Überblick: Teil D Betriebssystemabstraktionen

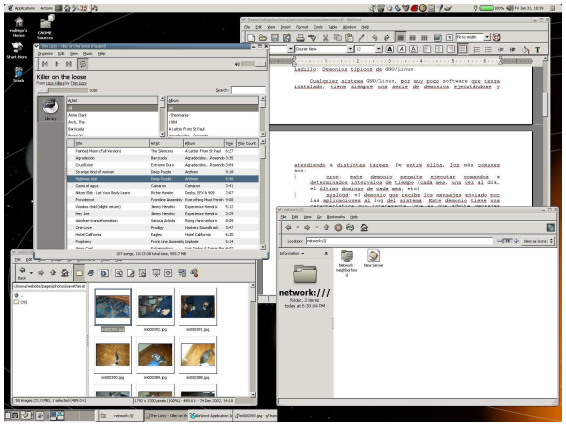
Definition „Betriebssystem”

- DIN 44300
  - „... die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften der Rechenanlage die Basis der möglichen Betriebsarten des digitalen Rechensystems bilden und die insbesondere die Abwicklung von Programmen steuern und überwachen.”
- Andy Tannenbaum
  - „... eine Software-Schicht ..., die alle Teile eines Systems verwaltet und dem Benutzer eine Schnittstelle oder eine virtuelle Maschine anbietet, die einfacher zu verstehen und zu programmieren ist [als die nackte Hardware].”
- Zusammenfassung:
  - Software zur Verwaltung und Virtualisierung der Hardware-Komponenten (Betriebsmittel)
  - Programm zur Steuerung und Überwachung anderer Programme

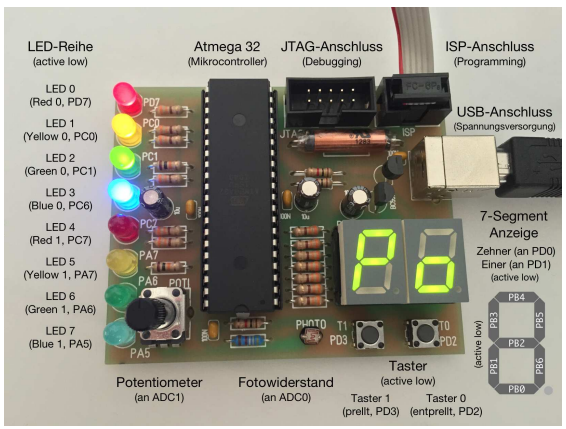
Überblick

Bisher:

- **Ein** Programm, dass
- **alleine**,
- **beim Booten** gestartet
- mit **Hardware-Zugriffen**
- seine Umgebung steuert.



Quelle: www.wikipedia.org



Jetzt:

- **Mehrere** Programme, die
- **nebenläufig**,
- **dynamisch** gestartet/beendet
- über **definierte E-/A-Funktionen**
- ihre Umgebung steuern.

Multitasking

Existiert mehr als eine Anwendung auf einem System („Multitasking”),

- müssen sich die Anwendungen abstimmen,
  - wer wann die/eine CPU bekommt,
  - wer welche Speicherbereiche verwenden darf,
  - wer welche Plattenblöcke verwenden darf,
  - wer welchen Teil des Bildschirms beschreiben darf,
  - ...

  Da keine Anwendung für sich allein z.B. entscheiden kann, welche Speicherbereiche noch ungenutzt sind, muss es **gemeinsam benutzte Methoden und Zustandsvariablen** geben.
- muss sichergestellt sein, dass
  - sich alle Anwendungen an die Abmachungen halten (auch die versehentlich/absichtlich fehlerhaft programmierten!)

  Hardware-Erweiterungen müssen Zugriffe auf unerlaubte Speicherbereiche bzw. E-/A-Geräte verhindern.

Betriebssystem-Kern

- „**gemeinsam benutzte Methoden und Zustandsvariablen**”
  - **Betriebssystem-Kern** („Kern”, „System-Kern”, „Kernel”)
- „**Hardware-Erweiterungen**”
  - **Priviligierungsstufen** („Privilege Level”, „Ringe”)
  - **Speicherschutz** („Memory Management Unit” („MMU”))

Priviligierungsstufen

- unpriviligierte Ebene
  („Anwendungsebene”, „User-Ebene”, „User-Ring”)
  - darf „normale” CPU-Instruktionen ausführen,
  - darf auf den ihr zugewiesenen Speicher zugreifen,
  - darf Betriebssystem-Methoden aufrufen
- priviligierte Ebene
  („System-Ebene”, „Kern-Ebene”, „Ring 0”)
  - darf alle CPU-Instruktionen ausführen,
  - darf auf jeden Speicherbereich zugreifen,
  - darf Speicherschutz umkonfigurieren,
  - darf auf E-/A-Geräte zugreifen

Wechsel in die priviligierte Ebene durch

- System-Aufrufe („System Calls”, „Traps”)
- Unterbrechnungen („Interrupts”)
- Ausnahmen („Exceptions”)

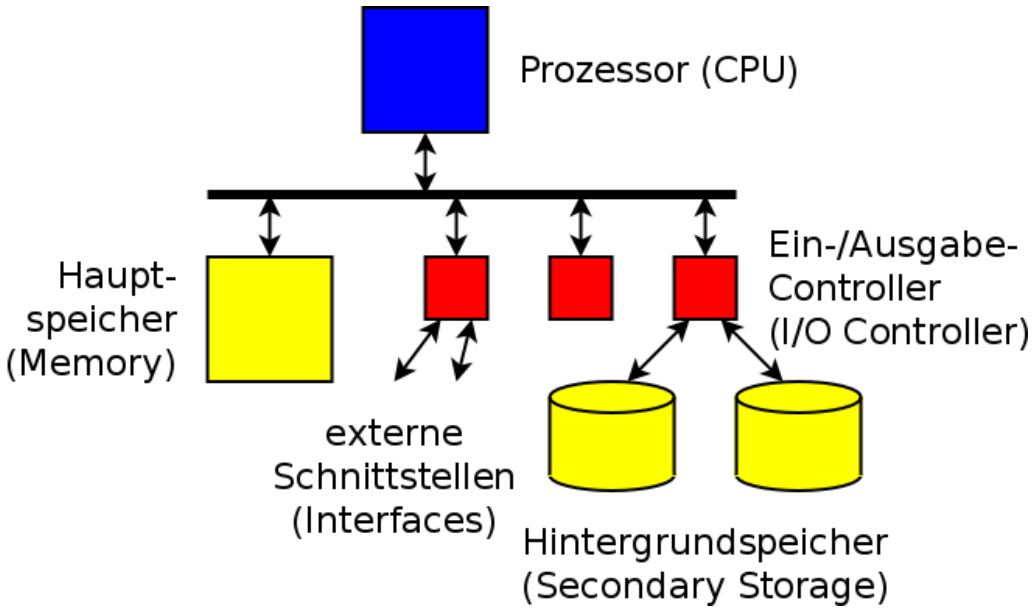
System-Aufrufe

Beispiel: Anwendung braucht mehr Speicher
Schritte:

- Anwendung berechnet, wieviel Speicher sie benötigt,
- legt Parameter in CPU-Registern ab,
- wechselt mit spezieller CPU-Instruktion in den Kern,
(=> ab jetzt priviligiert!)
- liest Parameter aus CPU-Registern,
- reserviert Speicher für sich,
- programmiert MMU um,
- legt Ergebnis in CPU-Registern ab,
- wechselt mit spezieller CPU-Instruktion zurück in Anwender-Ebene,
(=> ab jetzt wieder unpriviligiert!)
- und holt Ergebnis aus CPU-Register.

Verwaltung von Betriebsmitteln

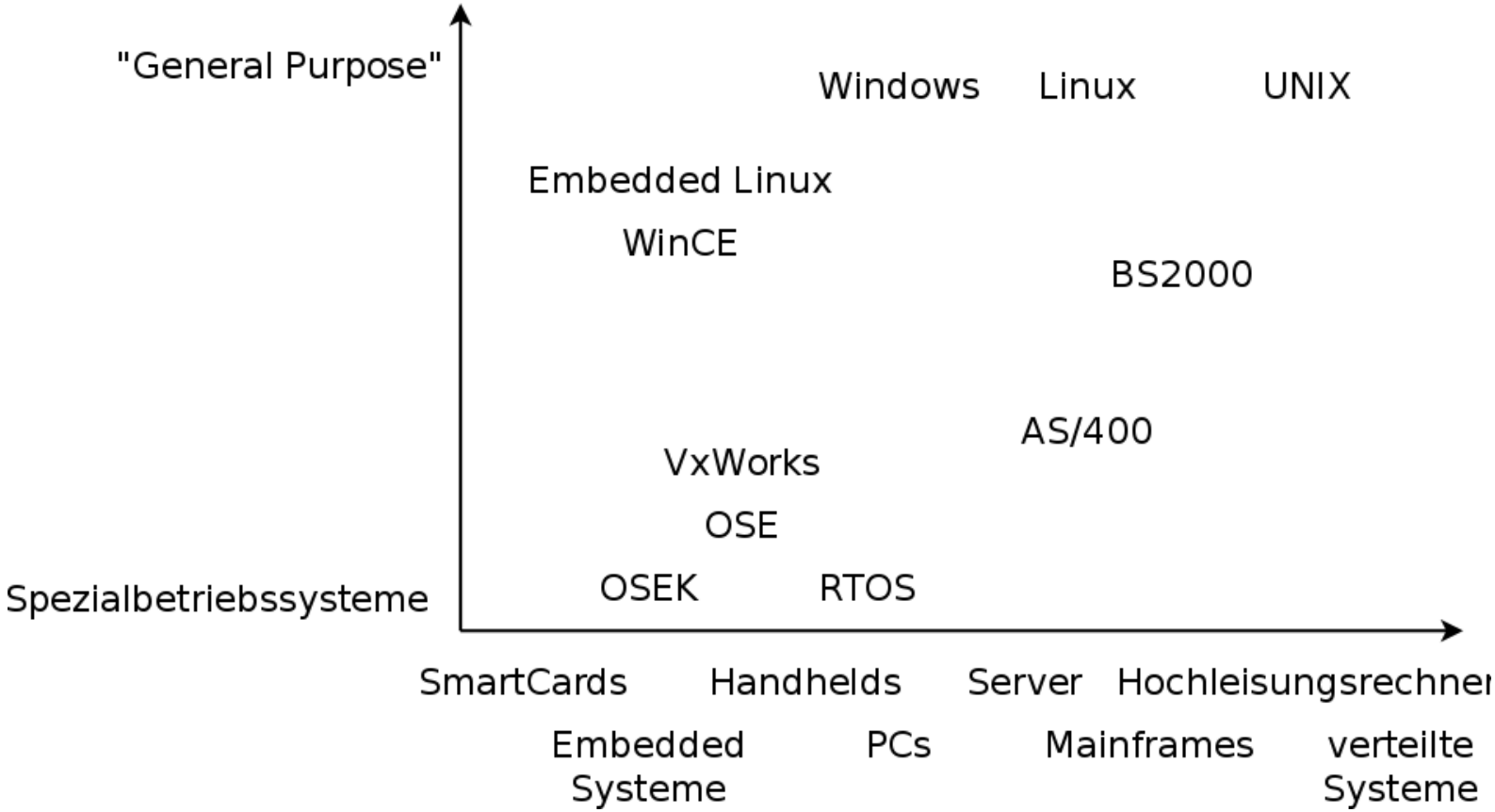
- Aufgaben des Betriebssystem-Kerns
  - Multiplexen von Betriebsmitteln für mehrere Benutzer bzw. Anwendungen
  - Schaffung von Schutzumgebungen
  - Bereitstellen von Abstraktionen zur besseren Handhabbarkeit der Betriebsmittel



- Ermöglichen einer koordinierten gemeinsamen Nutzung von Betriebsmitteln, klassifizierbar in
  - aktive, zeitlich aufteilbare (Prozessor)
  - passive, nur exklusiv nutzbare (periphere Geräte, z.B. Drucker u.Ä.)
  - passive, räumlich aufteilbare (Speicher, Plattenspeicher u.Ä.)
- Unterstützung bei der Fehlererholung

Klassifikation von Betriebssystemen

- Unterschiedliche Klassifikationskriterien
  - Zielplattform
  - Einsatzzweck
  - Funktionalität



Klassifikation von Betriebssystemen (2)

- Wenigen „General Purpose"-, Mainframe- und Höchstleistungsrechner-Betriebssystemen steht eine Vielzahl kleiner und kleinster Spezialbetriebssysteme gegenüber:

  C51, C166, C251, CMX RTOS, C-Smart/Raven, eCos, eRTOS, Embos, Ercos, Euros Plus, Hi Ross, Hynet-OS, LynxOS, MicroX/OS-II, Nucleus, OS-9, OSE, OSEK Flex, OSEK Turbo, OSEK Plus, OSEKtime, Pricise/MQX, Pricise/RTCS, proOSEK, SOS, PXROS, QNX, Realos, RTMOSxx, Real Time Architect, ThreadX, RTA, RTX51, RTX251, RTX166, RTXC, Softune, SSXS RTOS, VRTX, VxWorks, ...

  Einsatzbereich: Eingebettete Systeme, häufig Echtzeit-Betriebssysteme, über 50% proprietäte (in-house) Lösungen

- Alternative Klassifikation: nach Architektur

Betriebssystem-Architekturen

- Umfang: zehntausende bis mehrere Millionen Befehlszeilen
  => Strukturierung hilfreich
- Verschiedene Strukturkonzepte
  - Laufzeitbibliotheken (minimal, vor allem im Embedded-Bereich)
  - monolithische Systeme
  - geschichtete Systeme
  - Minimalkerne
- Unterschiedliche Schutzkonzepte
  - kein Schutz
  - Schutz des Betriebssystems
  - Schutz des Betriebssystems und Anwendungen untereinander
  - feingranularer Schutz auch innerhalb von Anwendungen

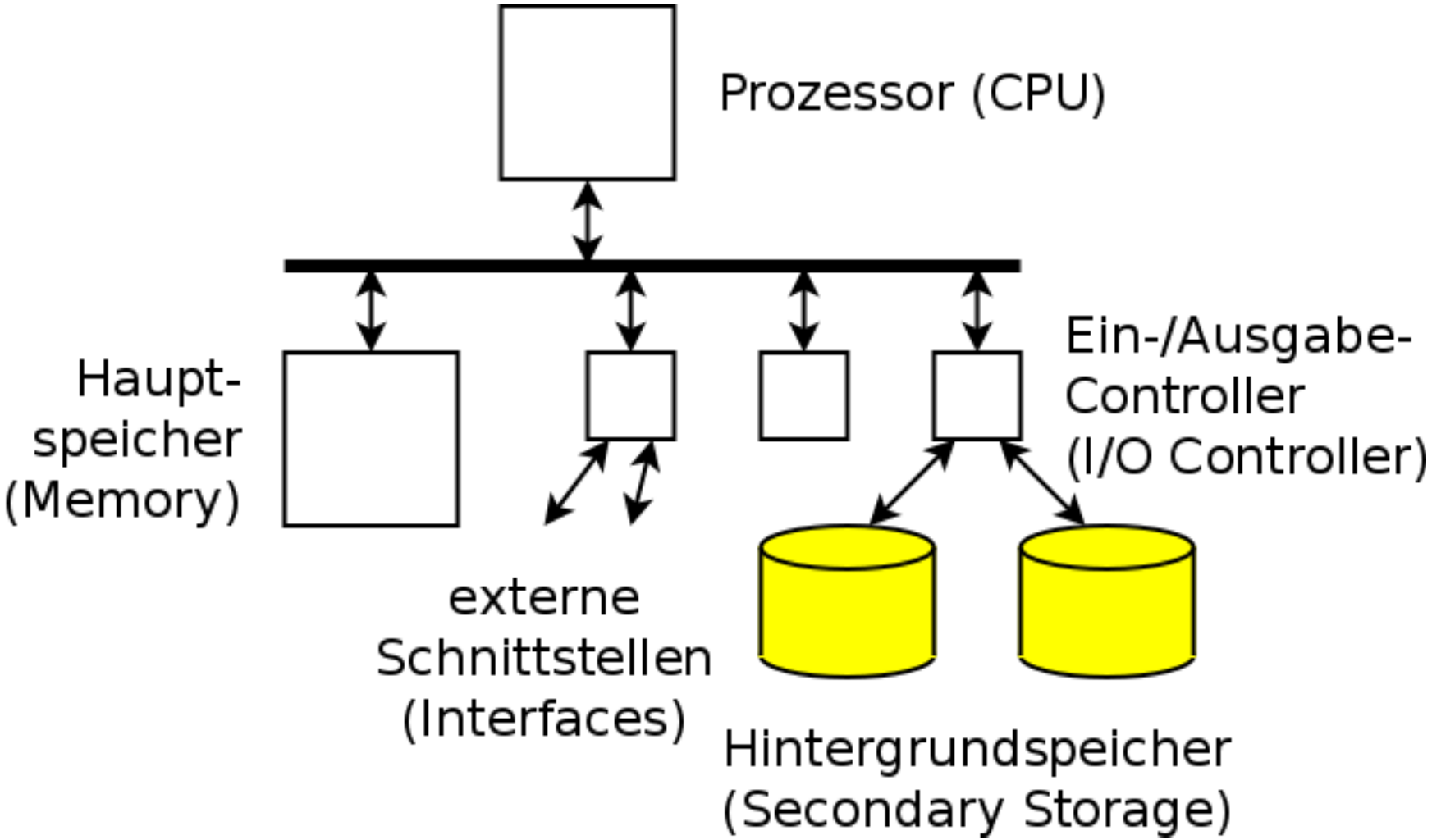
Betriebssystem-Komponenten

- Speicherverwaltung
  - Wer darf Informationen wohin im Speicher ablegen?
- Prozessverwaltung
  - Wann wird welche Aufgabe bearbeitet?
- Dateisystem
  - Speicherung und Schutz von Langzeitdaten
- Interprozesskommunikation
  - Kommunikation zwischen verschiedenen Anwendungen bzw. parallel ablaufenden Anwendungsteilen
- Ein-/Ausgabe
  - Kommunikation mit der „Außenwelt” (Benutzer/Rechner)

Überblick: Teil D Betriebssystemabstraktionen

Überblick

- Einordnung



Überblick (2)

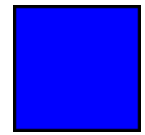
- Speichermedien (z.B. Platten, SSD / Flash-Speicher, DVD, CD-ROM) mit Unterschieden; Beispiele
  - Blockgrößen:
    - Festplatten: 512 Bytes/Block
    - CDs: 1024 Bytes/Block
    - Flash: 4096 Bytes/Block
  - Nutzung der Blöcke
    - Flash-Speicher hat nur begrenzte Anzahl von Schreibzyklen pro Block => gleichmäßig beschreiben
    - Festplatten können auf benachbarte Blöcke jeweils schneller zugreifen
  - Größe der Medien (typ.)
    - CD-ROM: ca. 750 MByte
    - DVD: ca. 8,5 GByte
    - Festplatte: ca. 4 TByte
    - SSD: ca. 500 GByte

Überblick (3)

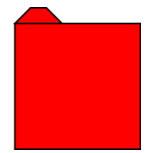
- Dateisysteme speichern Daten und Programme persistent in Dateien
  - Benutzer muss sich nicht um die Ansteuerung und Verwaltung verschiedener Speichermedien kümmern
  - einheitliche Sicht auf den Hintergrundspeicher
- Wesentliche Elemente eines Dateisystems:
  - Dateien (Files)
  - Verzeichnisse / Kataloge (Directories)
  - Partitionen (Partitions)

Überblick (4)

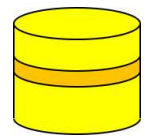
- Datei (File)
  - speichert Daten oder Programme
  - enthält Zusatzinformationen
- Verzeichnis / Katalog (Directory)
  - fasst Dateien (u. Verzeichnisse) zusammen
  - erlaubt Benennung der Dateien
  - ermöglicht Aufbau eines hierarchischen Namensraums
- Partition (Partition)
  - eine Menge von Verzeichnissen und deren Dateien
  - sie dienen zum physikalischen oder logischen Trennen von Dateimengen
    - physisch: Festplatte, Diskette
    - logisch: Teilbereich auf Platte oder CD



Datei



Verzeichnis



Partition

Dateien

- Kleinste Einheit, in der etwas auf den Hintergrundspeicher geschrieben werden kann.
- Unterscheidung:
  - eigentliche Daten (Bild, Text, Programm, ...)
  - Metadaten (Erstellungsdatum, Eigentümer, Zugriffsrechte, ...)

