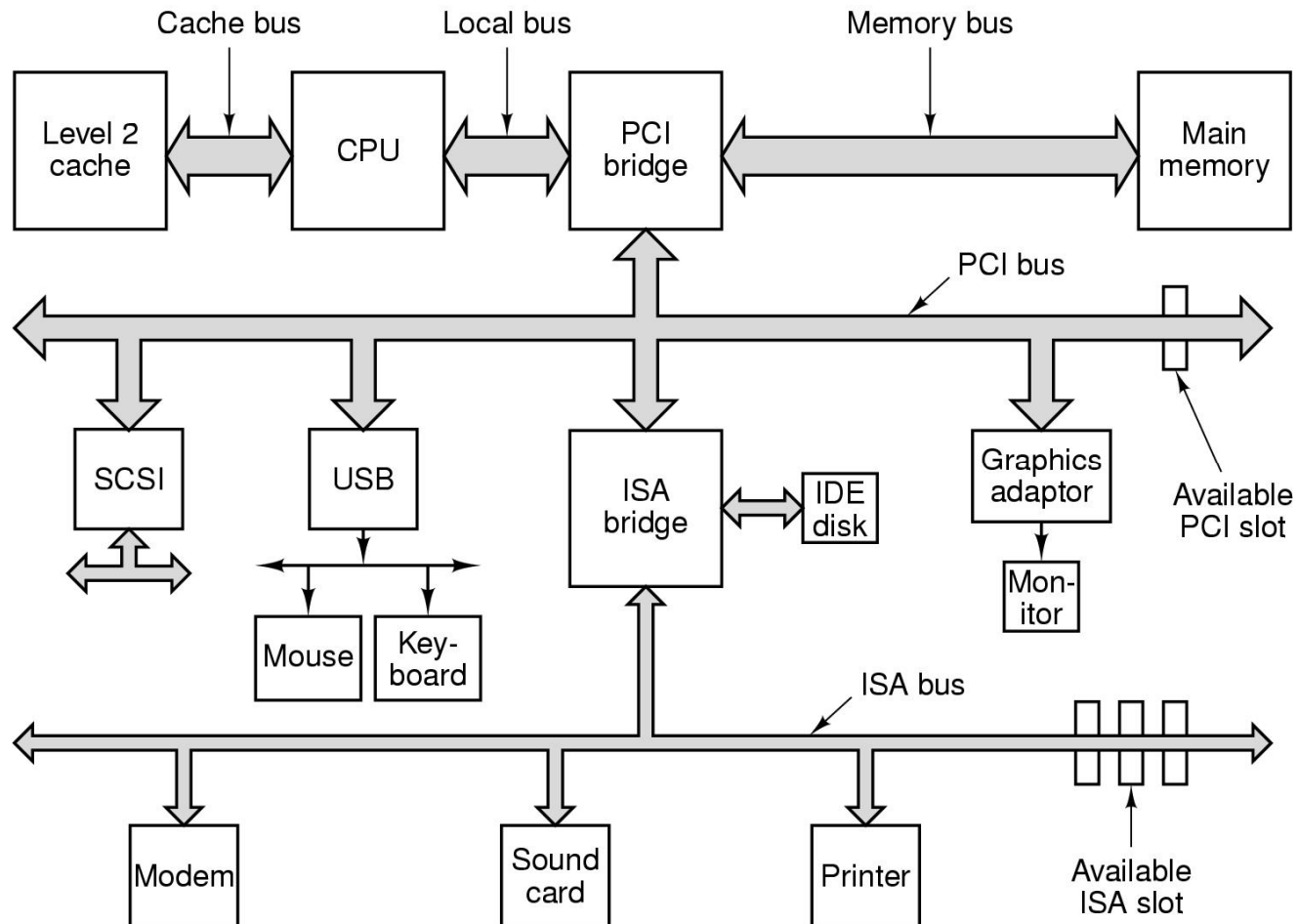


Betriebssysteme

Hardware & Systemprogrammierung

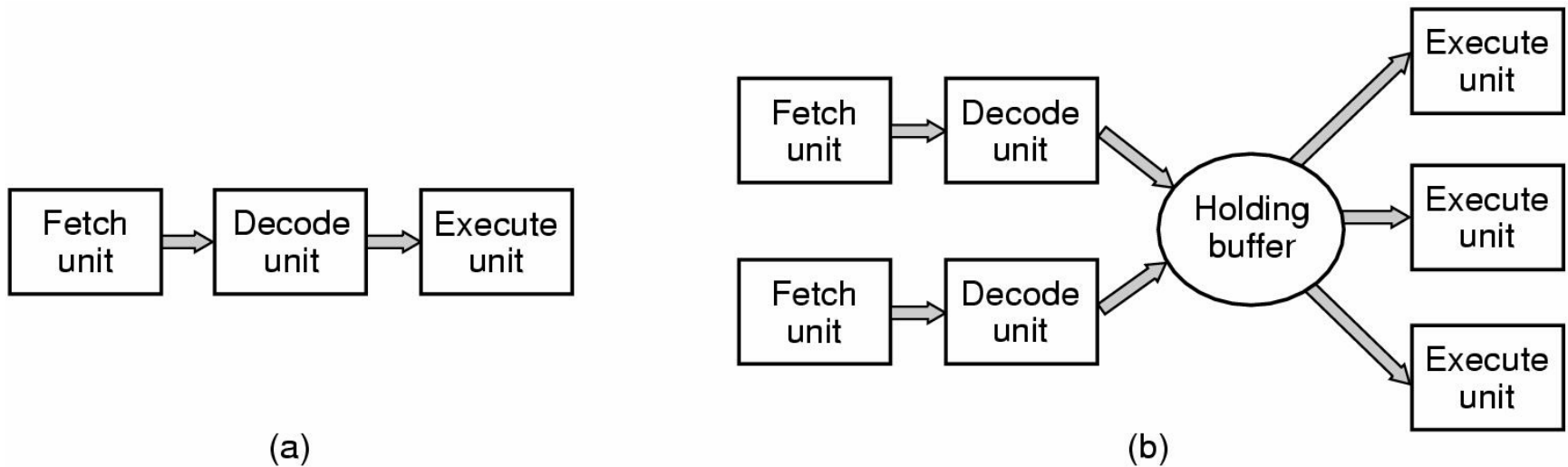
Prof. Dr.-Ing. Torben Weis
Universität Duisburg-Essen

