
      (z.B. Zeitpunkt, Nachricht, Lesen eines Plattenblock)
    - Terminieren des Prozesses
  - ◆ Wechsel nach Unterbrechungen
    - Ablauf einer Zeitscheibe
    - bevorzugter Prozeß wurde laufbereit
- Auswahlstrategie zur Wahl des nächsten Prozesses
  - ◆ *Scheduler*-Komponente

3 Operationen auf Prozessen (Solaris)

- Typische Operationen eines UNIX Betriebssystems
  - ◆ Erzeugen eines neuen Prozesses (dupliziert den gerade laufenden Prozeß)

```
pid_t fork( void );
```

Vater

```
pid_t p;
...
p= fork();
if( p == (pid_t)0 ) {
    /* child */
    ...
} else if( p != (pid_t)-1 ) {
    /* parent */
    ...
} else {
    /* error */
    ...
}
```

3 Operationen auf Prozessen (Solaris)

- Typische Operationen eines UNIX Betriebssystems
  - Erzeugen eines neuen Prozesses (dupliziert den gerade laufenden Prozeß)

```
pid_t fork( void );
```

Vater

```
pid_t p;
...
p= fork();
if( p == (pid_t)0 ) {
    /* child */
    ...
} else if( p != (pid_t)-1 ) {
    /* parent */
    ...
} else {
    /* error */
    ...
}
```

Kind

```
pid_t p;
...
p= fork();
if( p == (pid_t)0 ) {
    /* child */
    ...
} else if( p != (pid_t)-1 ) {
    /* parent */
    ...
} else {
    /* error */
    ...
}
```

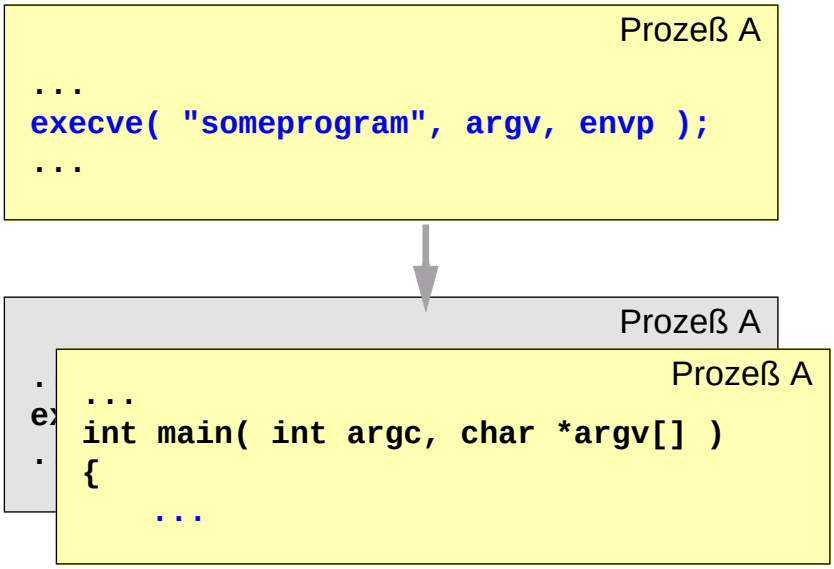

3 Operationen auf Prozessen (2)

- Der Kind-Prozeß ist eine perfekte Kopie des Sohns
  - Gleiches Programm
  - Gleiche Daten (gleiche Werte in Variablen)
  - Gleicher Programmzähler (nach der Kopie)
  - Gleicher Eigentümer
  - Gleiches aktuelles Verzeichnis
  - Gleiche Dateien geöffnet (selbst Schreib-, Lesezeiger ist gemeinsam)
  - ...
- Unterschiede:
  - Verschiedene PIDs
  - **`fork()`** liefert verschiedene Werte als Ergebnis für Vater und Kind

3 Operationen auf Prozessen (3)

◆ Ausführen eines Programms

```
int execve( const char *path, char *const argv[],
            char *const envp[] );
```