Metadaten / Dateiattribute:

Name: Symbolischer Name, vom Benutzer les- und interpretierbar

- z.B. `AUTOEXEC.BAT`

Typ: Für Dateisysteme, die verschiedene Dateitypen unterscheiden

- z.B. sequenzielle Datei, zeichenorientierte Datei, satzorientierte Datei

Ort: Wo werden die Daten physisch gespeichert?

- Nummern der Plattenblöcke

Dateiattribute (2)

Größe: Länge der Datei in Größeneinheiten (z.B. Bytes, Blöcke, Sätze)

- steht in engem Zusammenhang mit der Ortsinformation
- wird zum Prüfen der Dateigrenzen z.B. beim Lesen benötigt

Zeitstempel: z.B. Zeit und Datum der Erstellung, letzten Änderung

- für Backup, Entwicklungswerkzeuge, Benutzerüberwachung, ...

Rechte: Zugriffsrechte, z.B. Lese- und Schreibberechtigung

- z.B. nur für den Eigentümer schreibbar, für alle anderen nur lesbar

Eigentümer: Identifikation des Eigentümers

- eventuell eng mit den Rechten verknüpft
- Zuordnung beim Accouting (Abrechnung von Plattenplatz)

Operationen auf Dateien

- **Erzeugen (Create)**
  - Nötiger Speicherplatz wird angefordert
  - Verzeichniseintrag wird erstellt
  - Initiale Attribute werden gespeichert
- **Schreiben (Write)**
  - Identifikation der Datei
  - eventuell Nachfordern von Speicherplatz
  - Daten werden auf Platte geschrieben
  - eventuell Anpassung der Attribute (z.B. Länge der Datei, Zeitpunkt der letzten Änderung)
- **Lesen (Read)**
  - Identifikation der Datei
  - Daten werden von Platte gelesen
  - eventuell Anpassung der Attribute (z.B. Zugriffszeit)

Operationen auf Dateien

- **Positionieren** des Schreib-/Lesezeigers für die nächste Schreib- bzw. Leseoperation **(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 (oder ab Anfang) wird der Inhalt der Datei gelöscht
  - eventuell Freigeben von Speicherplatz
  - Anpassung der Attribute (z.B. Länge der Datei, Zeitpunkt der letzten Änderung)
- **Löschen (Delete)**
  - Identifikation der Datei
  - Entfernen der Datei aus dem Verzeichnis und Freigabe der Plattenblöcke

Verzeichnisse / Kataloge

- Ein Verzeichnis gruppiert Dateien und evtl. weitere Verzeichnisse
- Gruppierungsalternativen
  - Verknüpfung mit Benennung
    - Verzeichnis enthält Namen und Verweise auf Dateien und andere Verzeichnisse (z.B. UNIX, Windows)
  - Gruppierung über Bedingung
    - Verzeichnis enthält Namen und Verweise auf Dateien, die einer bestimmten Bedingung gehorchen:
      – z.B. gleiche Gruppennummer in CP/M
      – z.B. eigenschaftsorientierte und dynamische Gruppierung in BeOS-BFS
- Verzeichnis ermöglicht das Auffinden von Dateien
  - Vermittlung zwischen externer und interner Bezeichnung (Dateiname – Plattenblöcke)

Operationen auf Verzeichnissen

- **Lesen** der Einträge (**Read, Read Directory)**
  - Daten des Verzeichnisses werden gelesen und meist eintragsweise zurückgegeben
- **Erzeugen** und **Löschen** der Einträge erfolgt implizit beim Anlegen bzw. Löschen der Dateien
- **Erzeugen** von Verzeichnissen **(Create, Create Directory)**
- **Löschen** von Verzeichnissen **(Delete, Delete Directory)**

Attribute von Verzeichnissen

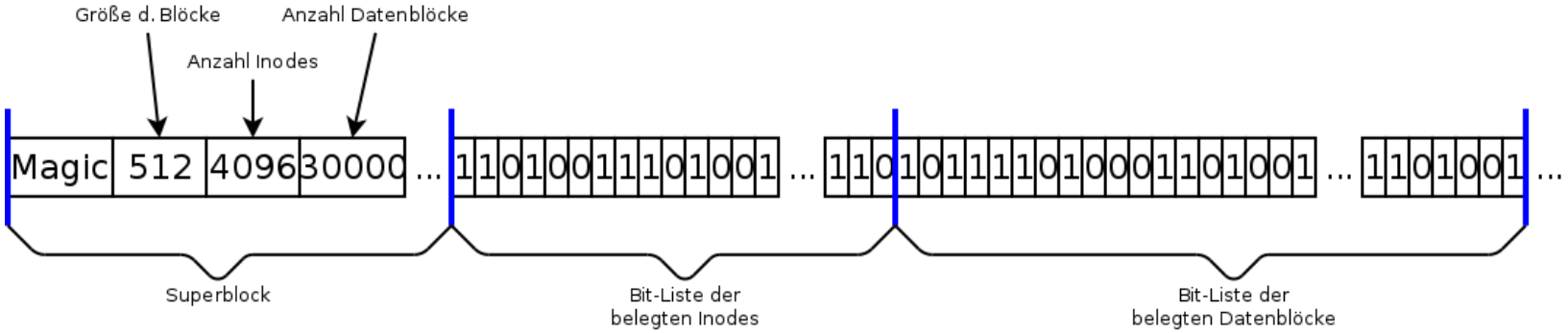
- Die meisten Attribute von Dateien treffen auch auf Verzeichnissen zu
  - Name, Ortsinformation, Größe, Zeitstempel, Rechte, Eigentümer, ...

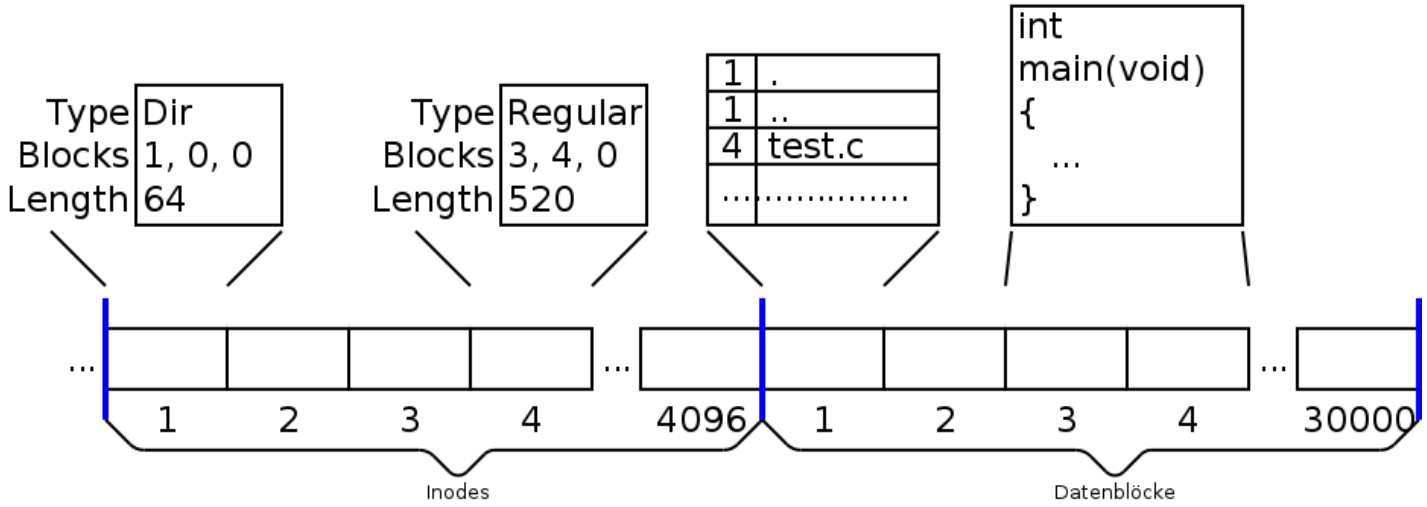
Dateisystem am Beispiel Linux/UNIX

- Datei
  - einfache, unstrukturierte Folge von Bytes
  - beliebiger Inhalt; für das Betriebssystem ist der Inhalt transparent
  - dynamisch erweiterbar
- Dateiattribute
  - das Betriebssystem verwaltet zu jeder Datei eine Reihe von Attributen (Rechte, Größe, Zugriffszeiten, Datenblöcke, ...)
  - die Attribute werden in einer speziellen Verwaltungsstruktur, dem *Dateikopf*, gespeichert
    - Linux/UNIX: *Inode*
    - Windows NTFS: *Master File Table*-Eintrag
- Namensraum
  - flacher Namensraum: Inodes sind einfach durchnummeriert
  - hierarchischer Namensraum: Verzeichnisstruktur bildet Datei- und Pfadnamen in einem Dateibaum auf Inode-Nummern ab

Dateisystem-Struktur

- Struktur auf Medium (vereinfacht)

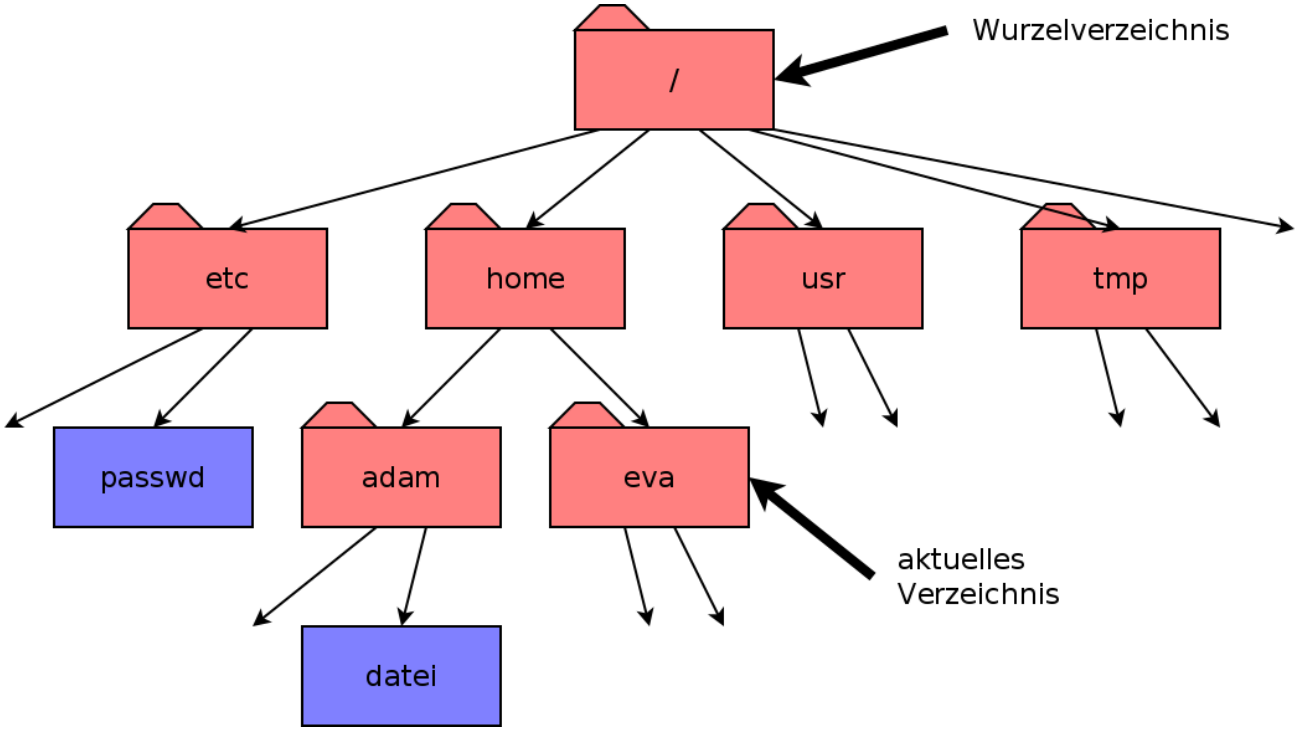




- `mkfs` legt leere Struktur an; `fsck` überprüft Struktur

Pfadnamen

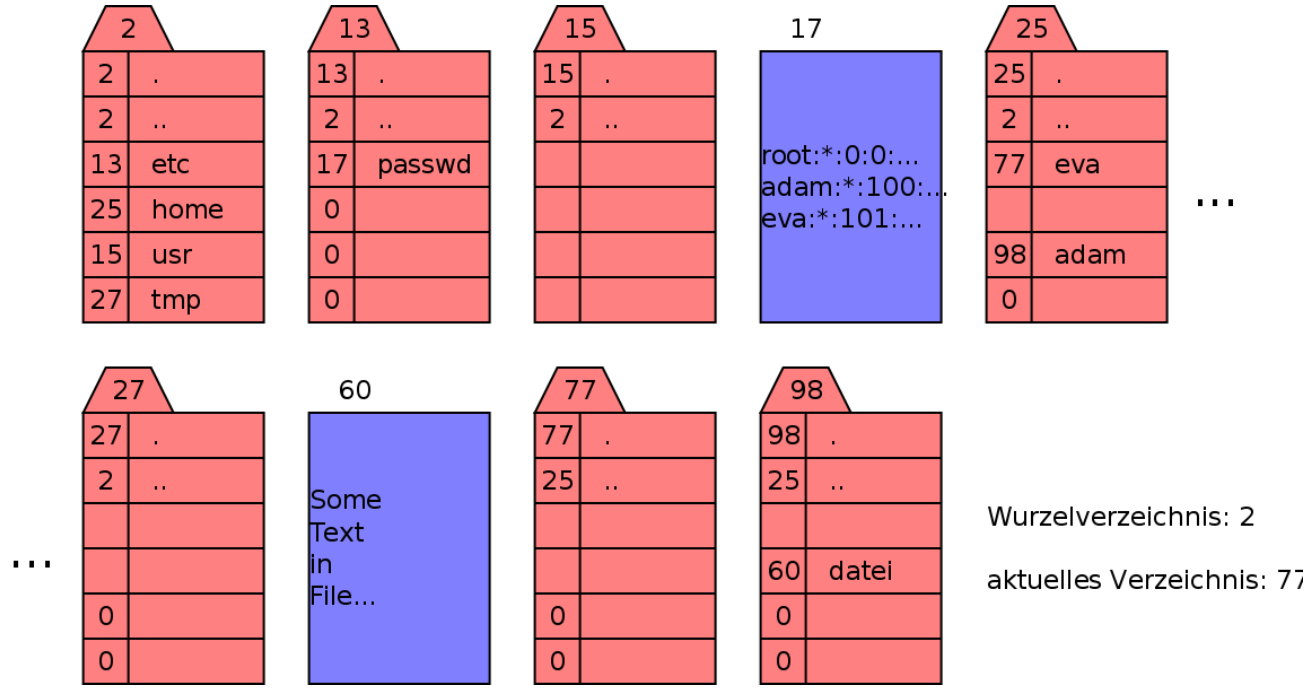
- Baumstruktur



- Pfade
  - z.B. `/home/adam/datei`, `/tmp`, `../adam/datei`
  - `/` ist Trennsymbol (*Slash*)
  - beginnender `/` ist Wurzelverzeichnis; sonst Beginn implizit mit dem aktuellen Verzeichnis

Pfadnamen (2)

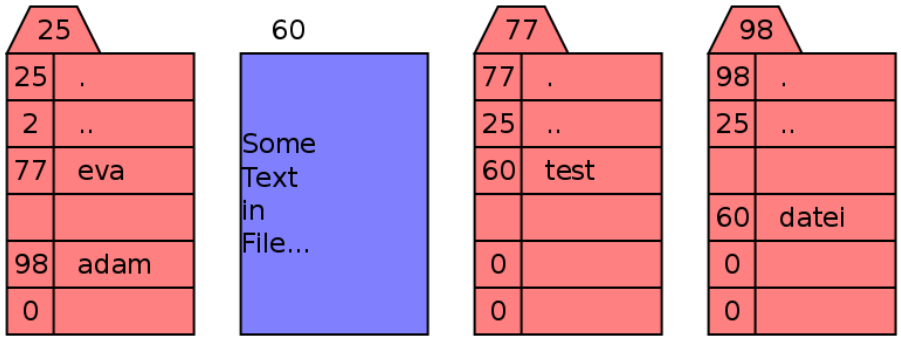
- eigentliche „Baumstruktur”



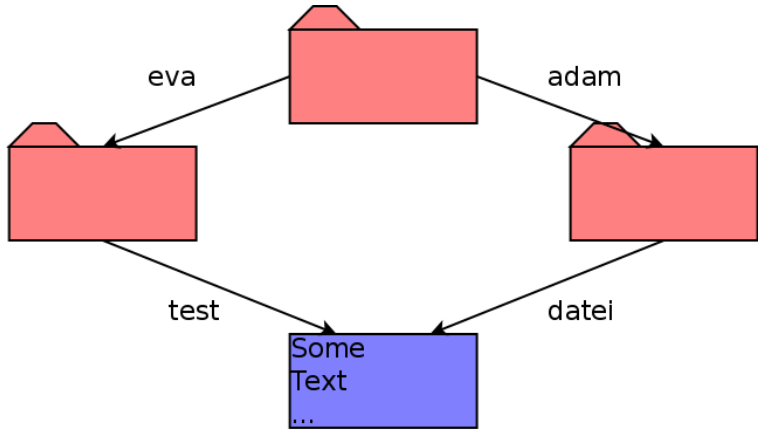
- Beispiel Pfadauflösung „`../adam/datei`”:
  - 77 + „`../adam/datei`” $\rightsquigarrow$ 25 + „`adam/datei`”
  - 25 + „`adam/datei`” $\rightsquigarrow$ 98 + „`datei`”
  - 98 + „`datei`” $\rightsquigarrow$ 60

Pfadnamen (3)

- Es können mehrere Verweise (**Hard Links**) auf eine Datei existieren:



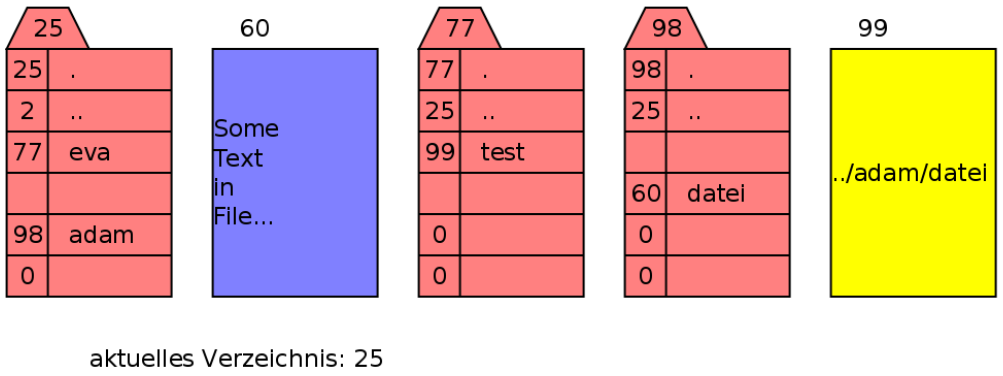
aktuelles Verzeichnis: 25

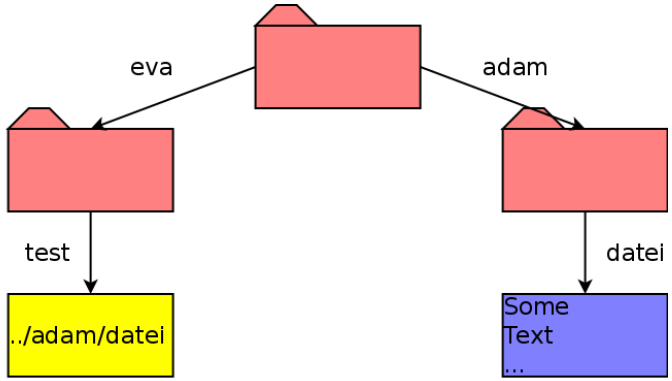


- Beispiel Pfadauflösung „`adam/datei`”:
  - 25 + „`adam/datei`” $\rightsquigarrow$ 98 + „`datei`”
  - 98 + „`datei`” $\rightsquigarrow$ 60
- Beispiel Pfadauflösung „`eva/test`”:
  - 25 + „`eva/test`” $\rightsquigarrow$ 77 + „`test`”
  - 77 + „`test`” $\rightsquigarrow$ 60
- Datei wird gelöscht, wenn keine Verweise auf sie mehr existieren

Pfadnamen (4)

- Es können mehrere symbolische Verweise (**Symbolic Links**) auf eine Datei oder ein Verzeichnis existieren:





- Beispiel Pfadauflösung „`eva/test`”:
  - 25 + „`eva/test`” $\rightsquigarrow$ 77 + „`test`”
  - 77 + „`test`” $\rightsquigarrow$ 99 $\rightsquigarrow$ 77 + „`../adam/datei`”
  - 77 + „`../adam/datei`” $\rightsquigarrow$ 25 + „`adam/datei`”
  - 25 + „`adam/datei`” $\rightsquigarrow$ 98 + „`datei`”
  - 98 + „`datei`” $\rightsquigarrow$ 60
- Symbolischer Name kann auch bestehen, wenn Datei oder Verzeichnis noch nicht bzw. nicht mehr existiert.