Struktur eines PCs (älterere Pentium)



Computer Hardware / CPU

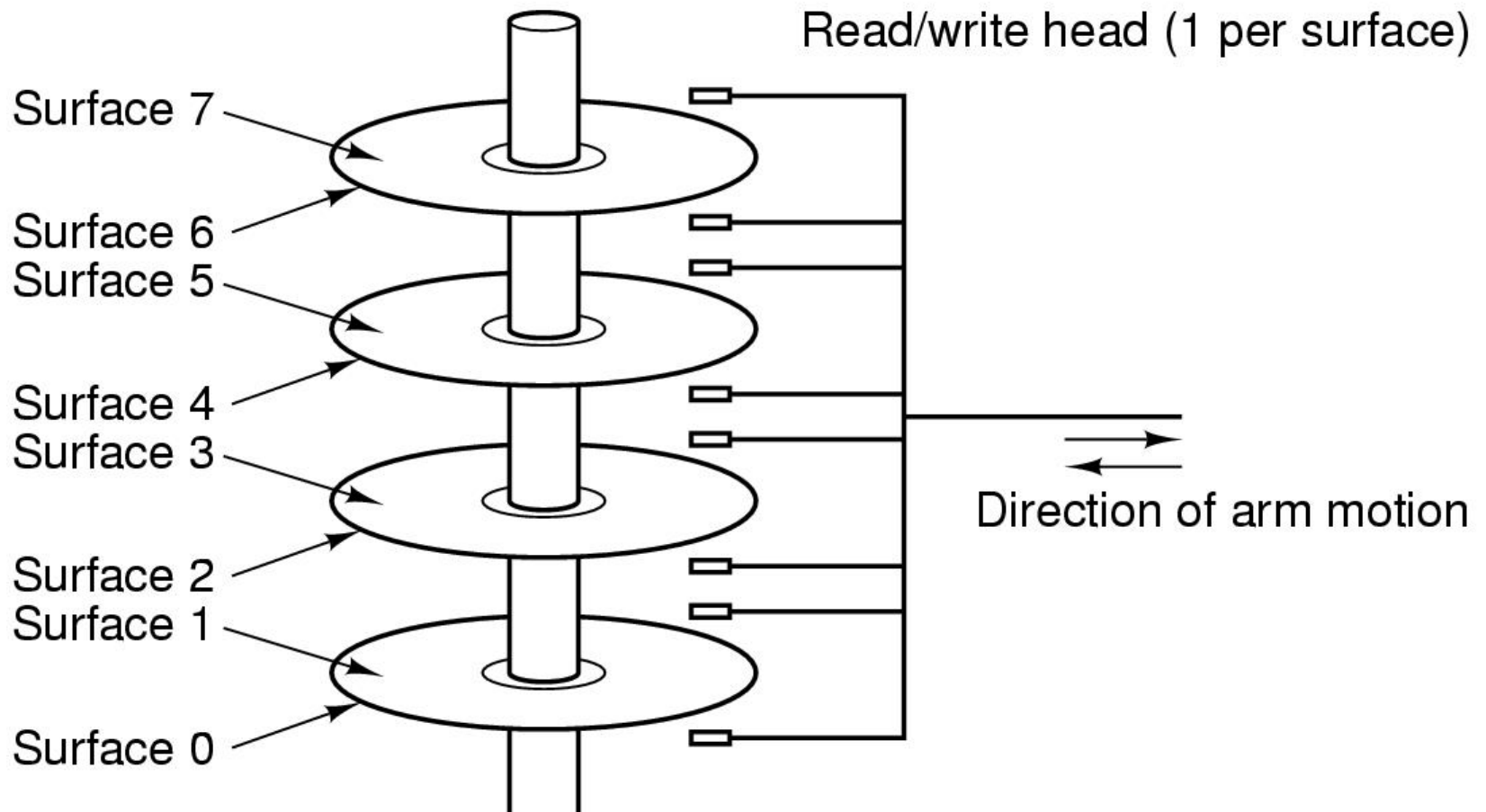
- Architektur der CPU kann entscheidend sein für das Scheduling