Prozeß führt neues Programm vom Beginn an aus

3 Operationen auf Prozessen (4)

◆ Prozeß beenden

```
void _exit( int status );
[ void exit( int status ); ]
```

◆ Prozeßidentifikator

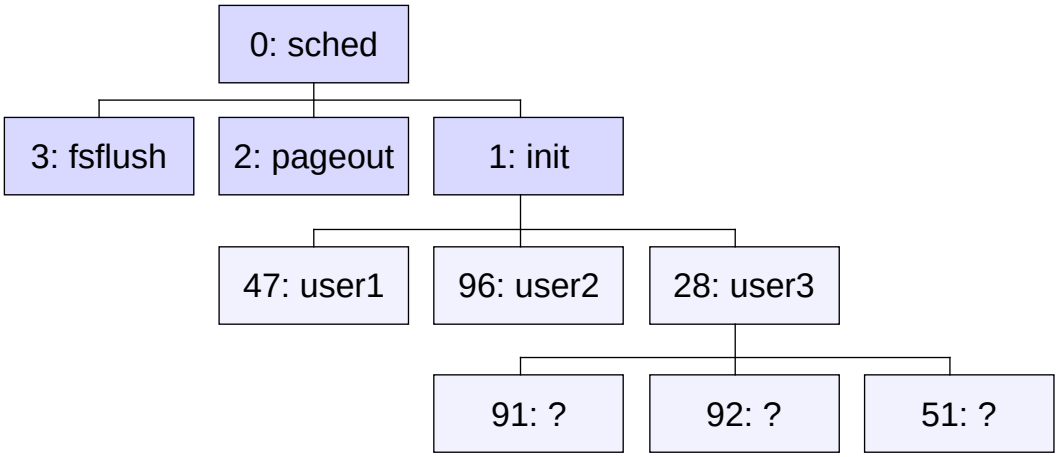
```
pid_t getpid( void );     /* eigene PID */
pid_t getppid( void );    /* PID des Vaterprozesses */
```

◆ Warten auf Beendigung eines Kindprozesses

```
pid_t wait( int *statusp );
```

4 Prozeßhierarchie (Solaris)

- Hierarchie wird durch Vater-Kind–Beziehung erzeugt



Frei nach Silberschatz 1994

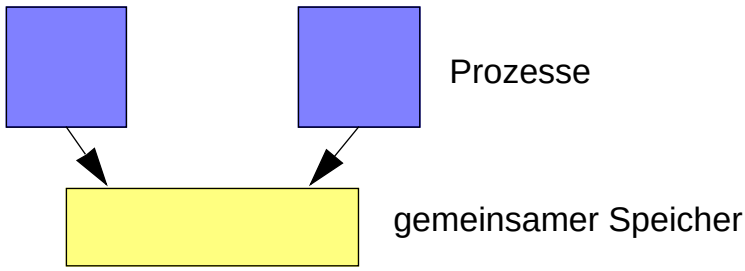
- ◆ Nur der Vater kann auf das Kind warten
- ◆ Init-Prozeß adoptiert verwaiste Kinder

D.3 Aktivitätsträger *(Threads)*

- UNIX-Prozeßkonzept ist für viele heutige Anwendungen unzureichend
  - ◆ in Multiprozessorsystemen werden häufig parallele Abläufe in einem virtuellen Adreßraum benötigt
  - ◆ zur besseren Strukturierung von Problemlösungen sind oft mehrere Aktivitätsträger innerhalb eines Adreßraums nützlich
  - ◆ typische UNIX-Server-Implementierungen benutzen die *fork*-Operation, um einen Server für jeden Client zu erzeugen
    - Verbrauch unnötig vieler Systemressourcen (Dateideskriptoren, Speicher etc.)