Eigentümer und Rechte

- Eigentümer
  - Jeder Eigentümer wird durch eindeutige Nummer (UID) repräsentiert
  - Ein Benutzer kann einer oder mehreren Benutzergruppen angehören, die jeweils durch eine eindeutige Nummer (GID) repräsentiert werden
  - Eine Datei oder ein Verzeichnis ist genau einem Benutzer und einer Gruppe zugeordnet
- Rechte auf Dateien
  - Lesen, Schreiben, Ausführen (nur vom Eigentümer änderbar)
  - Einzeln für den Eigentümer, für Angehörige der Gruppe und für alle anderen einstellbar
- Rechte auf Verzeichnissen
  - Lesen, Schreiben (Anlegen und Löschen von Dateien/Verzeichnissen), Durchgangsrecht
  - Schreibrecht ist einschränkbar auf eigene Dateien

Inodes

- Attribute (Zugriffsrechte, Eigentümer, usw.) einer Datei, eines Verzeichnisses werden in **Inodes** gespeichert (vereinfacht):

```
int st_mode;        /* Typ und Zugriffrechte */
int st_nlink;       /* Anzahl der Hard Links */
int st_uid;         /* Eigentuemer */
int st_gid;         /* Gruppe */
long st_size;       /* Laenge der Datei in Bytes */
int st_block[...]; /* Liste der (indirekten) Bloecke */
time_t st_atime;    /* Letzter Lesezeitpunkt */
time_t st_mtime;    /* Letzter Modifikationszeitpunkt */
time_t st_ctime;    /* Letzte Aenderung an Attributen */
```

- Jede Inode hat eine Nummer und einen Speicherort (Platte/Partition):

```
int st_ino;         /* Inode-Nummer */
int st_dev;         /* Platte/Partition-Nummer */
```


Programmierschnittstelle für Inodes

- `stat`, `lstat` liefern Dateiattribute aus Inodes
- Funktionsschnittstelle:

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

- Argumente:
  - `path`: Pfadname
  - `buf`: Zeiger auf Puffer, in den Inode-Informationen eingetragen werden
- Rückgabewert
  - 0, wenn OK
  - -1, wenn Fehler (`errno`-Variable enthält Fehlernummer)
- Beispiel:

```
struct stat buf;
stat("/etc/passwd", &buf); /* Fehlerbehandlung...! */
printf("Inode-Nummer: %d\n", buf.st_ino);
```


Programmierschnittstelle für Verzeichnisse

- Verzeichnisse, Links verwalten
  - Erzeugen (eines leeren Verzeichnisses)

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

  - Löschen (eines leeren Verzeichnisses)

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

  - Hard Link anlegen

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

  - Symbolischen Link anlegen

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

  - Link löschen (und damit ggf. auch Datei)

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

  - Symbolischen Link auslesen

```
int readlink(const char *path, char *buf, int size);
```


Programmierschnittstelle für Verzeichnisse (2)

- Verzeichnisse lesen (Schnittstelle des Linux-Kerns)
  - open(2), getdents(2), close(2)
  - Linux-spezifisch und damit nicht portabel
- Verzeichnisse lesen (Schnittstelle der C-Bibliothek)
  - Verzeichnis öffnen

```
DIR *opendir(const char *path);
```

  - einen Eintrag lesen

```
struct dirent *readdir(DIR *dirp);
```

  - Verzeichnis schließen

```
int closedir(DIR *dirp);
```

  - Struktur `struct dirent` (vereinfacht)

```
struct dirent {
    int d_ino;                  /* Inode-Nummer */
    char d_name[NAME_MAX + 1]; /* Name */
};
```


Verzeichnisse (3): `opendir`/`closedir`

- Funktionsschnittstelle:

```
#include <sys/types.h>
#include <dirent.h>

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

- Argument von `opendir`:
  - `path`: Verzeichnisname
- Rückgabewert von `opendir`:
  - Zeiger auf Datenstruktur vom Typ `DIR`, wenn OK
  - `NULL`, wenn Fehler (`errno`-Variable enthält Fehlernummer)

Verzeichnisse (4): `readdir`

- Funktionsschnittstelle:

```
#include <sys/types.h>
#include <dirent.h>

struct dirent *readdir(DIR *dirp);
```

- Argument:
  - `dirp`: Zeiger auf `DIR`-Datenstruktur
- Rückgabewert:
  - Zeiger auf Datenstruktur vom Typ `struct dirent`, wenn OK
  - NULL, wenn Verzeichnis zu Ende gelesen wurde (`errno`-Variable nicht verändert)
  - NULL, wenn Fehler aufgetreten ist (`errno`-Variable enthält Fehlercode)
- Hinweis: Der Speicher für `struct dirent` wird u.U. beim nächsten `readdir`-Aufruf überschrieben!

Verzeichnisse (5): Beispiel

- Ausgabe der Dateinamen im aktuellen Verzeichnis:

```
#include <sys/types.h>
#include <dirent.h>

DIR *dirp;
struct dirent *de;
int ret;

dirp = opendir(".");                    // akt. Verz. oeffnen
if (dirp == NULL) ...                   // Fehler

while (1) {
  errno = 0;
  de = readdir(dirp);                   // Eintrag lesen
  if (de == NULL && errno != 0) ...     // Fehler
  if (de == NULL) break;                // Ende erreicht

  printf("%s\n", de->d_name);
}

ret = closedir(dirp);                   // Verz. schliessen
if (ret < 0) ...                        // Fehler
```

Programmierschnittstelle für Dateien

- Funktionsschnittstelle:

```
#include <sys/types.h>
#include <fcntl.h>

int open(const char *path, int flags, ...);

int close(int fd);

ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

Dateien (2): Beispiel

- Kopierprogramm

```
#include <fcntl.h>

int ret;

int src_fd = open("src", O_RDONLY);
if (src_fd < 0) ...          // Fehler
int dst_fd = open("dst", O_CREAT | O_TRUNC | O_WRONLY, 0777);
if (dst_fd < 0) ...          // Fehler

while (1) {
  char buf[1024];
  len = read(src_fd, buf, sizeof(buf));
  if (len < 0) ...           // Fehler
  if (len == 0) break;
  ret = write(dst_fd, buf, len);
  if (ret < 0) ...           // Fehler
}

ret = close(dst_fd);
if (ret < 0) ...             // Fehler
ret = close(src_fd);
if (ret < 0) ...             // Fehler
```

Dateien (3)

- `write`-Aufruf muss
  - den File-Deskriptor überprüfen (Datei geöffnet, Datei beschreibbar?)
  - die Pufferadresse/-länge überprüfen
  - den/die zu beschreibenden Blöcke des Mediums ermitteln
  - den/die Blöcke vom Medium lesen (wenn nicht ganzer Block geschrieben wird)
  - die entsprechenden Bytes im gelesenen Block überschreiben
  - den/die Blöcke auf das Medium zurückübertragen
  - die Attribute anpassen (Datum letzte Modifikation, Länge der Datei)
  - und ist ein Betriebssystem-Aufruf
- => `write` ist eine zeitlich teure Operation (`read` entsprechend)!
- => Besser: viele Bytes (am Besten: Vielfache der Blockgröße) am Stück lesen/schreiben
- => `fopen`-, `fclose`-, `fread`-, `fwrite`-, `getchar`-, `putchar`-, `fscanf`-, `fprintf`-, ... -Funktionen aus der C-Bibliothek benutzen!

Spezialdateien

- Periphere Geräte (Platte, Drucker, CD, Terminal, Scanner, ...) werden als Spezialdateien repräsentiert (`/dev/sda`, `/dev/lp0`, `/dev/cdrom0`, `/dev/tty`, ...)
- in Inode steht
  - Typ:
    - Block-orientiertes Gerät (Platte, CD, DVD, SSD, ...)
    - Zeichen-orientiertes Gerät (Drucker, Terminal, Scanner, ...)
  - statt Blocknummern:
    - Major-Number: Typ des Gerätes (Platte, Drucker, ...)
    - Minor-Number: Nummer des Gerätes (3. Drucker, 5. Terminal, ...)
- Öffnen der Geräte schafft eine (evtl. exklusive) Verbindung zum Gerät, die durch Treiber hergestellt wird
- Geräte können dann mit `read`-, `write`- und `ioctl`-Operationen angesprochen werden

Spezialdateien (2): Beispiel

- Ausgabe auf Drucker

```
#include <linux/lp.h>
int fd, ret;

/* Verbindung zum Drucker 0 herstellen. */
fd = open("/dev/lp0", O_WRONLY);
if (fd < 0) ...

/* Druckerstatus abfragen. */
ret = ioctl(fd, LPGETSTATUS, &state);
if (ret < 0) ...
if (state & LP_POUTPA) {
  fprintf(stderr, "Out of paper!\n"); exit(1);
}

/* Auf Drucker schreiben. */
ret = write(fd, "Hallo, Drucker!\n\f", 17);
if (ret < 0) ...

/* Verbindung abbauen. */
ret = close(fd);
if (ret < 0) ...
```


Partitionen

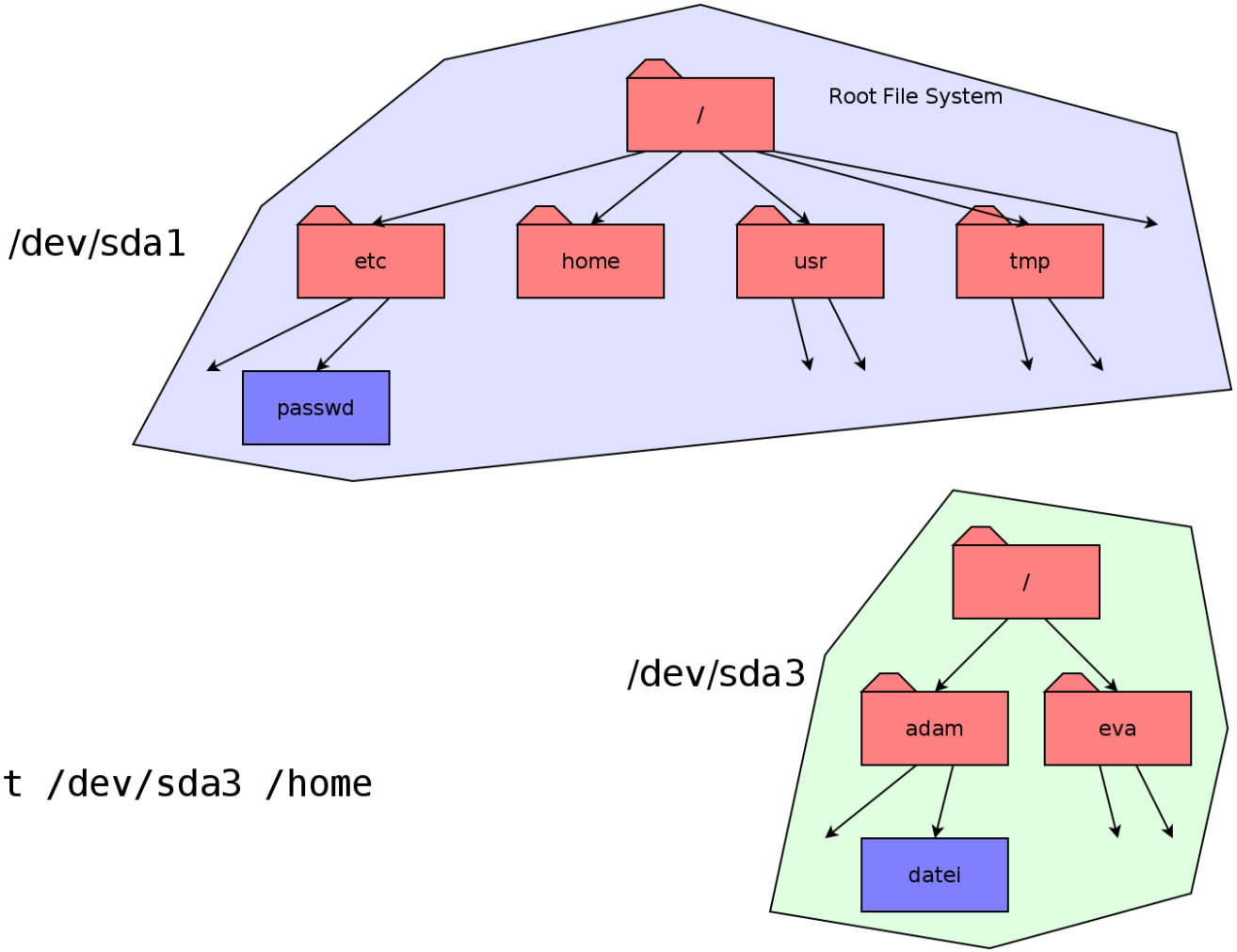
- jede Festplatte kann als Ganzes ein Dateisystem enthalten
  - Festplatte entspricht dann einer Partition
- jede Festplatte kann aber auch unterteilt werden in mehrere Partitionen
  - erster Block der Platte enthält Partitionstabelle
  - Partitionstabelle enthält Informationen
    - wieviele Partitionen existieren
    - wie groß die jeweiligen Partitionen sind
    - wo sie beginnen
- jede Partition
  - wird durch eine Spezialdatei repräsentiert; z.B.
    - `/dev/sda`, `/dev/sdb` (ganze Platte)
    - `/dev/sda1`, `/dev/sda2`, `/dev/sdb1` (Teile der jeweiligen Platte)
  - enthält eigenes Dateisystem

Partitionen (2)

- Bäume der Partitionen können zu einem homogenen Dateibaum zusammengesetzt werden (Grenzen für Anwender nicht sichtbar!)
  - „Montieren" von Dateibäumen (*mounting*)
- Ein ausgezeichnetes Dateisystem ist das *Root File System*, dessen Wurzelverzeichnis gleichzeitig das Wurzelverzeichnis des Gesamtsystems ist
  - Andere Dateisysteme können mit dem **mount**-Befehl in das bestehende System hineinmontiert werden bzw. mit dem **umount**-Befehl wieder entfernt werden.
  - Über das *Network File System* (NFS) können auch Verzeichnisse anderer Rechner in einen lokalen Dateibaum hineinmontiert werden => Grenzen zwischen Dateisystemen verschiedener Rechner werden unsichtbar

Montieren des Dateibaumes

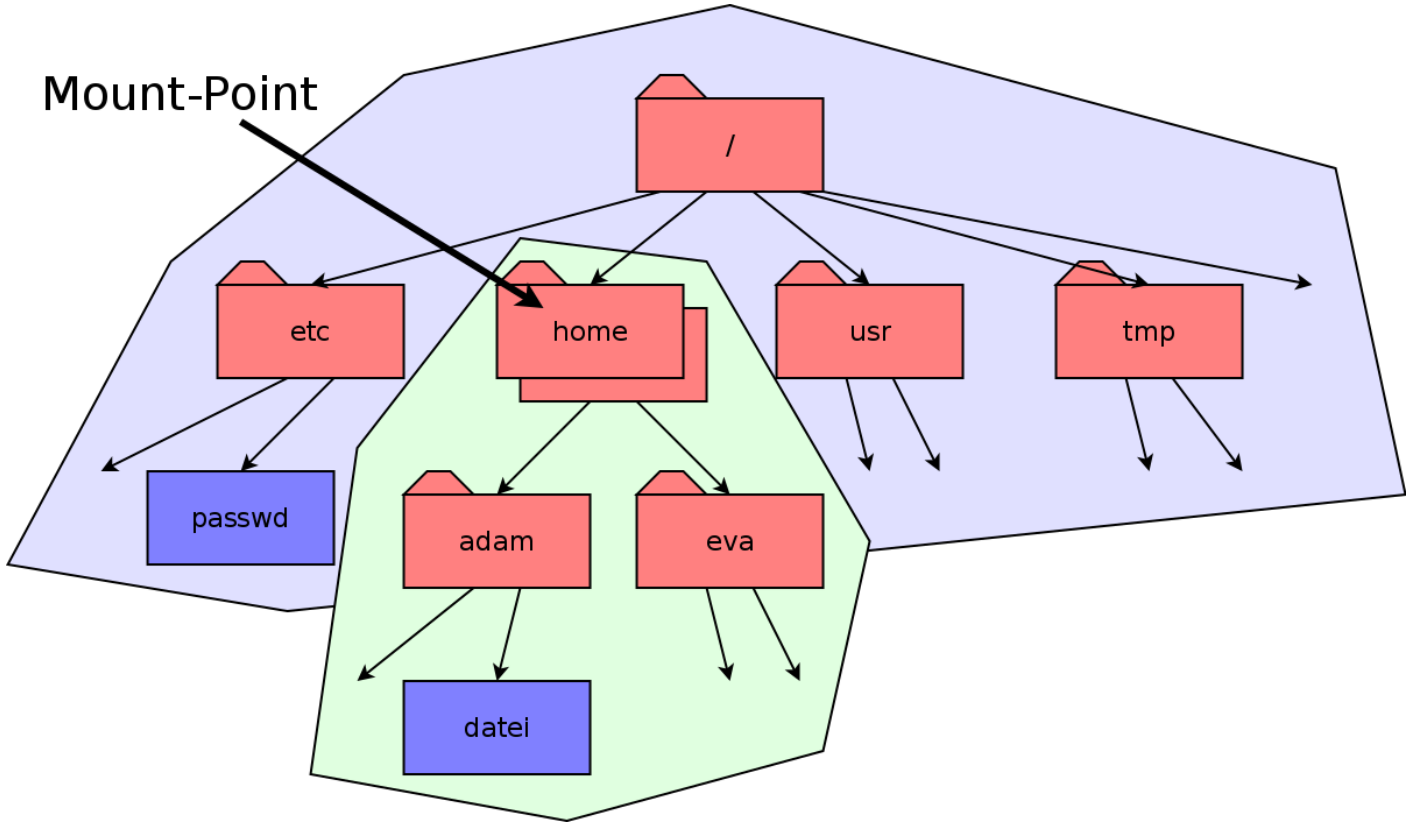
- Beispiel




```
mount /dev/sda3 /home
```

Montieren des Dateibaumes (2)

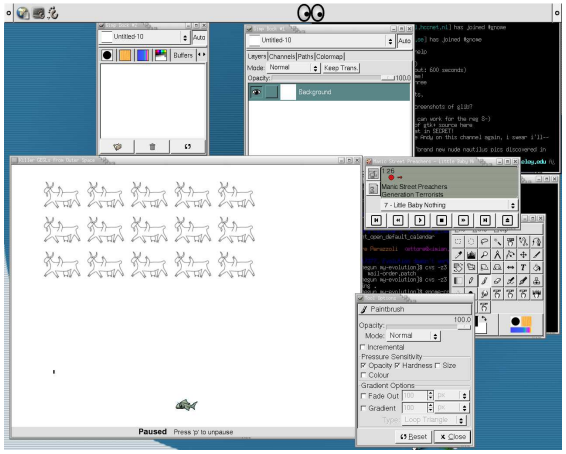
- Nach Ausführung des Montierbefehls



Überblick: Teil D Betriebssystemabstraktionen

Überblick

- **Mehrere** Programme, die
- **nebenläufig**,
- **dynamisch** gestartet/beendet
- über **definierte E/A-Funktionen**
- ihre Umgebung steuern.



Quelle: www.wikipedia.org

Jedes laufende Programm bekommt Hardware zugeteilt:

- CPU (Zeitanteile)
- Speicher (Teil des Gesamtspeichers)

und kann Betriebssystem-Kern-Funktionen aufrufen.

Definitionen

Programm: Folge von Anweisungen

Prozess: laufendes Programm mit seinen Daten

Hinweis: ein Programm kann sich mehrfach in Ausführung befinden!