(a) Einfache CPU mit 3-stufiger Pipeline

(b) Super skalare CPU

Computer Hardware / Festplatte



Angenommene Realisierung von persistentem Speicher

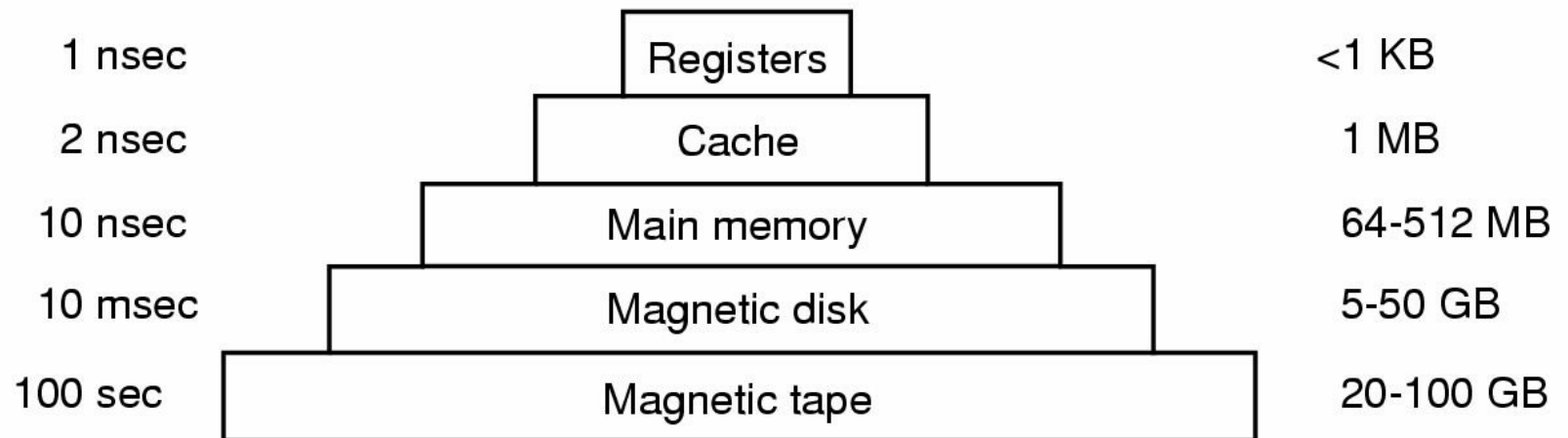
Computer Hardware / Speicher

- RAM
 - Random Access Memory
 - Nicht-persistent (Systemspeicher)
 - Persistent (USB Memory Stick)
- ROM
 - Read-only Memory
- EEPROM
 - Electrically Erasable ROM
- CMOS
 - Flüchtiger Speicher
 - Sehr geringer Stromverbrauch
 - Nützlich für Systemuhren und Hardware Config