1 Prozesse mit gemeinsamen Speicher

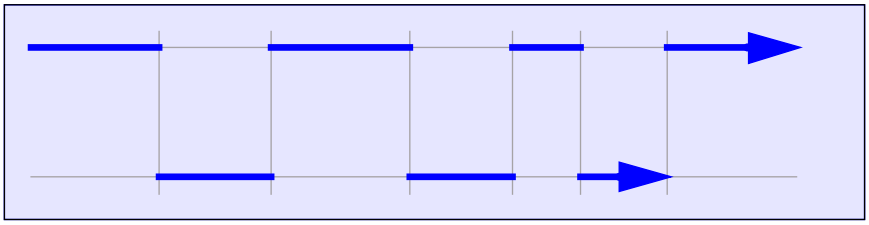
- Gemeinsame Nutzung von Speicherbereichen durch mehrere Prozesse



- ▲ Nachteile
  - ◆ viele Betriebsmittel zur Verwaltung eines Prozesses notwendig
  - ◆ Prozeßumschaltungen sind aufwendig
- ★ Vorteil
  - ◆ In Multiprozessorsystemen sind parallele Abläufe möglich

2 Koroutinen

- Einsatz von Koroutinen
  - ◆ einige Anwendungen lassen sich mit Hilfe von Koroutinen (auf Benutzerebene) innerhalb eines Prozesses gut realisieren



ein Prozeß
zwei Koroutinen

- ▲ Nachteile:
  - ◆ Scheduling zwischen den Koroutinen schwierig (Verdrängung meist nicht möglich)
  - ◆ in Multiprozessorsystemen keine parallelen Abläufe möglich
  - ◆ wird eine Koroutine in einem Systemaufruf blockiert, ist der gesamte Prozeß blockiert

3 Leichtgewichtige Prozesse

- Lösungsansatz: leichtgewichtige Prozesse *(Lightweighted processes)*
  - eine Gruppe leichtgewichtiger Prozesse (LWPs) nutzt gemeinsam eine Menge von Betriebsmitteln
  - jeder leichtgewichtige Prozeß ist aber ein eigener Aktivitätsträger
    - eigener Programmzähler
    - eigener Registersatz
    - eigener Stack
  - Umschalten zwischen zwei leichtgewichtigen Prozessen einer Gruppe ist erheblich billiger als eine normale Prozeßumschaltung
    - es müssen nur die Register und der Programmzähler gewechselt werden (entspricht dem Aufwand für einen Funktionsaufruf)
    - Adreßraum muß nicht gewechselt werden
    - alle Systemressourcen bleiben verfügbar

4 Aktivitätsträger *(Threads)*

- Thread als leichtgewichtiger Prozeß *(Leight weight process, LWP)*
  - keine eigene Speicherabbildung
  - mehrere Threads teilen sich einen Speicherbereich und arbeiten im gleichen Instruktionen- und Datenbereich

★ Ein Prozeß ist ein Task mit einem Thread
  - Solaris: Prozeß kann mehrere Threads besitzen

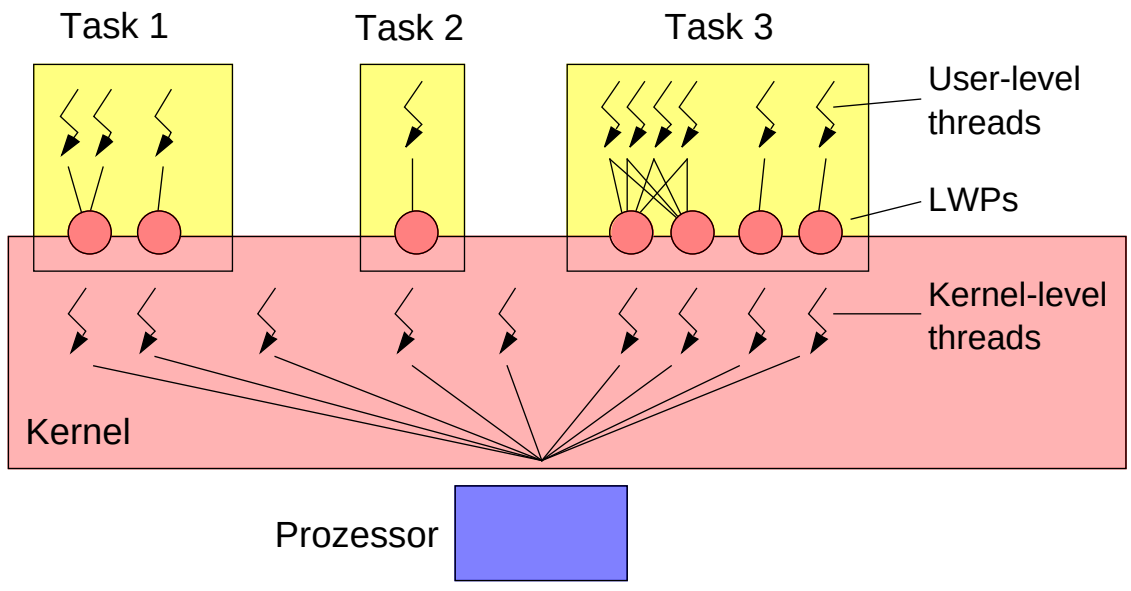
- Implementierungen von Threads
  - User-level threads
  - Kernel-level threads