Prozesse

- Definition „Prozess”: laufendes Programm mit seinen Daten
- eine etwas andere Sicht:

Prozessor	Zeitanteile am echten Prozessor
Speicher	virtueller Speicher
Interrupts	Signale
E/A-Geräte	E/A-Betriebssystem-Funktionen

Prozesse (2)

- Mehrprogrammbetrieb („Multitasking”)
  - mehrere Prozesse können quasi gleichzeitig ausgeführt werden
  - stehen weniger Prozessoren zur Verfügung, als Prozesse ausgeführt werden sollen, werden Zeitanteile der Rechenzeit an die Prozesse vergeben (**Time Sharing System**)
  - die Entscheidung, welcher Prozess zu welchem Zeitpunkt wieviel Rechenzeit bekommt, trifft der Betriebssystem-Kern (**Scheduler**)
  - die Umschaltung zwischen Prozessen erfolgt durch den Betriebssystem-Kern (**Dispatcher**)
  - laufende Prozesse wissen nicht, an welchen Stellen auf andere Prozesse umgeschaltet wird

Prozesszustände

Ein Prozess befindet sich in einem der folgenden Zustände

Erzeugt: (New)
Prozess wurde erzeugt, besitzt aber noch nicht alle zum Laufen notwendigen Betriebsmittel

Bereit: (Ready)
Prozess besitzt alle nötigen Betriebsmittel und ist bereit zu laufen

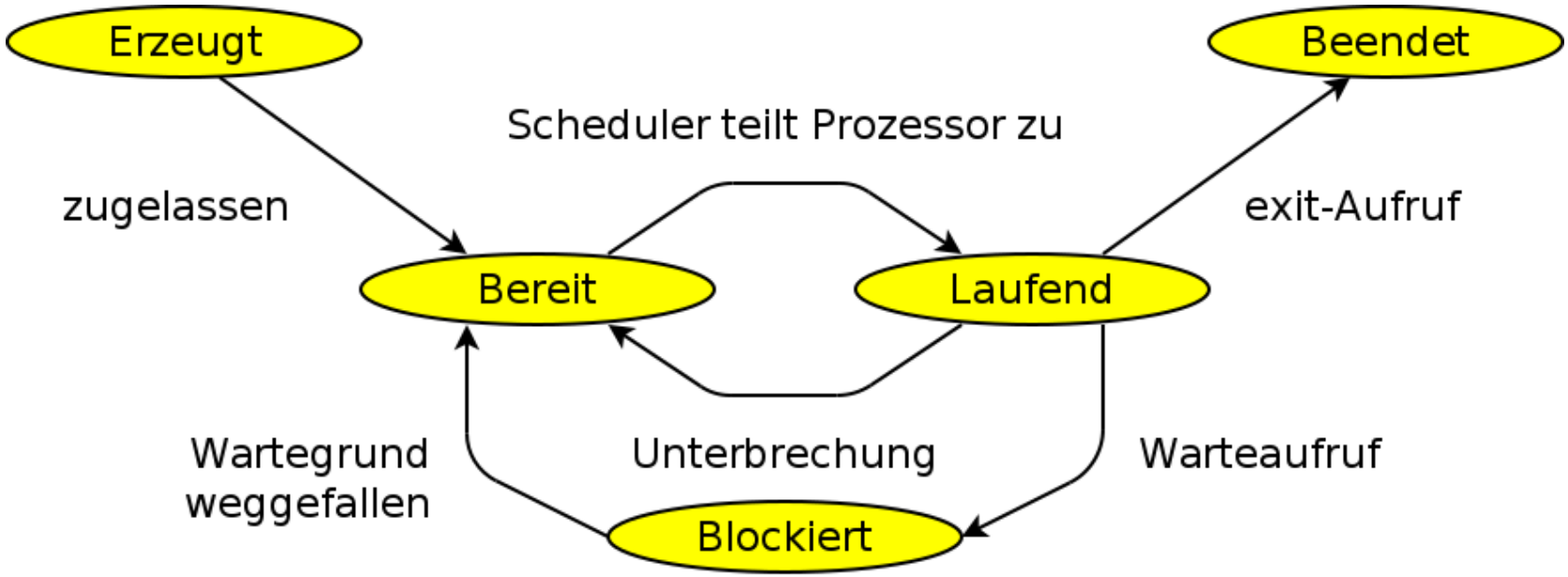
Laufend: (Running)
Prozess wird vom realen Prozessor ausgeführt

Blockiert: (Blocked)
Prozess wartet auf ein Ereignis (Fertigstellung einer Ein- oder Ausgabeoperation)

Beendet: (Terminated)
Prozess ist beendet, seine Betriebsmittel sind noch nicht alle freigegeben

Prozesszustände (2)

- Zustandsdiagramm mit Übergängen:



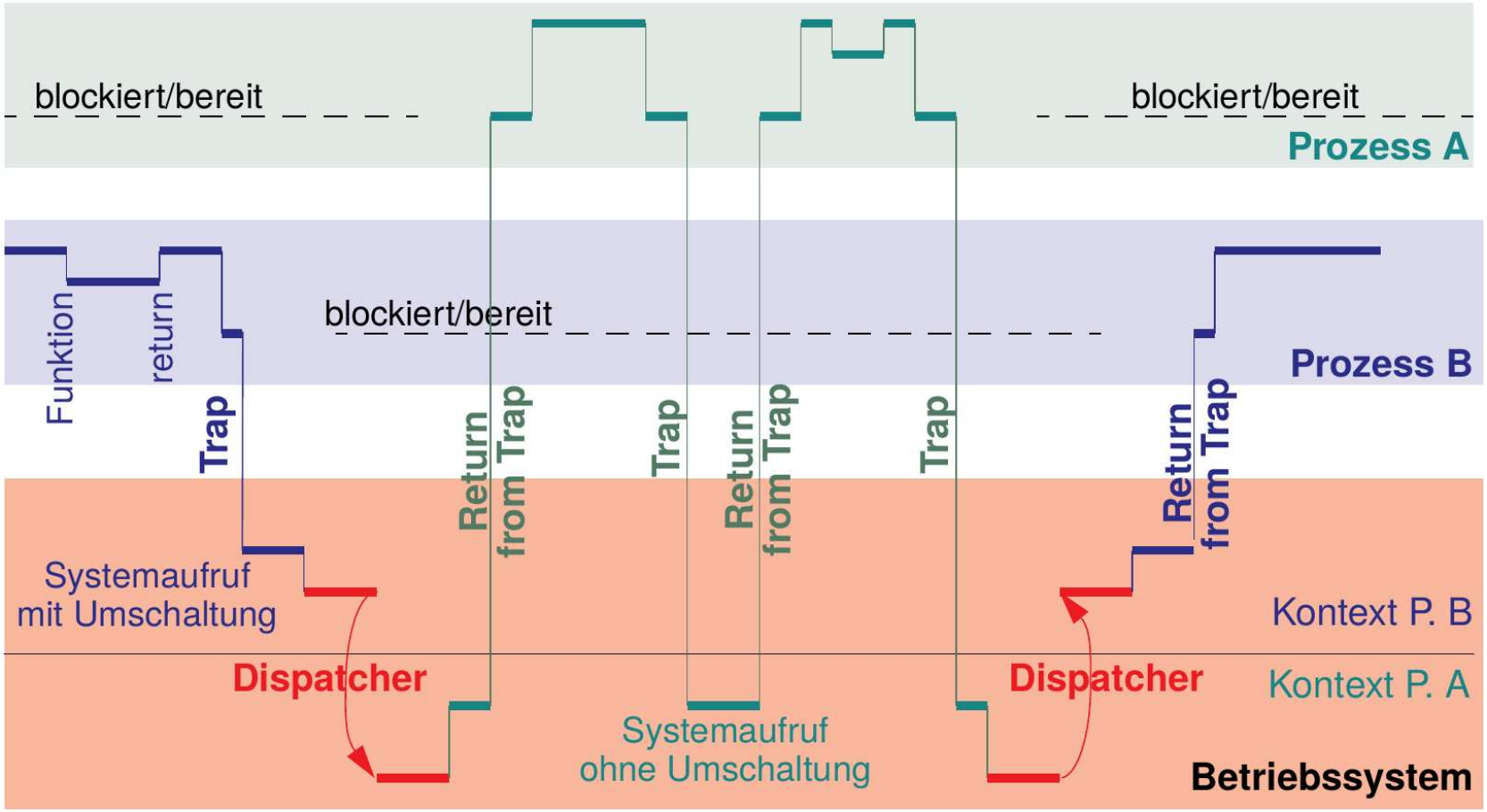
Nach Silberschatz, 1994

Prozesswechsel

- Jeder Prozess hat Zustand/Kontext
  - Registerinhalte des Prozessors
  - Inhalte der Speicherbereiche
  - offene Dateien, aktuelles Verzeichnis, ...
- Beim Prozesswechsel (Context Switch)
  - wird der Inhalt der Prozessorregister abgespeichert,
  - ein neuer Prozess ausgewählt,
  - die Ablaufumgebung des neuen Prozesses hergestellt
    - Umprogrammierung der MMU
    - Wechsel der offenen Dateien, des aktuellen Verzeichnisses, ...
  - werden die gesicherten Register des neuen Prozesses geladen.

Prozesswechsel

- Ablauf von zwei Prozessen in Benutzermodus und Kern mit Umschaltung



Prozesskontrollblock

- Prozesskontrollblock (Process Control Block – PCB)

Datenstruktur des Betriebssystem-Kerns, die alle notwendigen Daten für einen Prozess enthält.

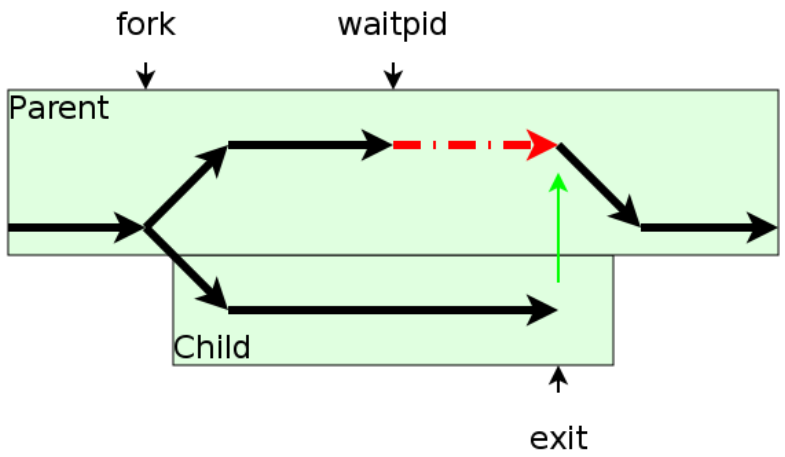
Beispiel UNIX:

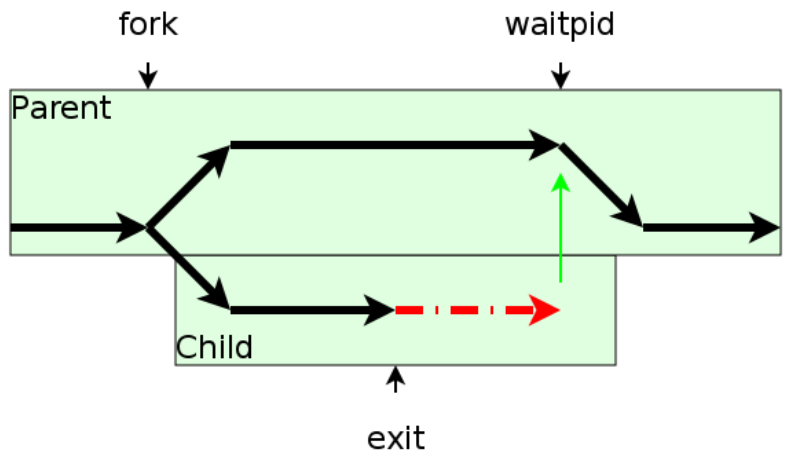
- Prozess-ID (PID)
- Prozesszustand (Laufend, Bereit, ...)
- Register
- Speicherabbildung
- Eigentümer (UID, GID)
- Wurzelverzeichnis, aktuelles Verzeichnis
- offene Dateien
- ...

Beispiel: UNIX

- Überblick:

fork: Erzeugung von neuen Prozessen
exit: Terminieren von Prozessen
waitpid: Warten auf das Terminieren von Prozessen





Programmierschnittstelle

- Duplizieren des gerade laufenden Prozesses

```
#include <unistd.h>

pid_t fork(void);
```

- Terminieren des aktuellen Prozesses

```
#include <stdlib.h>

void exit(int status);
```

- Warten auf das Terminieren eines anderen Prozesses

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status, int options);
```


Beispiel

```c
pid_t pid, ret;
int status;

pid = fork();
switch (pid) {
case -1: /* Error */
    perror("fork");
    exit(1);

case 0: /* Child */
    do_child_work();
    exit(13);

default: /* Parent */
    do_parent_work();
    ret = waitpid(pid, &status, 0);
    /*
     * In case of no error:
     * ret == pid
     * WIFEXITED(status) == 1
     * WEXITSTATUS(status) == 13
     */
    break;
}
```

Prozesserzeugung

- Der Kind-Prozess ist Kopie des Vater-Prozesses
  - gleiches Programm
  - gleiche Daten (Variablen-Inhalte)
  - gleicher Programmzähler
  - gleiches aktuelles Verzeichnis, Wurzelverzeichnis
  - gleiche geöffnete Dateien
- einzige Unterschiede
  - verschiedene Prozess-IDs
  - Rückgabewert von `fork`

Ausführung von Programmen

- Das von einem Prozess ausgeführte Programm kann durch ein neues Programm ersetzt werden:

```
#include <unistd.h>

int execv(const char *path, char *argv[]);
int execl(const char *path, char *arg0, ...);
```

Beispiel:

```
...                 /* Process A */
argv[0] = "ls";
argv[1] = "-l";
argv[2] = NULL;
execv("/bin/ls", argv);
/* Should not be reached. */
```

$\Rightarrow$

```
...                 /* Process A */
int
main(int argc, char *argv[])
{
    ...
}
```

Das zuvor laufende Programm wird beendet, das neue gestartet.

Es wird nur das Programm ausgetauscht.

Der Prozess läuft weiter!

Start eines Programms

- Beispiel: Start des Programms `./prog` mit Parametern `-a` und `-b`

... als Vordergrund-Prozess:

```
pid_t pid;

pid = fork();
switch (pid) {
case -1: /* Error */
    perror("fork");
    exit(EXIT_FAILURE);

case 0: /* Child */
    execl("./prog", "prog",
          "-a", "-b", NULL);
    perror("./prog");
    exit(EXIT_FAILURE);

default: /* Parent */
    waitpid(pid, NULL, 0);
    break;
}
```

... als Hintergrund-Prozess:

```
pid_t pid;

pid = fork();
switch (pid) {
case -1: /* Error */
    perror("fork");
    exit(EXIT_FAILURE);

case 0: /* Child */
    execl("./prog", "prog",
          "-a", "-b", NULL);
    perror("./prog");
    exit(EXIT_FAILURE);

default: /* Parent */
    /* No "waitpid" here! */
    break;
}
```


Signale

- Mikrocontroller kann auf nebenläufige Ereignisse (Interrupts) mit Interrupt-Service-Routinen reagieren.
- Ähnliches Konzept auf Prozess-Ebene: Signale

Signale (2)

Interrupt: asynchrones Signal aufgrund eines „externen” Ereignisses

- CTRL-C auf der Tastatur gedrückt
- Timer abgelaufen
- Kind-Prozess terminiert
- ...

Exception: synchrones Signal, ausgelöst durch die Aktivität des Prozesses

- Zugriff auf ungültige Speicheradresse
- Illegaler Maschinenbefehl
- Division durch 0
- Schreiben auf eine geschlossene Kommunikationsverbindung
- ...

Kommunikation: ein Prozess will einem anderen ein Ereignis signalisieren

Signale (3)

CTRL-C:

```
int main(void)
{
    while (1) {
    }
}
```

```
~> ./test
^C
~>
```

Illegaler Speicherzugriff:

```
int main(void)
{
    *(int *) 0 = 0;
    return 0;
}
```

```
~> ./test
Segmentation fault
~>
```

Inter-Prozess-Kommunikation:

```
int main(void)
{
    while (1) {
    }
}
```

```
~> ./test
Terminated
~>
```

```
~> killall test
~>
```

Reaktion auf Signale

abort:

erzeugt Core-Dump (Speicher- und Registerinhalte werden in Datei `./core` geschrieben) und beendet Prozess
Standardeinstellung für alle Exceptions (zum nachträglichen Debuggen)

exit:

beendet Prozess (ohne Core-Dump)
Standardeinstellung für z.B. CTRL-C, Kill-Signal

ignore:

Signal wird ignoriert
Standardeinstellung für alle „unwichtigen" Signale (z.B. Kindprozess terminiert, Größe des Terminal-Fensters hat sich geändert)

...

Reaktion auf Signale (2)

...

handler:
: Aufruf einer Signalbehandlungsfunktion, danach Fortsetzung des Prozesses
: nie Standardeinstellung, da Programm-abhängig

stop:
: stoppt Prozess
: Standardeinstellung für Stop-Signal

continue:
: setzt Prozess fort
: Standardeinstellung für Continue-Signal

Reaktion über System-Aufruf (`sigaction`) änderbar

Programmierschnittstelle

- Einstellen der Signalbehandlungsfunktion
  (entspricht dem Setzen der ISR-Funktion)

```
#include <signal.h>

int sigaction(int sig, struct sigaction *new, struct sigaction *old);
```

`struct sigaction` enthält:

```
void (*sa_handler)(int sig); /* handler function
                                or SIG_DFL or SIG_IGN */
sigset_t sa_mask;            /* list of blocked signals while
                                handler is executed */
int sa_flags;                /* ... */
```

- ...

Programmierschnittstelle

- ...
- Blockieren/Freigeben von Signalen
  (entsprich `cli()`, `sei()`)

```
#include <signal.h>

int sigprocmask(int how, sigset_t *nmask, sigset_t *omask);
```

  - `SIG_BLOCK`: angegebene Signale blockieren
  - `SIG_UNBLOCK`: angegebene Signale deblockieren
  - `SIG_SETMASK`: Signalmaske setzen
- Freigeben + Passives Warten auf Signal + wieder Blockieren
  (entspricht `sei(); sleep_cpu(); cli();`)

```
#include <signal.h>

int sigsuspend(sigset_t *mask);
```

- ...

Programmierschnittstelle

- ...
- Erstellen einer leeren Signal-Liste

```
#include <signal.h>

int sigemptyset(sigset_t *mask);
```

- Erstellen einer vollen Signal-Liste

```
int sigfillset(sigset_t *mask);
```

- Hinzufügen eines Signals zu einer Signal-Liste

```
int sigaddset(sigset_t *mask, int sig);
```

- Entfernen eines Signals aus einer Signal-Liste

```
int sigdelset(sigset_t *mask, int sig);
```


Programmierschnittstelle

- Typische Signale:

SIGSEGV: „Segmentation Fault” (ungültiger Speicherzugriff)
SIGINT: „Interrupt” (CTRL-C)
SIGALRM: „Alarm” (Timer abgelaufen)
SIGCHLD: „Child” (Kindprozess terminiert)
SIGTERM: „Terminate” (Abbruch des Prozesses; abfangbar)
SIGKILL: „Kill” (Abbruch des Prozesses; nicht abfangbar)

Signal-Beispiel 1

```c
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int main(void)
{
  // Call handler when
  // CTRL-C signal is received.
  struct sigaction sa;
  sa.sa_handler = handler;
  sigfillset(&sa.sa_mask);
  sa.sa_flags = 0;
  sigaction(SIGINT, &sa, NULL);

  for (int i = 0; ; i++) {
    printf("%d\n", i);
  }
  return 0;
}
```

```c
static void handler(int sig)
{
    char s[] = "CTRL-C!\n";
    write(STDOUT_FILENO,
          s, strlen(s));
}
```

(Fehlerbehandlung weggelassen...)

```
~> ./test
...
146431
146432
146433
14^CCTRL-C!
6434
146435
146436
...
~>
```


Signal-Beispiel 2

```c
int main(void)
{
  // Call handler when
  // timer signal is received.
  struct sigaction sa;
  sa.sa_handler = handler;
  sigfillset(&sa.sa_mask);
  sa.sa_flags = 0;
  sigaction(SIGALRM, &sa, NULL);

  // Send timer signal every sec.
  struct itimerval it;
  it.it_value.tv_sec  = 1;
  it.it_value.tv_usec = 0;
  it.it_interval.tv_sec  = 1;
  it.it_interval.tv_usec = 0;
  setitimer(ITIMER_REAL, &it, NULL);

  // Wait for timer ticks.
  sigset_t mask;
  sigemptyset(&mask);
  while (1) sigsuspend(&mask);
}
```

```c
static void handler(int sig)
{
  write(STDOUT_FILENO,
        "Tick\n", 5);
}
```

(Fehlerbehandlung weggelassen...)