4 Aktivitätsträger (2)

- User-level threads
  - ◆ Instruktionen im Anwendungsprogramm schalten zwischen den Threads hin- und her (ähnlich wie der Scheduler im Betriebssystem)
  - ◆ Betriebssystem sieht nur einen Thread
    - keine Systemaufrufe zum Umschalten erforderlich
    - effiziente Umschaltung
  - ◆ Bei blockierenden Systemaufrufen bleiben alle User-level threads stehen
- Kernel-level threads
  - ◆ Betriebssystem schaltet Threads um
    - weniger effizientes Umschalten
  - ◆ Kein Blockieren unbeteiligter Threads bei blockierenden Systemaufrufen
  - ◆ Fairneßverhalten nötig
    (zwischen Prozesen mit vielen und solchen mit wenigen Threads)

5 Beispiel: LWPs und Threads (Solaris)

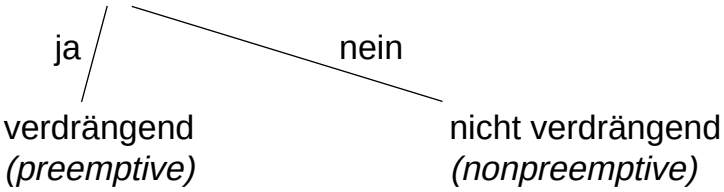
- Solaris kennt Kernel- und User-level threads



Nach Silberschatz, 1994

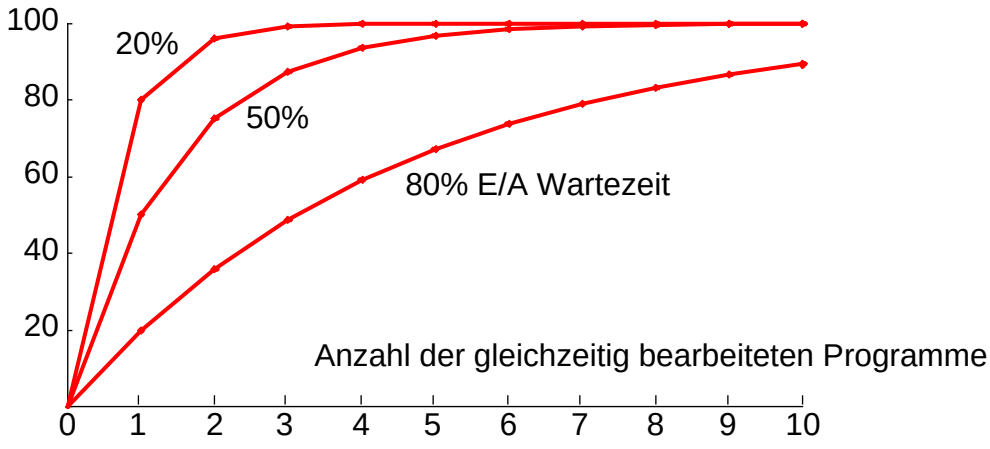
D.4 Auswahlstrategien

- Strategien zur Auswahl des nächsten Prozesses *(Scheduling strategies)*
  - Mögliche Stellen zum Treffen von Schedulingentscheidungen
    1. Prozeß wechselt vom Zustand „laufend“ zum Zustand „blockiert“ (z.B. Ein-, Ausgabeoperation)
    2. Prozeß wechselt von „laufend“ nach „bereit“ (z.B. bei einer Unterbrechung des Prozessors)
    3. Prozeß wechselt von „blockiert“ nach „bereit“
    4. Prozeß terminiert
  - Keine Wahl bei 1. und 4.
  - Wahl bei 2. und 3.



D.4 Auswahlstrategien (2)

- CPU Auslastung
  - CPU soll möglichst vollständig ausgelastet sein
- ★ CPU-Nutzung in Prozent, abhängig von der Anzahl der Programme und deren prozentualer Wartezeit



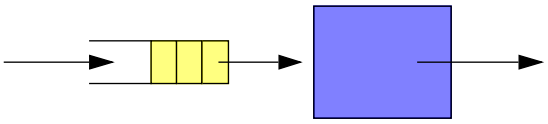
Nach Tanenbaum, 1995

D.4 Auswahlstrategien (3)

- Durchsatz
  - Möglichst hohe Anzahl bearbeiteter Prozesse pro Zeiteinheit
- Verweilzeit
  - Gesamtzeit des Prozesses in der Rechenanlage soll so gering wie möglich sein
- Wartezeit
  - Möglichst kurze Gesamtzeit, in der der Prozeß im Zustand „bereit“ ist
- Antwortzeit
  - Möglichst kurze Reaktionszeit des Prozesses im interaktiven Betrieb

1 First-Come, First Served

- Der erste Prozeß wird zuerst bearbeitet *(FCFS)*
  - „Wer zuerst kommt ...“
  - Nicht-verdrängend
- Warteschlange zum Zustand „bereit“
  - Prozesse werden hinten eingereiht
  - Prozesse werden vorne entnommen



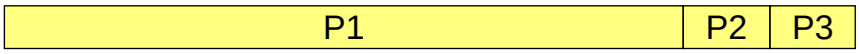
- Bewertung
  - fair (?)
  - Wartezeiten nicht minimal
  - nicht für Time sharing-Betrieb geeignet

1 First Come, First Served (2)

- Beispiel zur Betrachtung der Wartezeiten

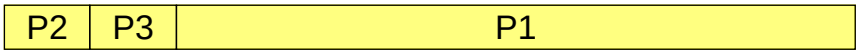
Prozeß 1:	24	} Zeiteinheiten
Prozeß 2:	3	
Prozeß 3:	3	

  - Reihenfolge: P1, P2, P3



mittlere Wartezeit: (0+24+27)/3 = 17

  - Reihenfolge: P2, P3, P1



mittlere Wartezeit: (6+0+3)/3 = 3

2 Shortest Job First

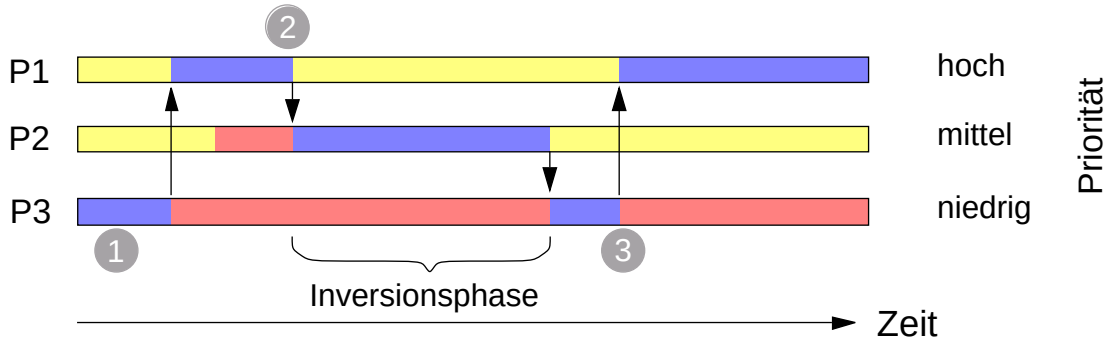
- Kürzester Job wird ausgewählt *(SJF)*
  - Länge bezieht sich auf die nächste Rechenphase bis zur nächsten Warteoperation (z.B. Ein-, Ausgabe)
- „bereit“-Warteschlange wird nach Länge der nächsten Rechenphase sortiert
  - Vorhersage der Länge durch Protokollieren der Länge bisheriger Rechenphasen (Mittelwert, exponentielle Glättung)
  - ... Protokollierung der Länge der vorherigen Rechenphase
- SJF optimiert die mittlere Wartezeit
  - Da Länge der Rechenphase in der Regel nicht genau vorhersagbar, nicht ganz optimal.
- Variante: verdrängend *(PSJF)* und nicht-verdrängend