```
~> ./test
Tick
Tick
Tick
^C
~>
```

Signal-Beispiel 3

```c
#include <signal.h>
#include <unistd.h>

int main(void)
{
  // Call handler when
  // I/O signal is received.
  struct sigaction sa;
  sa.sa_handler = handler;
  sigfillset(&sa.sa_mask);
  sa.sa_flags = 0;
  sigaction(SIGIO, &sa, NULL);

  // Send I/O signal when
  // STDIN can be read.
  int flags = fcntl(STDIN_FILENO,
      F_GETFL);
  flags |= O_ASYNC;
  fcntl(STDIN_FILENO, F_SETFL,
      flags);

  while (1) sleep(1);
}
```

```c
static void handler(int sig)
{
  char buf[256];
  int len;

  // Read chars from STDIN.
  len = read(STDIN_FILENO, buf,
          sizeof(buf));

  // Handle chars in buf.
  ...
}
```

(Fehlerbehandlung weggelassen...)

Signale und Nebenläufigkeit

- Signale erzeugen Nebenläufigkeit innerhalb von Prozessen
- resultierende Probleme völlig analog zu Nebenläufigkeit bei Interrupts auf einem Mikrocontroller

Nebenläufigkeitsbeispiel

```c
int main(void) {
  struct sigaction sa;
  struct itimerval it;
  /* Setup timer tick handler. */
  sa.sa_handler = tick;
  sa.sa_flags = 0;
  sigfillset(&sa.sa_mask);
  sigaction(SIGALRM, &sa, NULL);
  /* Setup timer. */
  it.it_value.tv_sec = 1;
  it.it_value.tv_usec = 0;
  it.it_interval.tv_sec = 1;
  it.it_interval.tv_usec = 0;
  setitimer(ITIMER_REAL, &it, NULL);
  /* Print time while working. */
  while (1) {
    int s = sec, m = min, h = hour;
    printf("%02d:%02d:%02d\n", h, m, s);
    do_work();
  }
}
```

↓ hier Signal

(Fehlerbehandlung weggelassen...)

```c
volatile int hour = 0;
volatile int min = 0;
volatile int sec = 0;

static void tick(int sig) {
  sec++;
  if (60 <= sec) {
    sec = 0; min++;
  }
  if (60 <= min) {
    min = 0; hour++;
  }
  if (24 <= hour) {
    hour = 0;
  }
}
```

```
~> ./test
...
23:59:59
00:59:59
00:00:00
...
```

← hier Problem!

Lösungen Nebenläufigkeitsbeispiel

1. Lösung

```
sigset_t nmask, omask;

/* Block SIGALRM. */
sigemptyset(&nmask);
sigaddset(&nmask, SIGALRM);
sigprocmask(SIG_BLOCK,
            &nmask, &omask);

/* Get current time. */
int s = sec, m = min, h = hour;

/* Restore signal mask. */
sigprocmask(SIG_SETMASK,
            &omask, NULL);

/* Print current time. */
printf("%02d:%02d:%02d\n",
       h, m, s);
```

2. Lösung

```
/* Get current time. */
int s, m, h;
do {
  s = sec;
  m = min;
  h = hour;
} while (s != sec
      || m != min
      || h != hour);

/* Print current time. */
printf("%02d:%02d:%02d\n",
       h, m, s);
```

Weitere Lösungen existieren...

Nebenläufigkeitsprobleme

- Zusätzliches Problem:
  interne Funktionsweise von Bibliotheksfunktionen i.A. unbekannt
- Beispiel 1:
  `printf` fügt Zeichen in Puffer ein
  => Nutzung von `printf` im Hauptprogramm *und* in Signal-Behandlungsfunktion u.U. gefährlich
- Beispiel 2:
  `malloc` durchsucht Liste nach freiem Speicherbereich; `free` fügt Block in Liste ein
  => Nutzung von `malloc`/`free` im Hauptprogramm *und* in Signal-Behandlungsfunktion u.U. gefährlich
- Lösung:
  - Signale während der Ausführung kritischer Bereiche blockieren oder
  - keine unbekannten Bibliotheksfunktionen aus Signal-Behandlungsfunktionen heraus aufrufen

Überblick: Teil D Betriebssystemabstraktionen

Speicherorganisation

```
int a;                          // a: global, uninitialized
int b = 1;                      // b: global, initialized
const int c = 2;                // c: global, const

void main() {
  static int s = 3;             // s: local, static, initialized
  int x, y;                     // x: local, auto; y: local, auto
  char* p = malloc( 100 );      // p: local, auto; *p: heap (100 byte)
}
```

Wo kommt der Speicher für diese Variablen her?

- **Statische Allokation** – Reservierung beim Übersetzen / Linken
  - Betrifft alle globalen/statischen Variablen, sowie den Code ↪ 12–5
  - Allokation durch Platzierung in einer Sektion

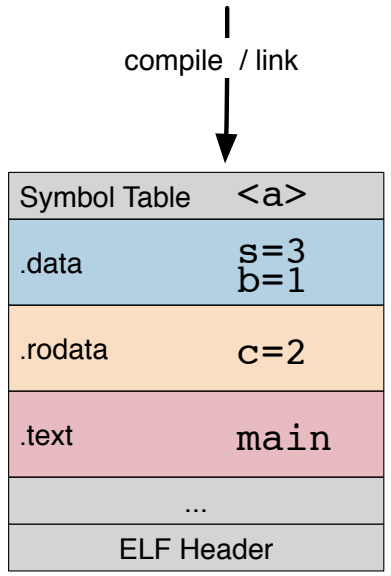
.text	– enthält den Programmcode	`main()`
.bss	– enthält alle mit 0 initialisierten Variablen	`a`
.data	– enthält alle mit anderen Werten initalisierten Variablen	`b,s`
.rodata	– enthält alle unveränderlichen Variablen	`c`

- **Dynamische Allokation** – Reservierung zur Laufzeit
  - Betrifft lokale auto-Variablen und explizit angeforderten Speicher

Stack	– enthält alle aktuell lebendigen auto-Variablen	`x,y,p`
Heap	– enthält explizit mit `malloc()` angeforderte Speicherbereiche	`*p`

Speicherorganisation auf einem μC

```c
int a;                          // a: global, uninitialized
int b = 1;                      // b: global, initialized
const int c = 2;                // c: global, const

void main() {
  static int s = 3;             // s: local, static, initialized
  int x, y;                     // x: local, auto; y: local, auto
  char* p = malloc( 100 );      // p: local, auto; *p: heap (100 byte)
}
```



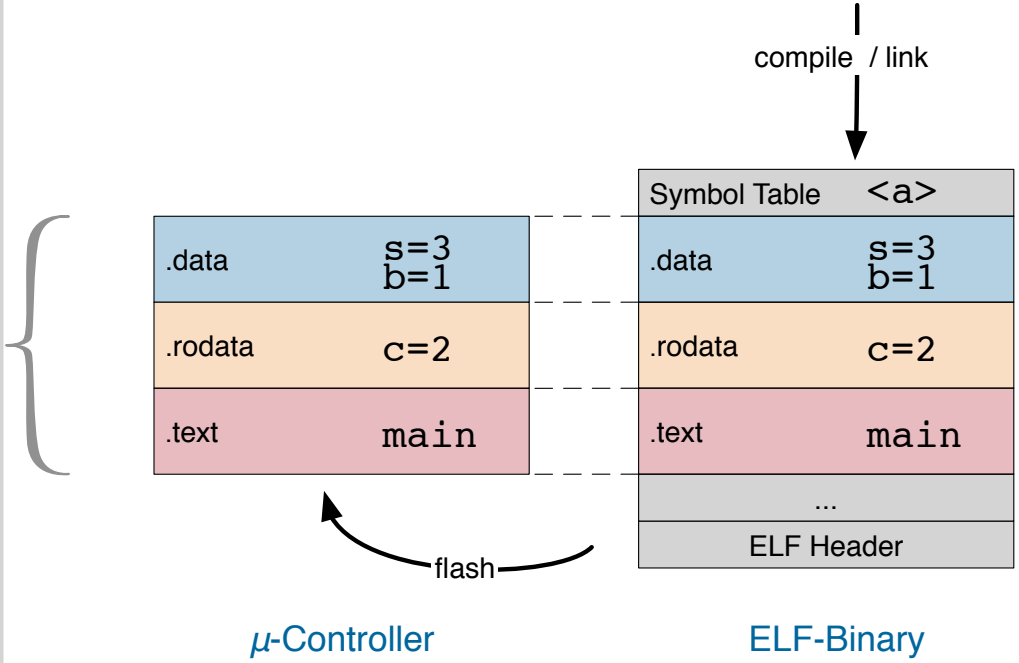
Quellprogramm

ELF-Binary

Beim Übersetzen und Linken werden die Programmelemente in entsprechenden Sektionen der ELF-Datei zusammen gefasst. Informationen zur Größe der `.bss`-Sektion landen ebenfalls in der Symboltabelle.

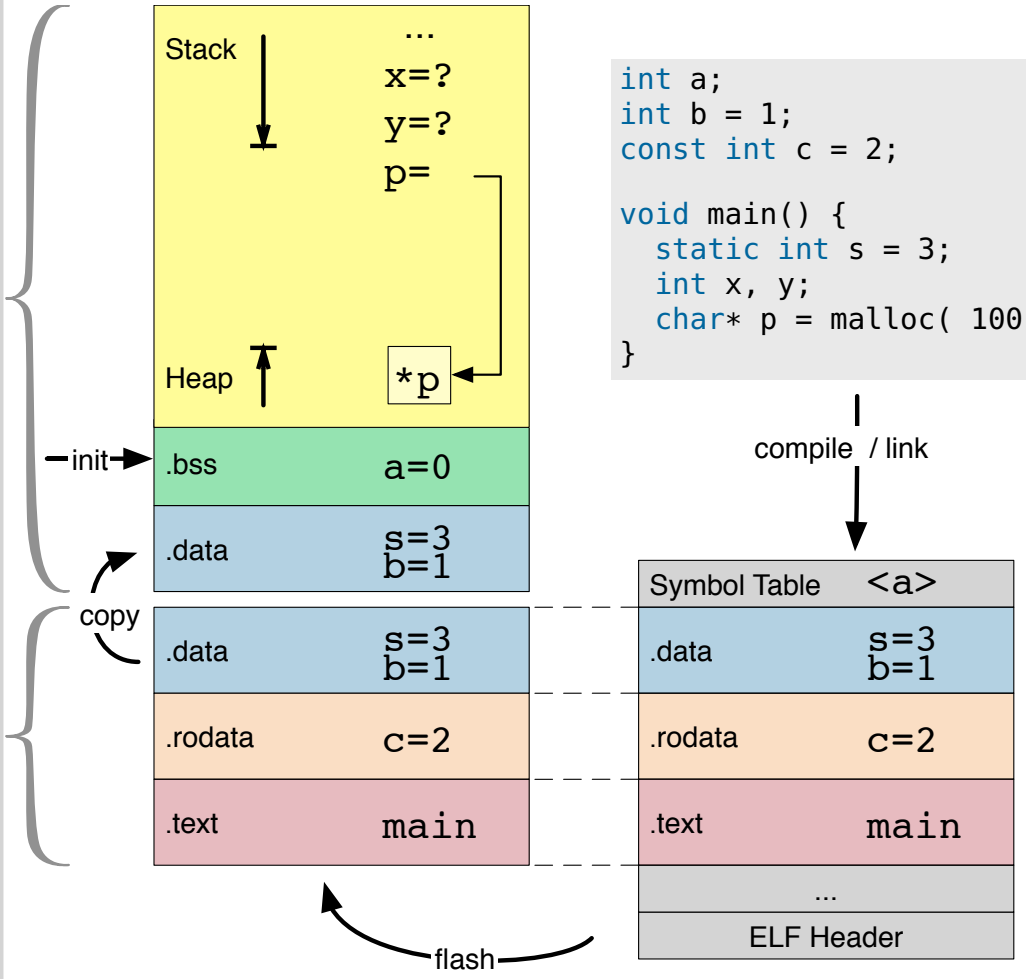
Speicherorganisation auf einem μC

```c
int a;                          // a: global, uninitialized
int b = 1;                      // b: global, initialized
const int c = 2;                // c: global, const

void main() {
  static int s = 3;             // s: local, static, initialized
  int x, y;                     // x: local, auto; y: local, auto
  char* p = malloc( 100 );      // p: local, auto; *p: heap (100 byte)
}
```



Zur Installation auf dem μC werden `.text` und `.[ro]data` in den Flash-Speicher des μC geladen.

Speicherorganisation auf einem μC




```c
int a;                          // a: global, uninitialized
int b = 1;                      // b: global, initialized
const int c = 2;                // c: global, const

void main() {
  static int s = 3;             // s: local, static, initialized
  int x, y;                     // x: local, auto; y: local, auto
  char* p = malloc( 100 );      // p: local, auto; *p: heap (100 byte)
}
```

Quellprogramm

Beim Systemstart wird das `.bss`-Segment im RAM angelegt und mit `0` initialisiert, das `.data`-Segment wird aus dem Flash ins RAM kopiert.
Das verbleibende RAM wird für den Stack und (falls vorhanden) den Heap verwendet.

Verfügt die Architektur über keinen Daten-Flashspeicher (beim ATmega der Fall ↪ 14–3), so werden konstante Variablen ebenfalls in `.data` abgelegt (und belegen zur Laufzeit RAM).

Speicherorganisation in einem UNIX-Prozess

- **Programm:** Folge von Anweisungen
- **Prozess:** Betriebssystemkonzept zur Ausführung von Programmen
  - Programm, das sich in Ausführung befindet, und seine Daten (Beachte: ein Programm kann sich mehrfach in Ausführung befinden)
  - Eine konkrete Ausführungsumgebung für ein Programm (Prozessor, Speicher, ...) $\mapsto$ vom Betriebssystem verwalteter *virtueller Computer*
- Jeder Prozess bekommt einen virtuellen Adressraum zugeteilt
  - 4 GB auf einem 32-Bit-System, davon bis zu 3 GB für die Anwendung
    - In das verbleibende GB werden Betriebssystem und *memory-mapped* Hardware (z. B. PCI-Geräte) eingeblendet
    - Daten des Betriebssystems werden durch Zugriffsrechte geschützt
  - Zugriff auf andere Prozesse ist nur über das Betriebssystem möglich
  - Virtueller Speicher wird durch das Betriebssystem auf physikalischen (Hintergrund-)Speicher abgebildet

Speicherorganisation in einem UNIX-Prozess (Forts.)

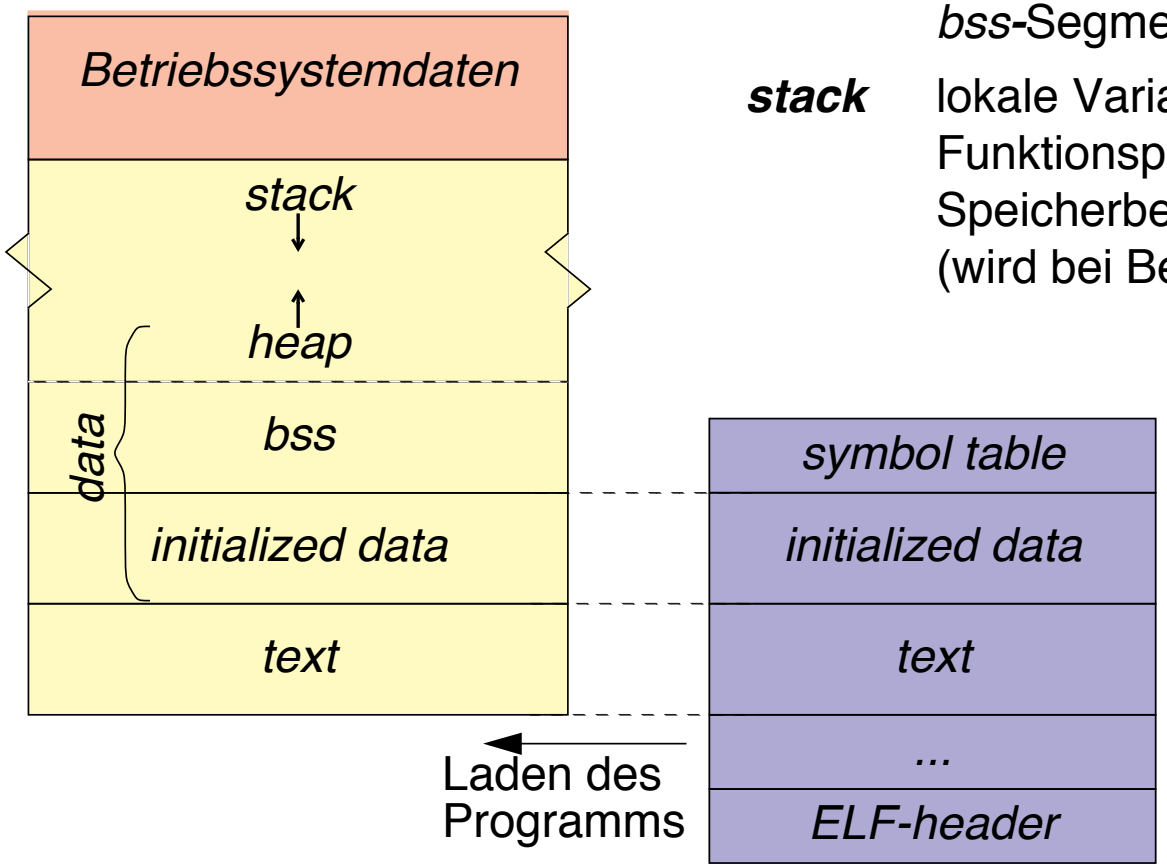
text Programmkode

data globale und static Variablen

bss nicht initialisierte globale und *static* Variablen (wird vor der Vergabe an den Prozess mit 0 vorbelegt)

heap dynamische Erweiterungen des *bss*-Segments (***sbrk(2)***, ***malloc(3)***)

stack lokale Variablen, Funktionsparameter, Speicherbereiche für Registerinhalte, (wird bei Bedarf dynamisch erweitert)

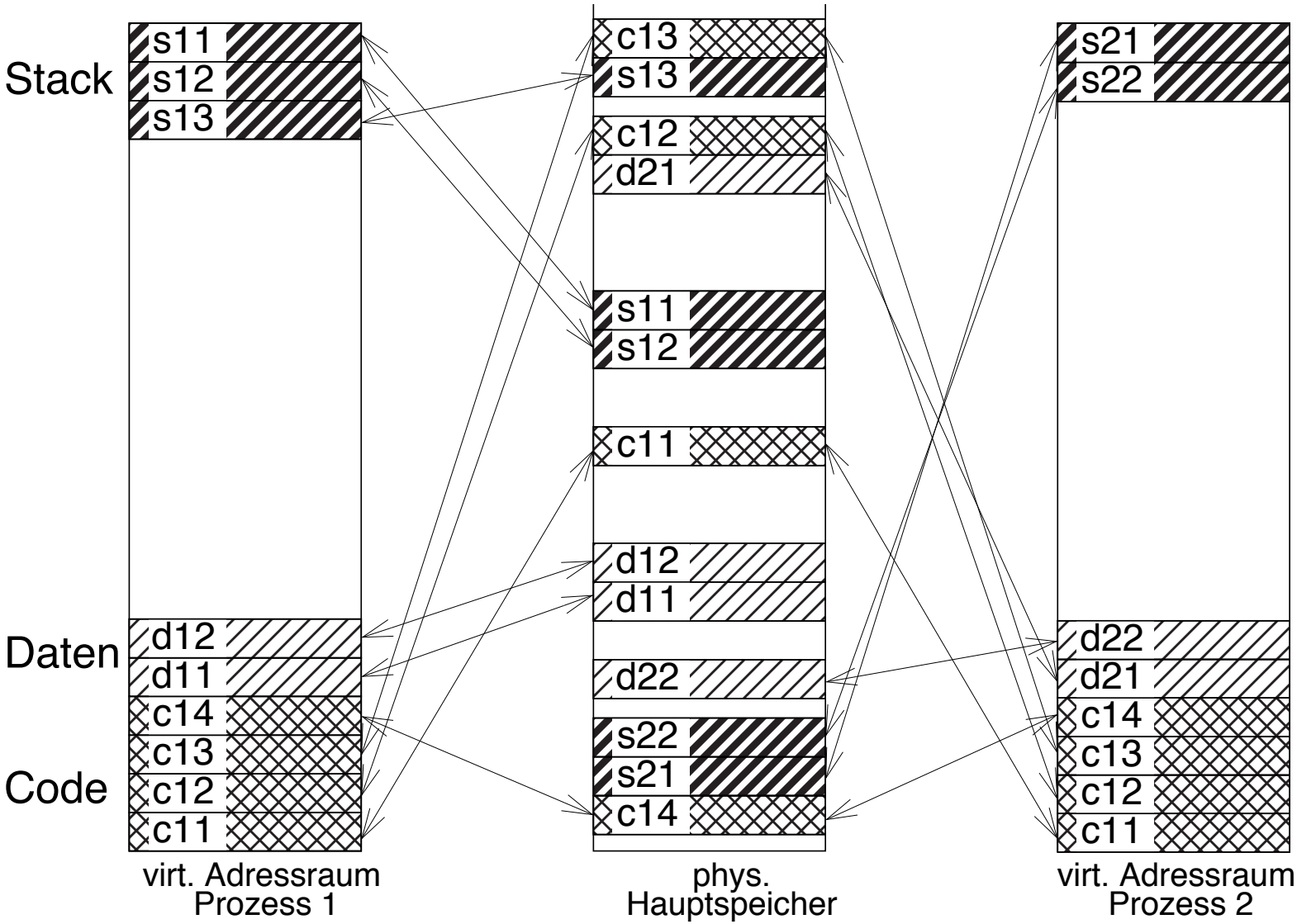


Aufbau eines Programms (ELF-Format)

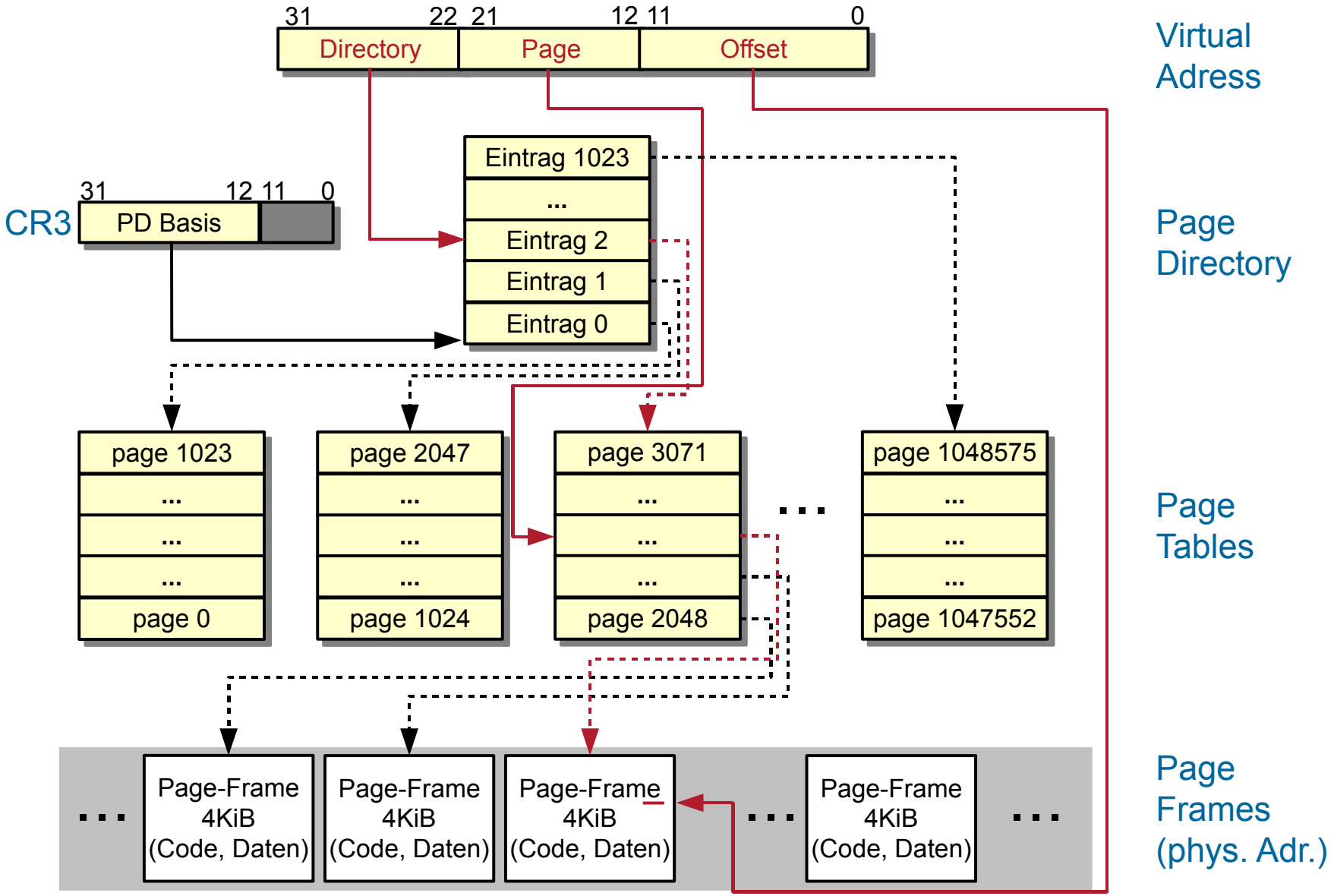
Seitenbasierte Speicherverwaltung

- Die Abbildung von virtuellem Speicher (*VS*) auf physikalischen Speicher (*PS*) erfolgt durch Seitenaddressierung (*Paging*)
  - *VS* eines Prozesses ist unterteilt in Speicherseiten (*Memory Pages*)
    - kleine Adressblöcke, üblich sind z. B. 4 KiB und 4 MiB Seiten
    - in dieser Granularität wird Speicher vom Betriebssystem zugewiesen
  - *PS* ist analog unterteilt in Speicherrahmen (*Page Frames*)
  - Abbildung: *Seite* $\mapsto$ *Rahmen* über eine Seitentabelle (*Page Table*)
    - Umrechnung *VS* auf *PS* bei jedem Speicherzugriff
    - Hardwareunterstützung durch MMU (*Memory Management Unit*)
    - Betriebssystem kann Seiten auf den Hintergrundspeicher auslagern
    - Abbildung ist nicht linkseindeutig: Seiten aus mehreren Prozesse können auf denselben Rahmen verweisen (z. B. gemeinsamer Programmcode)
- Seitenbasierte Speicherverwaltung ist auch ein Schutzkonzept
  - Seiten sind mit Zugriffsrechten versehen: *Read*, *Read–Write*, *Execute*
  - MMU überprüft bei der Umrechnung, ob der Zugriff erlaubt ist

Seitenbasierte Speicherverwaltung



Seitenbasierte Speicherverwaltung



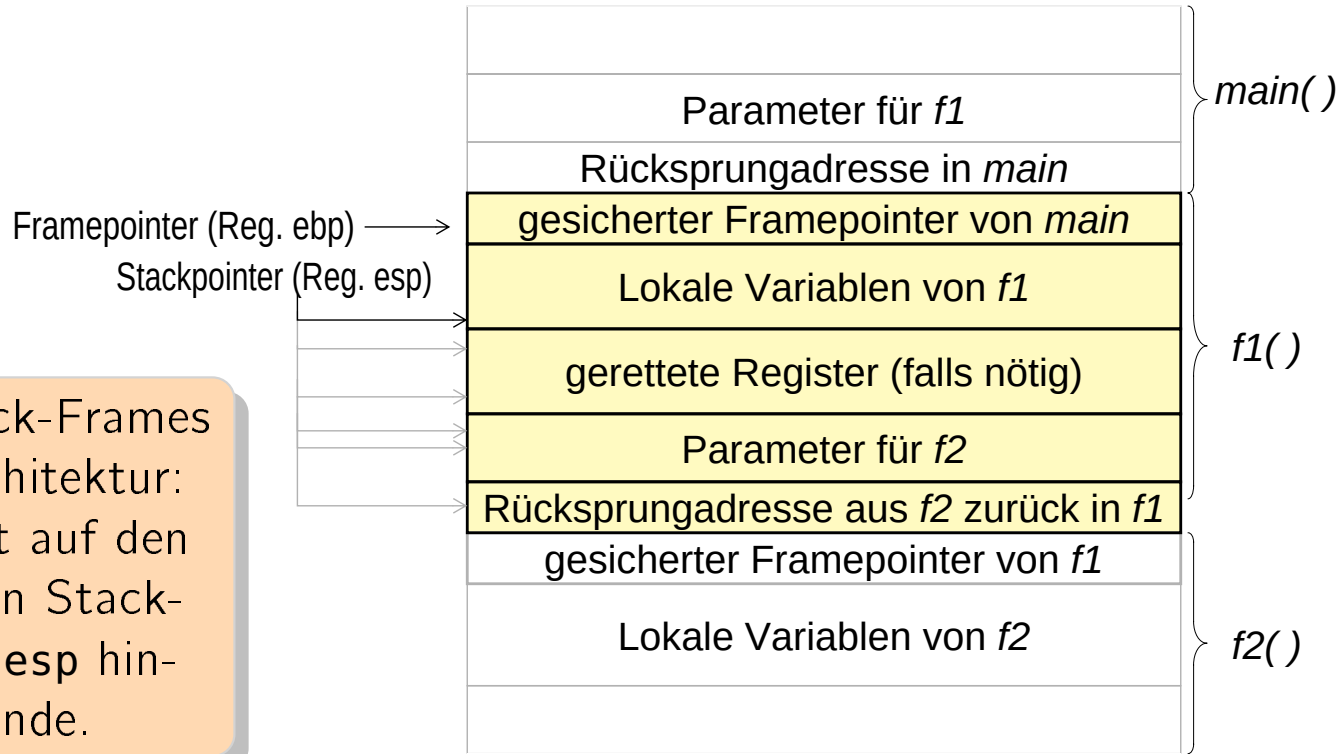
Dynamische Speicherallokation: Heap

- **Heap** := Vom Programm explizit verwalteter RAM-Speicher
  - Lebensdauer ist unabhängig von der Programmstruktur
- Anforderung und Wiederfreigabe über zwei Basisoperationen
  - `void* malloc( size_t n )` fordert einen Speicherblock der Größe *n* an; Rückgabe bei Fehler: 0-Zeiger (`NULL`)
  - `void free( void* pmem )` gibt einen zuvor mit `malloc()` angeforderten Speicherblock vollständig wieder frei
- Beispiel

```
#include <stdlib.h>
int* intArray( uint16_t n ) {    // alloc int[n] array
  return (int*) malloc( n * sizeof int );
}
void main() {
  int* array  = intArray(100);   // alloc memory for 100 ints
  if( array ) {                  // malloc() returns NULL on failure
    ...                          // if succeeded, use array
    array[99]   = 4711;
    ...
    free( array );               // free allocated block (** IMPORTANT! **)
  }
}
```


Dynamische Speicherallokation: Stack

- Lokale Variablen, Funktionsparameter und Rücksprungadressen werden vom Übersetzer auf dem Stack (Stapel, Keller) verwaltet
  - Prozessorregister `[e]sp` zeigt immer auf den nächsten freien Eintrag
  - Stack „wächst“ (architekturabhängig) „von oben nach unten“
- Die Verwaltung erfolgt in Form von Stack-Frames

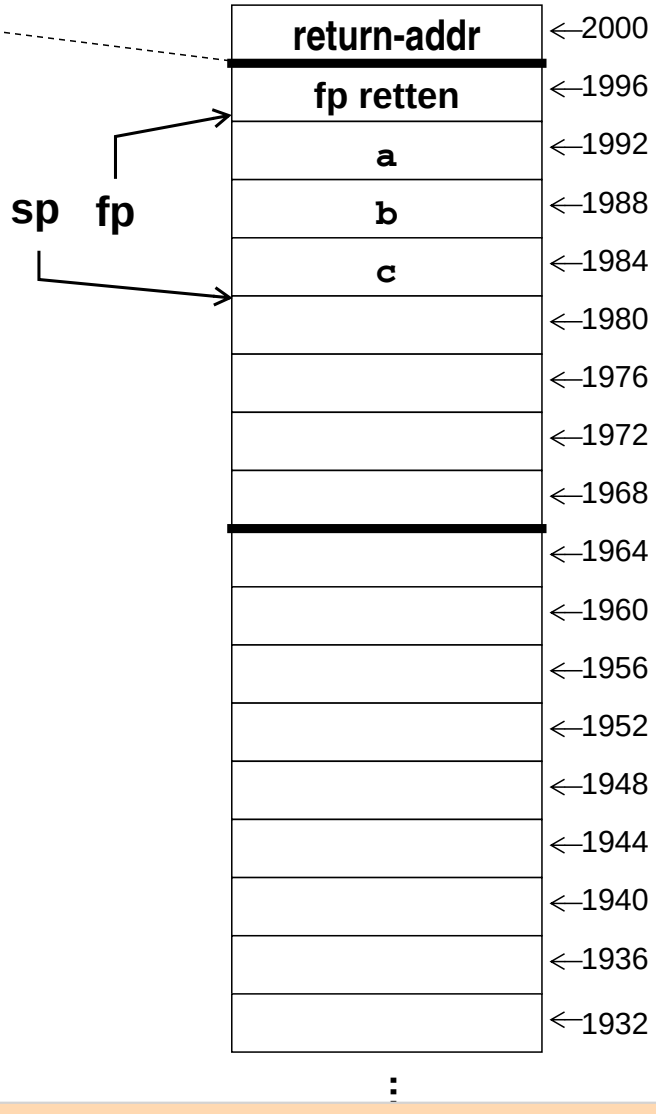


Aufbau eines Stack-Frames auf der IA-32-Architektur: Register **ebp** zeigt auf den Beginn des aktiven Stack-Frames; Register **esp** hinter das aktuelle Ende.

Stack-Aufbau bei Funktionsaufrufen

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

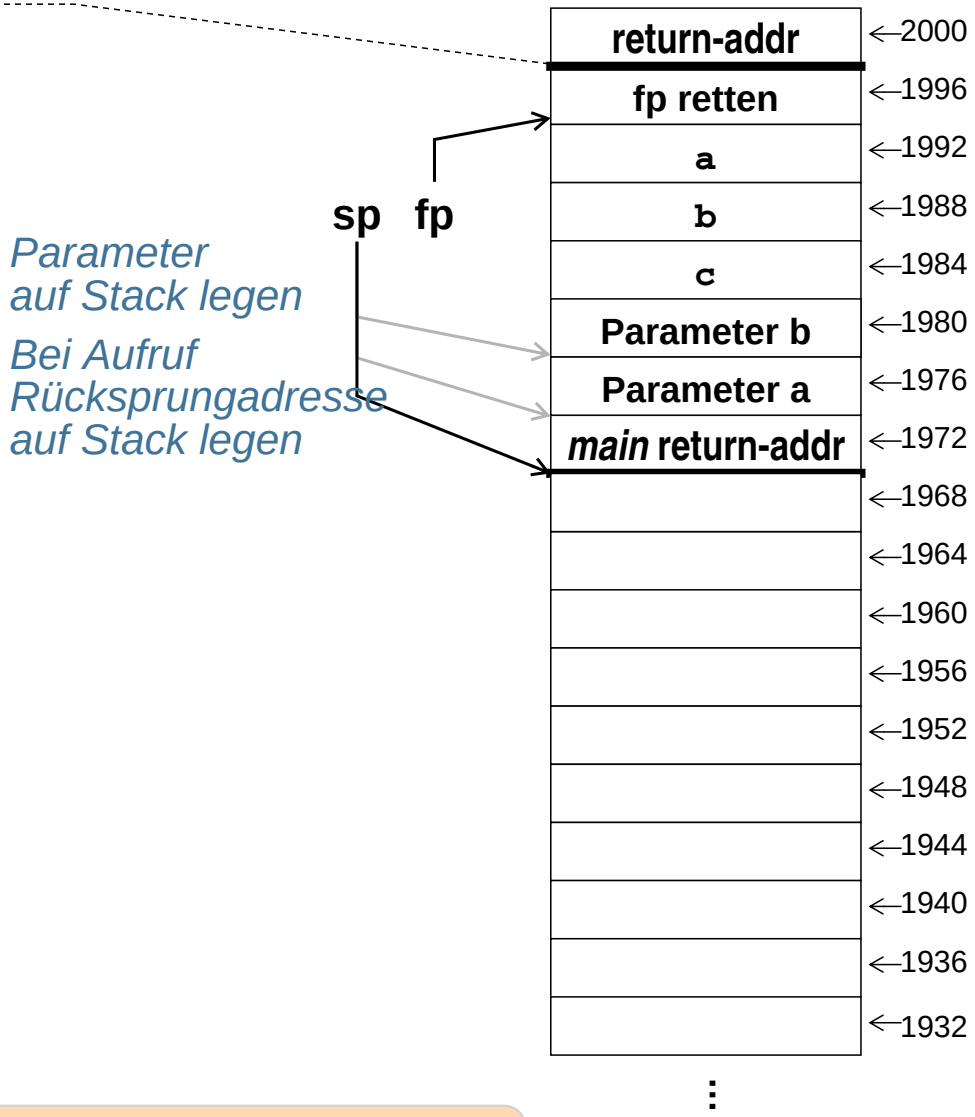
Stack-Frame für main erstellen
&a = fp-4
&b = fp-8
&c = fp-12



Beispiel hier für 32-Bit-Architektur (4-Byte `int`s), `main()` wurde soeben betreten

Stack-Aufbau bei Funktionsaufrufen

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

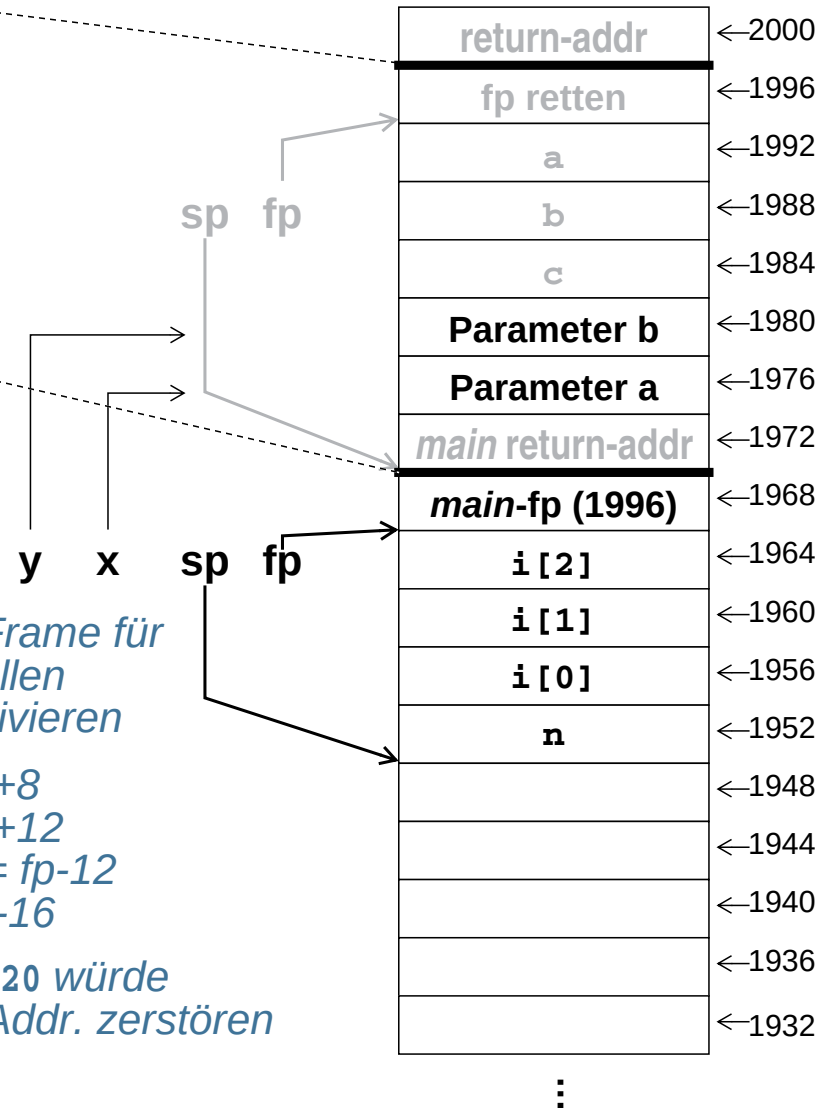


`main()` bereitet den Aufruf von `f1(int, int)` vor

Stack-Aufbau bei Funktionsaufrufen

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