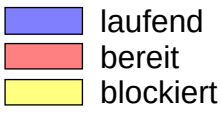
3 Prioritäten

- Prozeß mit höchster Priorität wird ausgewählt
  - dynamisch — statisch
    (z.B. SJF: dynamische Vergabe von Prioritäten gemäß Länge der nächsten Rechenphase)
    (z.B. statische Prioritäten in Echtzeitsystemen; Vorhersagbarkeit von Reaktionszeiten)
  - verdrängend — nicht-verdrängend
- Probleme
  - Aushungerung
    Ein Prozeß kommt nie zum Zuge, da immer andere mit höherer Priorität vorhanden sind.
  - Prioritätenumkehr *(Priority inversion)*

3 Prioritäten (2)

- Prioritätenumkehr
  - hochpriorer Prozeß wartet auf ein Betriebsmittel, das ein niedrigpriorer Prozeß besitzt; dieser wiederum wird durch einen mittelprioren Prozeß verdrängt und kann daher das Betriebsmittel gar nicht freigeben





1. P3 fordert Betriebsmittel an
2. P1 wartet auf das gleiche Betriebsmittel
3. P3 gibt Betriebsmittel frei

3 Prioritäten (3)

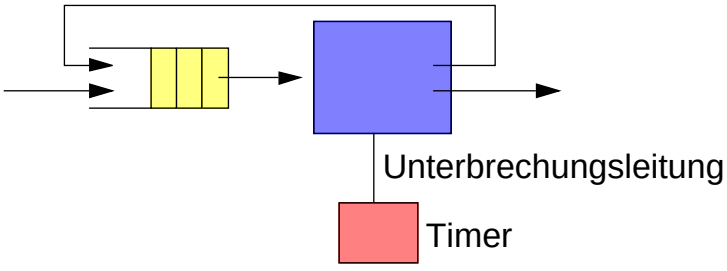
★ Lösungen

- ◆ zur Prioritätenumkehr:
  dynamische Anhebung der Priorität für kritische Prozesse
- ◆ zur Aushungerung:
  dynamische Anhebung der Priorität für lange wartende Prozesse
  (Alterung, *Aging*)

4 Round Robin Scheduling

■ Zuteilung und Auswahl erfolgt reihum

- ◆ ähnlich FCFS aber mit Verdrängung
- ◆ Zeitquant *(Time quantum)* oder Zeitscheibe *(Time slice)* wird zugeteilt
- ◆ geeignet für *Time sharing*-Betrieb



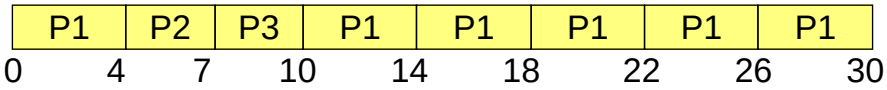
- ◆ Wartezeit ist jedoch eventuell relativ lang