```
int f1(int x, int y) {
        int i[3];
        int n;

        x++;

        n = f2(x);

        return(n);
}
```



Stack-Frame für f1 erstellen und aktivieren

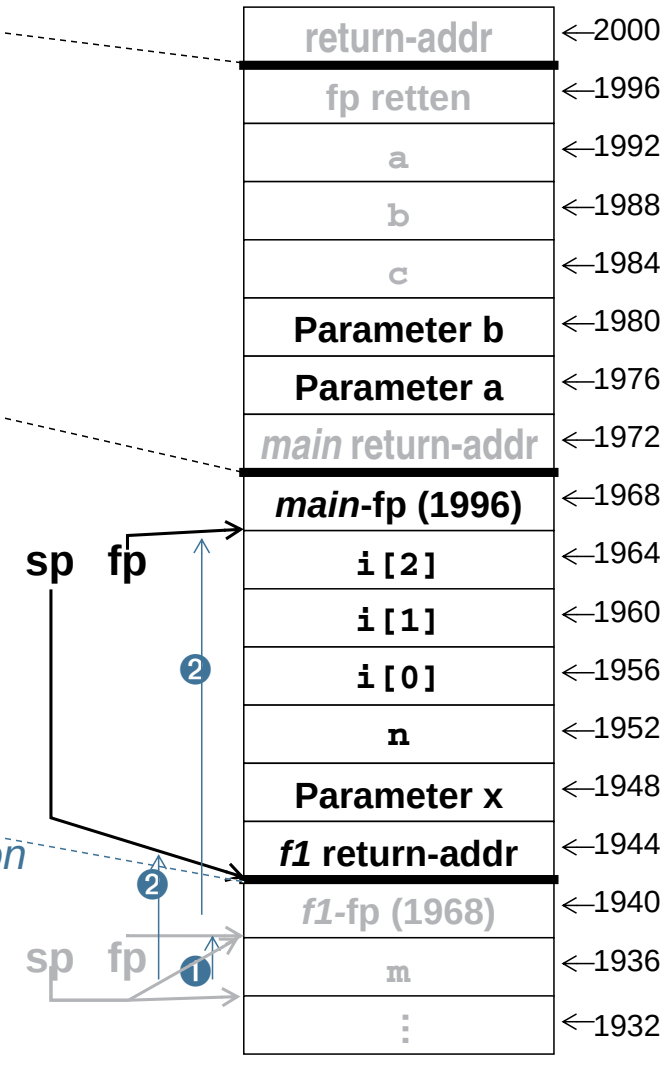
&x = fp+8
&y = fp+12
&(i[0]) = fp-12
&n = fp-16

`i[4] = 20` *würde return-Addr. zerstören*

f1() wurde soeben betreten

Stack-Aufbau bei Funktionsaufrufen

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}


int f1(int x, int y) {
        int i[3];
        int n;

        x++;

        n = f2(x);

        return(n);
}

int f2(int z) {
        int m;

        m = 100;

        return(z+1);
}
```



Stack-Frame von f2 abräumen

❶ *sp = fp*
❷ *fp = pop(sp)*

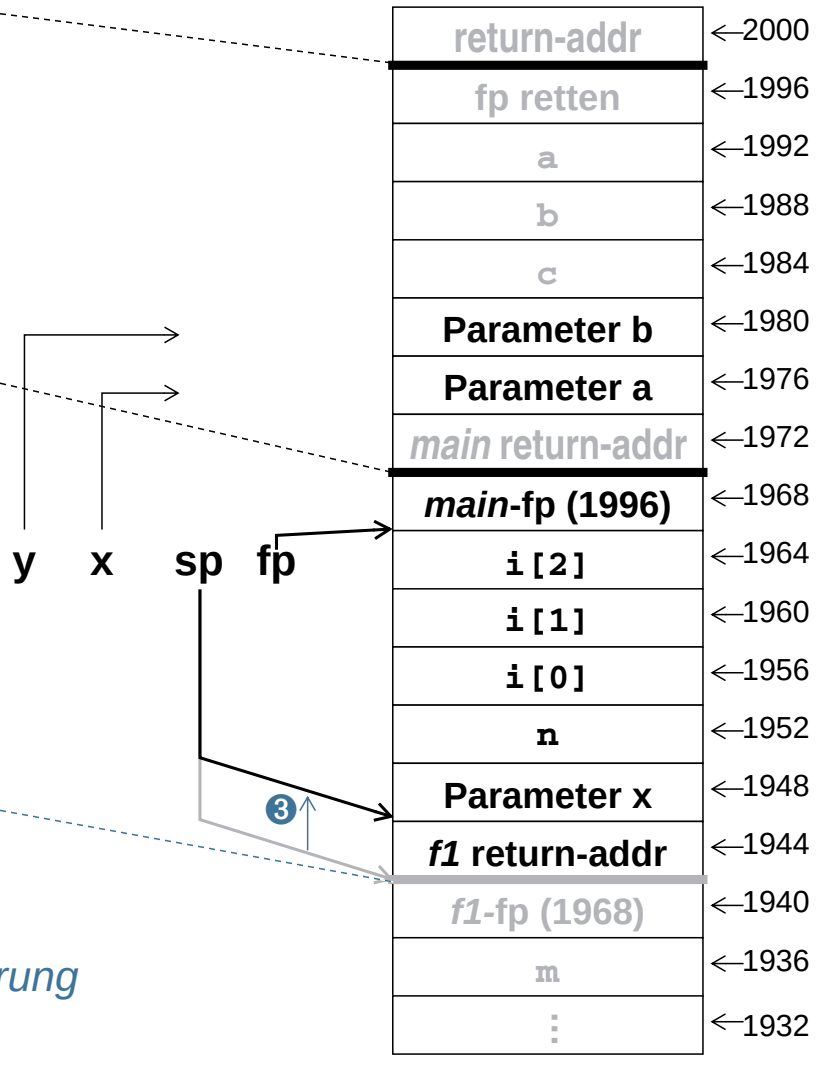
f2() bereitet die Terminierung vor (wurde von f1() aufgerufen und ausgeführt)

Stack-Aufbau bei Funktionsaufrufen

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}


int f1(int x, int y) {
        int i[3];
        int n;

        x++;

        n = f2(x);

        return(n);
}

int f2(int z) {
        int m;

        m = 100;

        return(z+1);
}
```

Rücksprung

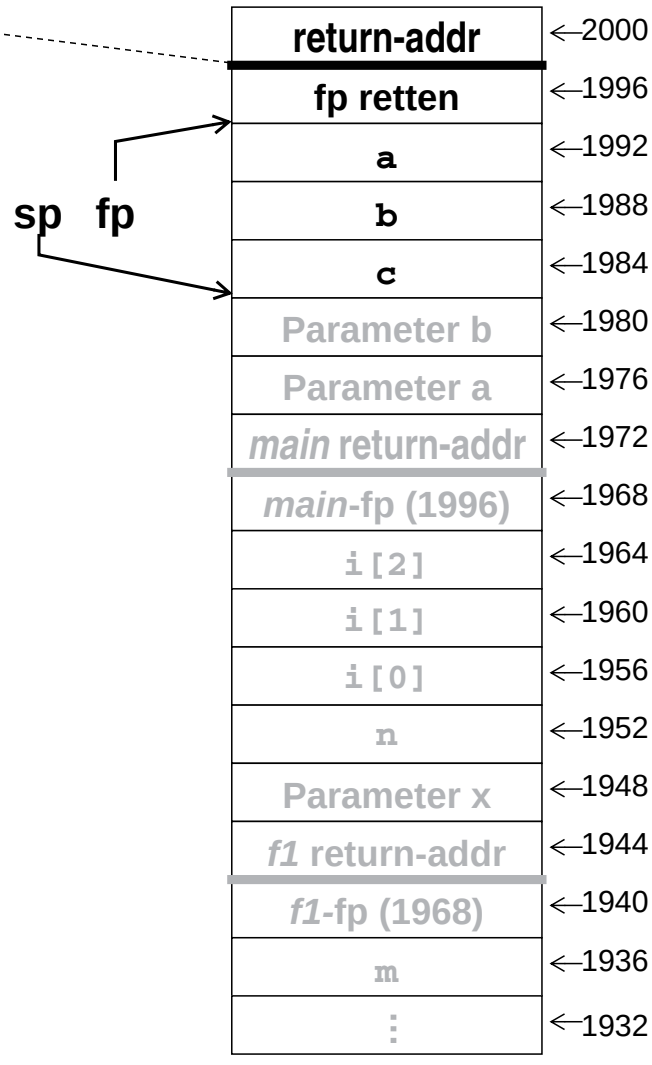
❸return



f2() wird verlassen

Stack-Aufbau bei Funktionsaufrufen

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

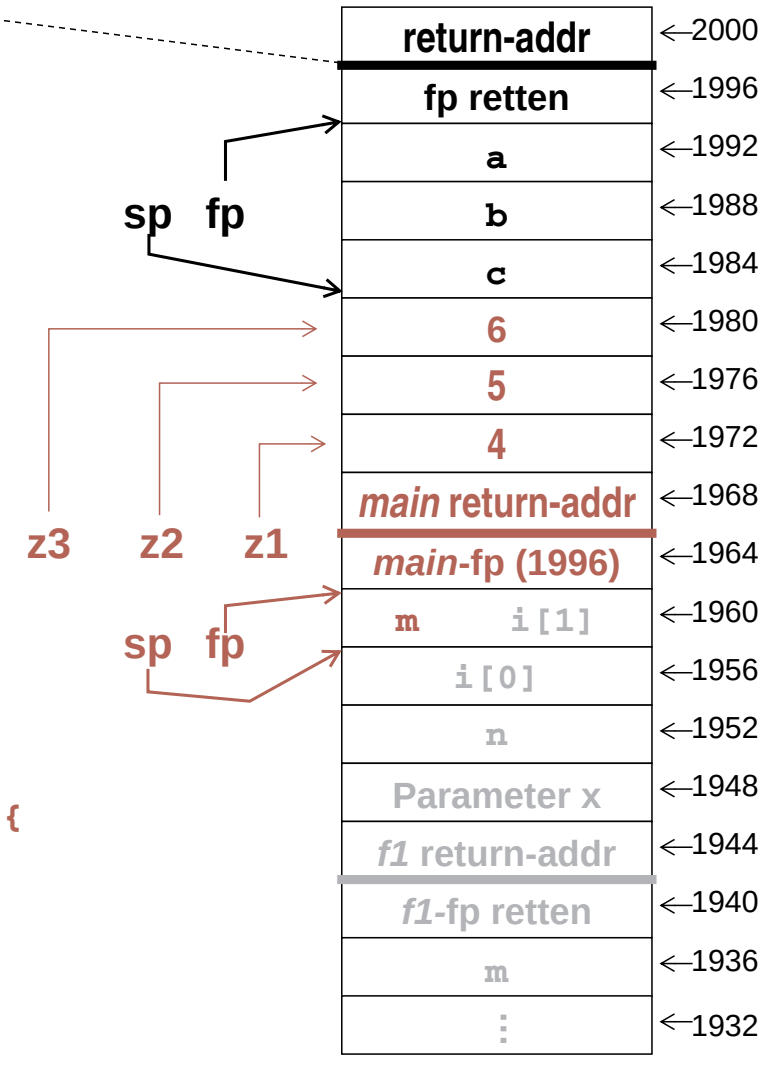


zurück in main()

Stack-Aufbau bei Funktionsaufrufen

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        f3(4,5,6);
}
```

was wäre, wenn man nach f1 jetzt eine Funktion f3 aufrufen würde?




```
int f3(int z1, int z2, int z3) {
        int m;

        return(m);
}
```

m wird nicht initialisiert ⇝„erbt“ alten Wert vom Stapel

Statische versus dynamische Allokation

- Bei der **μC-Entwicklung** wird statische Allokation bevorzugt
  - **Vorteil:** Speicherplatzbedarf ist bereits nach dem Übersetzen / Linken exakt bekannt (kann z. B. mit `size` ausgegeben werden)
  - Speicherprobleme frühzeitig erkennbar (Speicher ist knapp! ↪ 1–4)

```
lohmann@faui48a:$ size sections.avr
text     data     bss     dec     hex filename
682      10       6       698     2ba sections.avr
```

Sektionsgrößen des Programms von ↪ 20–1

⇝ Speicher möglichst durch `static`-Variablen anfordern

- Regel der geringstmöglichen Sichtbarkeit beachten ↪ 12–6
- Regel der geringstmöglichen Lebensdauer „sinnvoll“ anwenden

- Ein Heap ist verhältnismäßig teuer ⇝ wird möglichst vermieden
  - Zusätzliche Speicherkosten durch Verwaltungsstrukturen und Code
  - Speicherbedarf zur Laufzeit schlecht abschätzbar
  - Risiko von Programmierfehlern und Speicherlecks

Statische versus dynamische Allokation (Forts.)

- Bei der Entwicklung für eine **Betriebssystemplattform** ist dynamische Allokation hingegen sinnvoll
  - **Vorteil:** Dynamische Anpassung an die Größe der Eingabedaten (z. B. bei Strings)
  - Reduktion der Gefahr von *Buffer-Overflow*-Angriffen

$\rightsquigarrow$ Speicher für Eingabedaten möglichst auf dem Heap anfordern

- Das Risiko von Programmierfehlern und Speicherlecks bleibt!

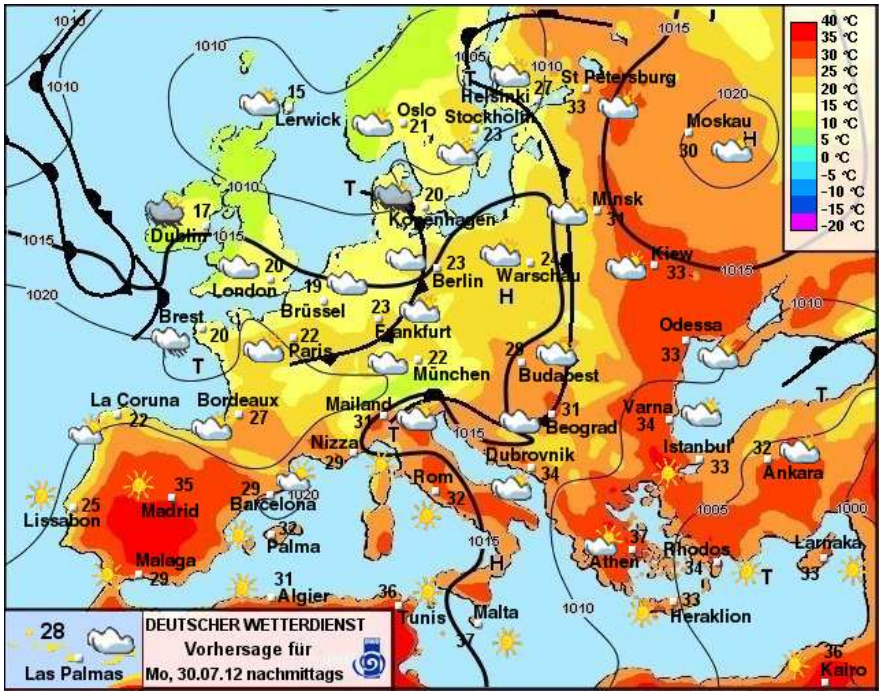
Überblick: Teil D Betriebssystemabstraktionen

Aktivitätsträger (Fäden / Threads)

- Mehrere Prozesse zur Strukturierung von Problemlösungen
  Aufgaben einer Anwendung leichter modellierbar, wenn sie in mehrere kooperierende Prozesse unterteilt wird
  - z.B. Anwendungen mit mehreren Fenstern (ein Prozess pro Fenster)
  - z.B. Anwendungen mit vielen gleichzeitigen Aufgaben (Web-Browser)
  - z.B. Client-Server-Anwendungen;
    pro Anfrage wird ein neuer Prozess gestartet (Web-Server)
- Multiprozessorsysteme werden erst mit mehreren parallel laufenden Prozessen ausgenutzt
  - früher nur bei Hochleistungsrechnern (Aerodynamik, Wettervorhersage)
  - durch Multicore-Systeme jetzt massive Verbreitung

Beispiel: Berechnung einer Wetterkarte

- Berechnung der Wetterkarte muss so schnell wie möglich erfolgen

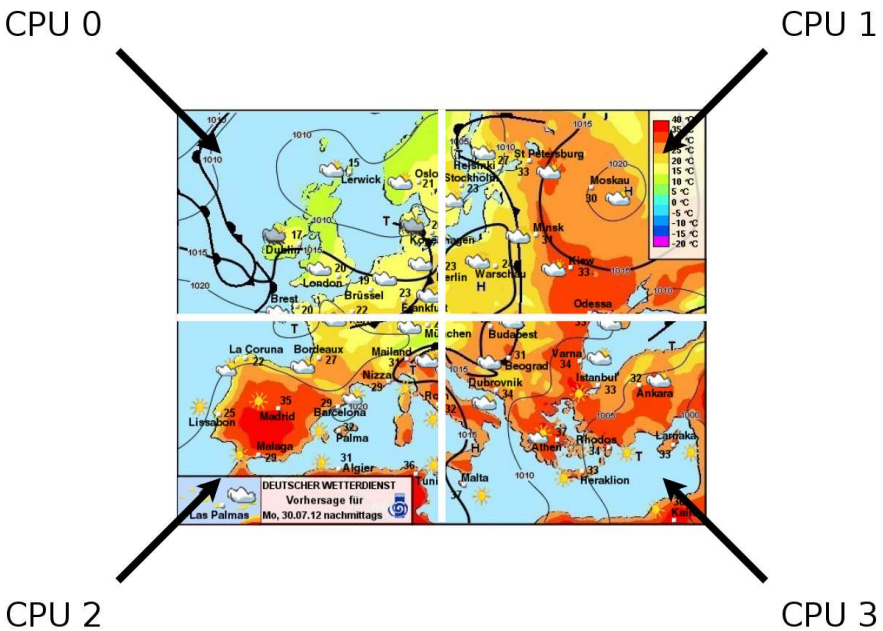


Quelle: www.wetterdienst.de

- Ansatz: Mehrere Prozessoren berechnen jeweils einen Teil der Karte

Beispiel: Berechnung einer Wetterkarte (2)

- Z.B. Berechnung der Wetterkarte aufgeteilt auf 4 Prozessoren:



- Alle Prozessoren greifen auf einen gemeinsamen Speicher zu, in dem das Ergebnis berechnet wird.

Prozesse mit gemeinsamem Speicher

- Nutzung von gemeinsamen Speicher durch mehrere Prozesse

```
char *ptr = mmap(NULL, NBYTES, PROT_READ | PROT_WRITE,
                 MAP_SHARED | MAP_ANONYMOUS, -1, 0);
if (ptr == MAP_FAILED) ... // Fehler

for (i = 0; i < NPROCESSES; i++) {
    pid[i] = fork();
    switch (pid[i]) {
    case -1: ...              // Fehler
    case 0:
        do_work(i, ptr);
        _exit(0);
    default:;
    }
}
for (i = 0; i < NPROCESSES; i++) {
    ret = waitpid(pid[i], NULL, 0);
    if (ret < 0) ...          // Fehler
}

ret = munmap(ptr, NBYTES);
if (ret < 0) ...              // Fehler
```

Beispiel: Vektorlänge

- Berechnung der Länge/Norm eines N-Elemente-Vektors mit einem Prozess:

```
#include <math.h>

double
veclen(double vec[])
{
    double sum = 0.0;

    for (int i = 0; i < N; i++) {
        sum += vec[i] * vec[i];
    }

    return sqrt(sum);
}
```


Beispiel: Vektorlänge (2)

- Berechnung der Länge eines N-Elemente-Vektors mit vier Prozessen:

```
double veclen(double vec[]) {
    pid_t pid[4];
    double *ptr = mmap(NULL, 4096, PROT_READ | PROT_WRITE,
                       MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    for (int p = 0; p < 4; p++) {
        if ((pid[p] = fork()) == 0) {
            double sum = 0.0;
            for (int i = p * N / 4; i < (p + 1) * N / 4; i++)
                sum += vec[i] * vec[i];
            ptr[p] = sum;
            _exit(0);
        }
    }
    for (int p = 0; p < 4; p++)
        waitpid(pid[p], NULL, 0);
    double sum = 0.0;
    for (int p = 0; p < 4; p++)
        sum += ptr[p];
    munmap(ptr, 4096);
    return sqrt(sum);
}
```


Beispiel: Vektorlänge (3)

- Hinweis: Beispiel unvollständig
  - `#includes` fehlen
  - Fehlerbehandlung fehlt
  - ...
- Trotzdem sieht man
  - Programmierung sehr viel aufwändiger
  - Programm sehr viel unübersichtlicher
  - eigentlicher Algorithmus kaum noch erkennbar
- Ergebnis ernüchternd
  - Aufwand lohnt sich bei aktuellen Rechnern erst ab etwa $N = 100000$

Prozesse mit gemeinsamem Speicher (2)

- Vorteil der obigen Lösung: in Multiprozessorsystemen sind echt parallele Abläufe möglich; aber
- jeder Prozess hat eigene Betriebsmittel
  - Speicherabbildung
  - Rechte
  - offene Dateien
  - Wurzel- und aktuelles Verzeichnis
  - ...
- => Prozess-Erzeugung, Prozess-Terminierung und Prozess-Umschaltungen sind teuer

Fäden in einem Prozess

- Lösung: mehrere Aktivitätsträger in einem Prozess
  - Fäden
  - Threads
  - Light-weight Processes (LWP)
  - ...
- jeder Faden repräsentiert einen eigenen aktiven Ablauf
  - eigener Programmzähler
  - eigener Registersatz
  - eigener Stack (für lokale Variablen)
- eine Gruppe von Fäden nutzt gemeinsam eine Menge von Betriebsmitteln (gemeinsame Ausführungsumgebung)
  - Speicherabbildung
  - Rechte
  - offene Dateien
  - Wurzel- und aktuelles Verzeichnis
  - ...