4 Round Robin Scheduling (2)

- Beispiel zur Betrachtung der Wartezeiten

Prozeß 1:	24	} Zeiteinheiten
Prozeß 2:	3	
Prozeß 3:	3	

  - ◆ Zeitquant ist 4 Zeiteinheiten
  - ◆ Reihenfolge in der „bereit“-Warteschlange: P1, P2, P3



mittlere Wartezeit: (6+4+7)/3 = 5.7

4 Round Robin Scheduling (3)

- Effizienz hängt von der Größe der Zeitscheibe ab
  - ◆ kurze Zeitscheiben: Zeit zum Kontextwechsel wird dominant
  - ◆ lange Zeitscheiben: Round Robin nähert sich FCFS an
- Verweilzeit und Wartezeit hängt ebenfalls von der Zeitscheibengröße ab
  - ◆ Beispiel: 3 Prozesse mit je 10 Zeiteinheiten Rechenbedarf
    - Zeitscheibengröße 1:
      durchschnittliche Verweilzeit: 29 Zeiteinheiten = (28+29+30)/3
      durchschnittliche Wartezeit: 19 Zeiteinheiten = (18+19+20)/3
    - Zeitscheibengröße 10:
      durchschnittliche Verweilzeit: 20 Zeiteinheiten = (10+20+30)/3
      durchschnittliche Wartezeit: 10 Zeiteinheiten = (0+10+20)/3

5 Multilevel Queue Scheduling

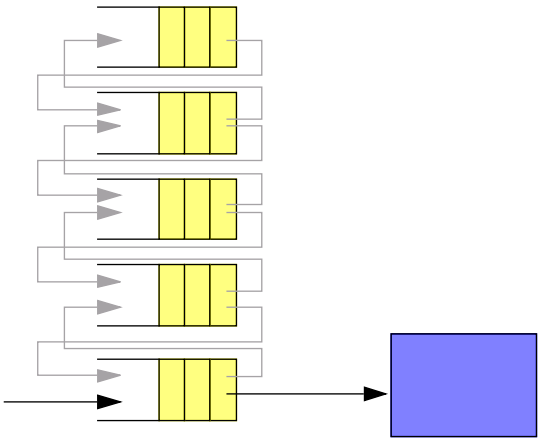
- Verschiedene Schedulingklassen
  - z.B. Hintergrundprozesse (Batch) und Vordergrundprozesse (interaktive Prozesse)
  - jede Klasse besitzt ihre eigenen Warteschlangen und verwaltet diese nach einem eigenen Algorithmus
  - zwischen den Klassen gibt es ebenfalls ein Schedulingalgorithmus z.B. feste Prioritäten (Vordergrundprozesse immer vor Hintergrundprozessen)
- Beispiel: Solaris
  - Schedulingklassen
    - Systemprozesse
    - Real-time Prozesse
    - Time-sharing Prozesse
    - interaktive Prozesse

5 Multilevel Queue Scheduling

- Scheduling zwischen den Klassen mit fester Priorität (z.B. Real-time Prozesse vor Time sharing-Prozessen)
- In jeder Klasse wird ein eigener Algorithmus benutzt:
  - Systemprozesse: FCFS
  - Real-time Prozesse: statische Prioritäten
  - Time-sharing und interaktive Prozesse: ausgefeiltes Verfahren zur Sicherung von:
    - kurzen Reaktionszeiten
    - fairer Zeitaufteilung zwischen rechenintensiven und I/O-intensiven Prozessen
    - gewisser Benutzersteuerung
- ★ Multilevel Feedback Queue Scheduling

6 Multilevel Feedback Queue Scheduling

- Mehrere Warteschlangen *(MLFB)*
  - jede Warteschlange mit eigener Behandlung
  - Prozesse können von einer zur anderen Warteschlange transferiert werden



6 Multilevel Feedback Queue Scheduling (2)

- Beispiel:
  - mehrere Warteschlangen mit Prioritäten (wie bei Multilevel Queue)
  - Prozesse, die lange rechnen, wandern langsam in Warteschlangen mit niedrigerer Priorität (bevorzugt interaktive Prozesse)
  - Prozesse, die lange warten müssen, wandern langsam wieder in höherpriore Warteschlangen *(Aging)*