Fäden in einem Prozess (2)

- Das Konzept eines Prozesses wird aufgespalten in eine **Ausführungsumgebung** und ein oder mehrere **Aktivitätsträger**
- Ein klassischer UNIX-Prozess ist ein Aktivitätsträger in einer Ausführungsumgebung

Fäden in einem Prozess (3)

- Erzeugen/Terminieren eines Fadens in einem Prozess erheblich billiger als das Erzeugen/Terminieren eines Prozesses (keine eigenen Betriebsmittel)
- Umschalten zwischen Fäden innerhalb eines Prozesses erheblich billiger als das Umschalten zwischen Prozessen
  - es müssen nur die Register und der Programmzähler gewechselt werden (entspricht in etwa einem Funktionsaufruf)
  - Speicherabbildung muss nicht gewechselt werden (Cache-Inhalte bleiben gültig!)
- Implementierung von Fäden auf
  - Anwender-/Bibliotheks-Ebene (User-Level Threads)
  - Betriebssystem-Kern-Ebene (Kernel-Level Threads)

User-Level Threads

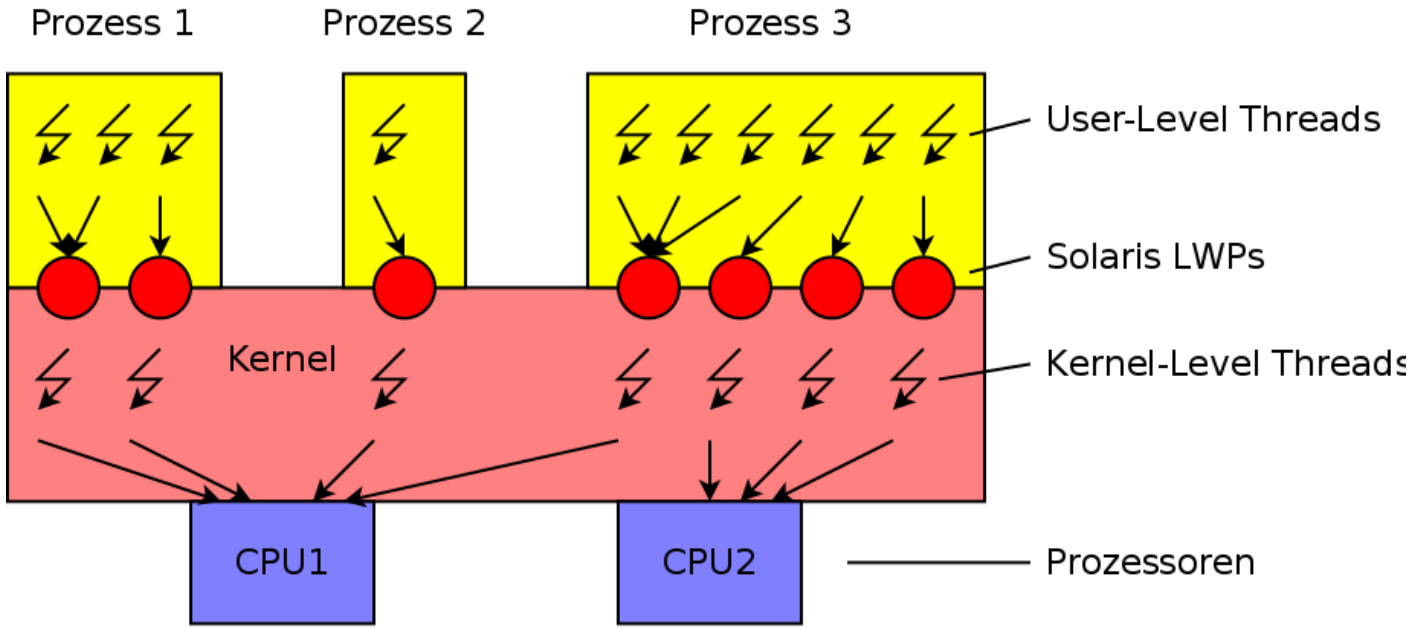
- Implementierung
  - Instruktionen im Anwendungsprogramm schalten zwischen den Fäden hin und her (ähnlich wie Scheduler im Betriebssystem)
  - Realisierung durch Bibliotheksfunktionen
  - Betriebssystem sieht nur einen Faden
- Vorteile
  - keine Systemaufrufe zum Umschalten erforderlich
  - effiziente Umschaltung (einige wenige Maschinenbefehle)
  - Scheduling-Strategie in der Hand des Anwendungsprogrammierers
- Nachteile
  - bei blockierenden Systemaufrufen bleibt die ganze Anwendung (und damit alle User-Level Threads) stehen
  - kein Ausnutzen eines Multiprozessors möglich

Kernel-Level Threads

- Implementierung
  - Betriebssystem kennt Kernel-Level Threads
  - Betriebssystem schaltet zwischen Threads um
- Vorteile
  - kein Blockieren unbeteiligter Fäden bei blockierenden Systemaufrufen
  - Betriebssystem kann mehrere Fäden einer Anwendung gleichzeitig auf verschiedenen Prozessoren laufen lassen
- Nachteile
  - weniger effizientes Umschalten zwischen Fäden (Systemaufruf notwendig)
  - Scheduling-Strategie meist durch Betriebssystem vorgegeben

Mischform: LWPs und Threads (Bsp. Solaris)

- Solaris kennt Kernel-, User-Level Threads und LWPs



Nach: Silberschatz, 1994

=> wenige Kernel-Level Threads um Parallelität zu erreichen, viele User-Level Threads, um die unabhängigen Abläufe in der Anwendung zu strukturieren

Koordinierung / Synchronisierung

- Fäden arbeiten nebenläufig/parallel und haben gemeinsamen Speicher

  => alle von Unterbrechungen und Signalen bekannten Probleme beim Zugriff auf gemeinsame Daten treten auch bei Fäden auf
- Unterschied zwischen Fäden und Interrupt-Service-Routinen bzw. Signal-Handler-Funktionen:
  - „Haupt-Faden" der Anwendung und eine ISR bzw. ein Signal-Handler sind nicht gleichberechtigt
    - ISR bzw. Signal-Handler unterbricht den Haupt-Faden aber ISR bzw. Signal-Handler werden nicht unterbrochen
  - zwei Fäden sind gleichberechtigt
    - ein Faden kann jederzeit zugunsten eines anderen unterbrochen werden (Scheduler) oder parallel zu einem anderen laufen (MPS)

=> Unterbrechungen sperren oder Signale blockieren hilft nicht!

Koordinierung / Synchronisierung (2)

- Grundlegende Probleme
  - gegenseitiger Ausschluss (**Koordinierung**)
    Beispiel:
    Ein Faden möchte einen Datensatz lesen und verhindern, dass ein anderer Faden ihn während dessen verändert.
  - gegenseitiges Warten (**Synchronisierung**)
    Beispiel:
    Ein Faden wartet auf andere Fäden, die jeweils Teilergebnisse berechnen sollen, die dann zusammengefasst werden.

Koordinierung / Synchronisierung (3)

- Komplexere Koordinierungs-/Synchronisierungsprobleme (Beispiele)
  - **Bounded Buffer**:
    Fäden schreiben Daten in Pufferspeicher, andere entnehmen Daten; kritische Situationen:
    - Zugriff auf den Puffer
    - Puffer leer
    - Puffer voll
  - **Philosophenproblem**:
    ein Faden reserviert sich zuerst den Zugriff auf Datenbereich 1, dann auf Datenbereich 2; ein anderer Faden umgekehrt;
    Problem:
    - kann zu Verklemmungen führen

Gegenseitiger Ausschluss (Mutual Exclusion)

- Einfache Implementierung durch **mutex**-Variablen

```
volatile int m = 0; /* 0: free; 1: locked */
volatile int counter = 0;
```

```
...         /* Thread 1 */
lock(&m);
counter++;
unlock(&m);
...
```

```
...         /* Thread 2 */
lock(&m);
printf("%d\n", counter);
counter = 0;
unlock(&m);
...
```

Nur der Faden, der `lock` aufgerufen hat, darf `unlock` aufrufen!

- Realisierung (nur konzeptionell!)

```
void lock(volatile int *m) {
    while (*m == 1) {
        /* Wait... */
    }
    *m = 1;
}
```

```
void unlock(volatile int *m) {
    *m = 0;
}
```

`lock` (und ggf. `unlock`) müssen **atomar** ausgeführt werden!

Zählende Semaphore

- Ein `Semaphor` (griech. Zeichenträger) ist eine Datenstruktur mit zwei Operationen (nach *Dijkstra*):
  - `P-Operation` (*proberen; passeren; wait; down*)

```
void P(volatile int *s)
{
    while (*s <= 0) {
        /* Wait/sleep... */
    }
    *s -= 1;
}
```

  - `V-Operation` (*verhogen; vrijgeven; signal; up*)

```
void V(volatile int *s)
{
    *s += 1;
    /* Wakeup... */
}
```

P und V müssen **atomar** ausgeführt werden!

P und V müssen nicht vom selben Faden aufgerufen werden.

Zählende Semaphore (2)

- Semaphor-Beispiel:

```
volatile int barrier = 0;
int result;
```

```
...         /* Thread 1 */
result = f1();
V(&barrier);
...
```

```
...         /* Thread 2 */
P(&barrier);
f2(result);
...
```

- Faden 1 läuft immer ungehindert durch.
- Faden 2 blockiert an `P`, falls Faden 1 die V-Operation noch nicht ausgeführt hat (und wartet auf die V-Operation) – andernfalls läuft Faden 2 auch durch.

Spin Lock vs. Sleeping Lock

- **Spin Lock**
  - aktives Warten, bis Mutex-Variable frei ($= 0$) wird
  - entspricht konzeptionell einem Pollen
  - Faden bleibt im Zustand „laufend”

  Problem: wenn nur ein Prozessor verfügbar ist, wird Rechenzeit vergeudet, bis durch den Scheduler eine Umschaltung erfolgt
  - nur ein anderer, laufender Faden kann den Mutex freigeben
- **Sleeping Lock**
  - passives Warten
  - Faden geht in den Zustand „blockiert”
  - im Rahmen von `unlock` wird der blockierte Faden in den Zustand „bereit” zurückgeführt

  Problem: bei sehr kurzen kritischen Abschnitten ist der Aufwand für das Blockieren/Aufwecken und die Umschaltung unverhältnismäßig teuer

Implementierung Spin Lock

- zentrales Problem: Atomarität von mutex-Abfrage und -Setzen

```
void lock(volatile int *m) {
    while (*m == 1) {
        /* Wait... */
    }
    *m = 1;
}
```

kritischer Abschnitt

- Lösung: spezielle Maschinenbefehle, die atomar eine Abfrage und eine Modifikation einer Hauptspeicherzelle ermöglichen
  - *Test-and-Set*, *Compare-and-Swap*, *Load-Link/Store-Conditional*, ...

Implementierung Spin Lock (2)

- Beispiel: Implementierung mit *Compare-and-Swap*-Befehl

```
void lock(volatile int *m)
{
    while (compare_and_swap(m, 0, 1) != 0) {
        /* Wait... */
    }
}
```

mit

```
int compare_and_swap(volatile int *ptr, int oldval, int newval)
{
    int res = *ptr;
    if (*ptr == oldval) {
        *ptr = newval;
    }
    return res;
}
```

Diese Funktion existiert z.B. bei x86-Prozessoren als ein atomar ablaufender Maschinenbefehl (`cmpxchg`).

Implementierung Sleeping Lock

- zwei Probleme:

1. Konflikt mit einer zweiten `lock`-Operation:
   Atomarität von mutex-Abfrage und -Setzen

```
void lock(volatile int *m) {
    while (*m == 1) {
        sleep();
    }
    *m = 1;
}
```

kritischer Abschnitt 1

2. Konflikt mit einem `unlock`: *lost-wakeup*-Problem

```
void lock(volatile int *m) {
    while (*m == 1) {
        sleep();
    }
    *m = 1;
}
```

kritischer Abschnitt 2

- Ursachen:

1. Prozessumschaltung während der `lock`-Operation
2. Echt-parallel laufende `lock`- und/oder `unlock`-Operationen

Implementierung Sleeping Lock (2)

- Behebung von Ursache (1):
  Prozessumschaltungen verhindern
  - Prozessumschaltung ist eine Funktion des BS-Kerns
    - erfolgt im Rahmen eines BS-Aufrufs (z.B. `exit`)
    - oder im Rahmen einer Unterbrechungs-Behandlung (z.B. Zeitscheiben-Unterbrechung)

  => `lock/unlock` werden ebenfalls im BS-Kern implementiert; BS-Kern mit Unterbrechungs-Sperre

```
void lock(volatile int *m)
{
    enter_OS();
    cli();
    while (*m == 1) {
        block_thread_and_schedule();
    }
    *m = 1;
    sei();
    leave_OS();
}
```

```
void unlock(volatile int *m)
{
    enter_OS();
    cli();
    *m = 0;
    wakeup_waiting_threads();
    sei();
    leave_OS();
}
```

Implementierung Sleeping Lock (3)

- Behebung von Ursache (2):
  Parallele Ausführung auf anderem Prozessor verhindern

```
void lock(volatile int *m)
{
    enter_OS();
    cli();
    spin_lock();
    while (*m == 1) {
        block_thread_and_schedule();
    }
    *m = 1;
    spin_unlock();
    sei();
    leave_OS();
}
```

```
void unlock(volatile int *m)
{
    enter_OS();
    cli();
    spin_lock();
    *m = 0;
    wakeup_waiting_threads();
    spin_unlock();
    sei();
    leave_OS();
}
```


Beispiel: POSIX Threads (pthread)

- Programmierschnittstelle standardisiert: **pthread-Bibliothek** (IEEE-POSIX-Standard P1003.4a)
- pthread-Schnittstelle (Basisfunktionen):

  `pthread_create:` Faden erzeugen
  `pthread_exit:` Faden beendet sich selbst
  `pthread_join:` auf Ende eines Fadens warten
  `pthread_self:` eigene Faden-ID abfragen
  `pthread_yield:` Faden gibt CPU freiwillig auf

- Funktionen in pthread-Bibliothek zusammengefasst

```
gcc ... -pthread ...
```


pthread-Schnittstelle

- Faden-Erzeugung

```
#include <pthread.h>

int pthread_create(pthread_t *tid, const pthread_attr_t *attr,
                   void *(*func)(void *), void *param);
```

- Parameter

  `tid`: Zeiger auf Variable, in der die Faden-ID abgelegt werden soll.

  `attr`: Zeiger auf Attribute (z.B. Stack-Größe) des Fadens. `NULL` für Standard-Attribute.

  `func`, `param`: Der neu erzeugte Faden führt die Funktion `func` mit dem Parameter `param` aus.

- Als Rückgabewert wird 0 geliefert. Im Fehlerfall wird ein Fehlercode (ähnlich `errno`) zurückgeliefert.

pthread-Schnittstelle

- Faden beenden (bei `return` aus `func` oder):

```
#include <pthread.h>

void pthread_exit(void *retval);
```

Der Faden wird beendet und `retval` wird als Rückgabewert zurückgeliefert (siehe `pthread_join`.

- Auf Faden warten und `pthread_exit`-Status abfragen:

```
#include <pthread.h>

int pthread_join(pthread_t tid, void **retvalp);
```

Wartet auf den Faden mit der Faden-ID `tid` und liefert dessen Rückgabewert über `retvalp` zurück.
Als Rückgabewert wird 0 geliefert. Im Fehlerfall wird ein Fehlercode (ähnlich `errno`) zurückgeliefert.

pthread-Beispiel

- Beispiel (Multiplikation Matrix mit Vektor; $\vec{c} = A\vec{b}$):

```
double a[100][100], b[100], c[100];

static void *mult(void *ci) {
    int i = (double *) ci - c;

    double sum = 0.0;
    for (int j = 0; j < 100; j++) {
        sum += a[i][j] * b[j];
    }
    c[i] = sum;
    return NULL;
}

int main(void) {
    pthread_t tid[100];

    for (int i = 0; i < 100; i++) {
        pthread_create(&tid[i], NULL, mult, &c[i]);
    }
    for (int i = 0; i < 100; i++) {
        pthread_join(tid[i], NULL);
    }
}
```

pthread-Koordinierung und -Synchronisierung

- Koordinierung durch Mutex-Variablen
  - Erzeugung von Mutex-Variablen

```
pthread_mutex_t m;
pthread_mutex_init(&m, NULL);
```

  - `lock`-Operation

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *m);
```

  - `unlock`-Operation

```
#include <pthread.h>

int pthread_mutex_unlock(pthread_mutex_t *m);
```


pthread-Beispiel

- Mutex-Beispiel:

```
volatile int counter = 0;
pthread_mutex_t m;
pthread_mutex_init(&m, NULL);
```

```
...      /* Thread 1 */
pthread_mutex_lock(&m);
counter++;
pthread_mutex_unlock(&m);
...
```

```
...      /* Thread 2 */
pthread_mutex_lock(&m);
printf("counter = %d\n", counter);
counter = 0;
pthread_mutex_unlock(&m);
...
```

pthread-Koordinierung und -Synchronisierung (2)

- Synchronisierung durch Bedingungs-Variablen (Condition Variable)
  - auf eine Bedingung kann gewartet werden (sleep)
  - eine Bedingung kann signalisiert werden (wakeup)
  - Erzeugung einer Condition-Variablen

```
pthread_cond_t c;
pthread_cond_init(&c, NULL);
```

  - auf eine Bedingung warten

```
#include <pthread.h>

int pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
```

  - eine Bedingung signalisieren

```
#include <pthread.h>

int pthread_cond_signal(pthread_cond_t *c);
int pthread_cond_broadcast(pthread_cond_t *c);
```

`pthread_cond_signal` weckt *einen* Faden, `pthread_cond_broadcast` weckt *alle* auf die Bedingung wartenden Fäden auf

pthread-Beispiel (2)

- Beispiel: zählende Semaphore

```
pthread_mutex_t m;
pthread_cond_t c;

pthread_mutex_init(&m, NULL);
pthread_cond_init(&c, NULL);
```

```
void P(volatile int *s) {
    pthread_mutex_lock(&m);
    while (*s == 0) {
        pthread_cond_wait(&c, &m);
    }
    *s -= 1;
    pthread_mutex_unlock(&m);
}
```

```
void V(volatile int *s) {
    pthread_mutex_lock(&m);
    *s += 1;
    pthread_cond_broadcast(&c);
    pthread_mutex_unlock(&m);
}
```

Fäden, Koordinierung, Synchronisierung in Java

- Faden-Konzept, Koordinierung und Synchronisierung in Java integriert
- Erzeugung von Fäden über die Thread-Klasse; Beispiel:

```
class MyClass implements Runnable {
    public void run() {
        System.out.println("Hello!");
    }
}
...
MyClass o = new MyClass();    // create object
Thread t1 = new Thread(o);    // create thread to run in o
t1.start();                   // start thread
Thread t2 = new Thread(o);    // create second thread
t2.start();                   // start second thread
```


Fäden, Koordinierung, Synchronisierung in Java (2)

- Koordinierung und Synchronisierung über jedes beliebige Objekt
  - Koordinierung über `synchronized`-Blöcke

```
synchronized(obj) {
    ...
}
```

    Ein solcher Block ruft zu Block-Beginn ein `lock` auf das Objekt `obj` auf, führt die angegebenen Anweisungen aus, und ruft vor dem Verlassen des Blockes das entsprechende `unlock` auf.
  - Synchronisierung über `wait`, `notify` und `notifyAll`

    `obj.wait()`: wartet auf die Signalisierung einer Bedingung auf dem angegebenen Objekt `obj`.

    `obj.notify()`: signalisiert eine Bedingung auf dem angegebenen Objekt `obj` an *einen* wartenden Faden.

    `obj.notify()`: signalisiert eine Bedingung auf dem angegebenen Objekt `obj` *allen* wartenden Fäden.

Fäden, Koordinierung, Synchronisierung in Java (3)

- Beispiel Koordinierung und Synchronisierung:

```
public class Semaphore {
    private int s;

    public Semaphore(int s0) {
        s = s0;
    }
    public void P() {
        synchronized(this) {
            while (s == 0)
                this.wait();
            s--;
        }
    }
    public void V() {
        synchronized(this) {
            s++;
            this.notifyAll();
        }
    }
}
```

Entspricht dem pthread-Beispiel...

Fäden, Koordinierung, Synchronisierung in Java (4)

- Vereinfachte Schreibweise (entspricht „Monitor"-Konzept):

```
public class Semaphore {
    private int s;

    public Semaphore(int s0) {
        s = s0;
    }
    public synchronized void P() {
        while (s == 0) {
            wait();
        }
        s--;
    }
    public synchronized void V() {
        s++;
        notifyAll();
    }
}
```