Computer Hardware / Zugriffszeiten

Typical access time

Typical capacity



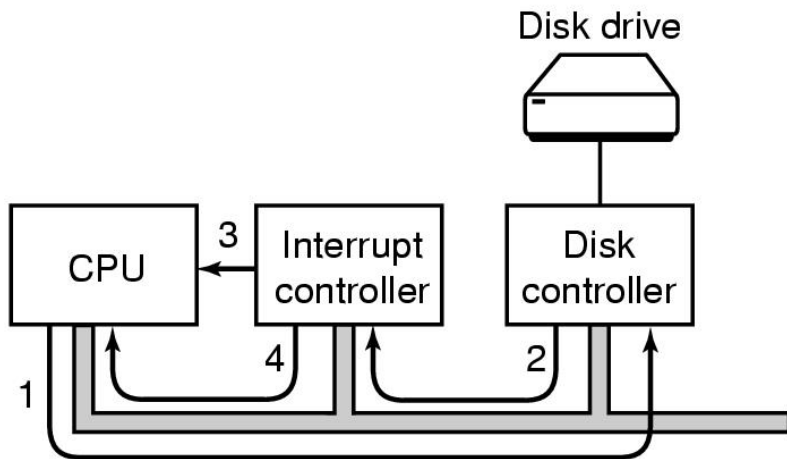
Computer Hardware / Treiber

- Einbindung in den Kernel
 - Dynamisch oder statisch
- 3 Methoden
 - 1) Compilieren des Kerns mit dem Treiber
 - Zu umständlich für PCs
 - Beispiel: Alte UNIX Kernels
 - 2) Laden des Treibers gemäß einer Config Datei
 - Beim Systemstart
 - Beispiel: MS-DOS, Windows 3.x, Windows 95
 - 3) Dynamisches Laden des Treibers
 - Notwendig für Bus mit Hot-Plug Geräten (USB)
 - Beispiel: Alle modernen PC Betriebssysteme

Computer Hardware / Controller

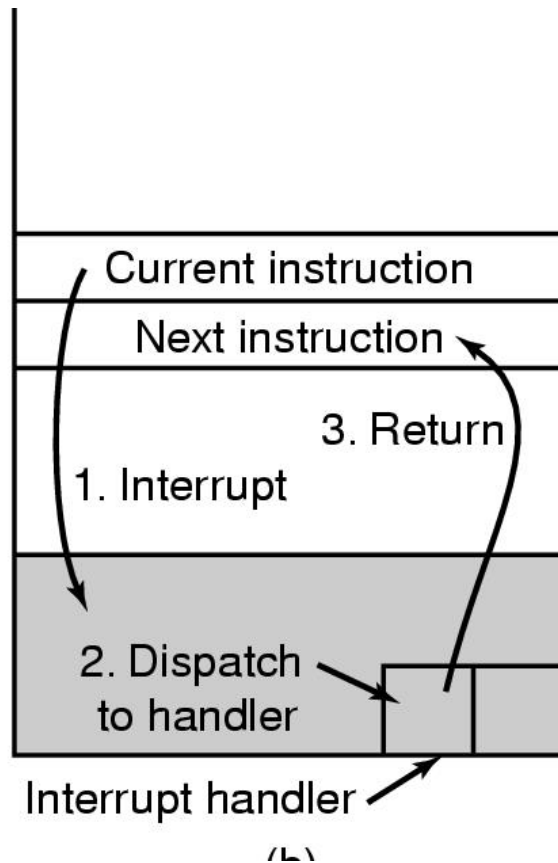
- Controller verbinden PC und Geräte
 - Ansteuerung durch den Treiber
- Jeder Controller hat Register
 - Ein-/Ausgabe Register
- Ansprechen über Speicher Adressen
 - Register werden in Speicher eingeblendet
 - Zugriffsschutz über MMU geregelt
- Ansprechen über IO Instruktionen
 - Benötigt spezielle CPU Kommandos
 - IO Instruktionen nur im Kernel Mode erlaubt

Computer Hardware / Interrupt



- 1) Treiber beauftragt den Disk Controller
- 2) Disk Controller ist fertig und möchte einen Interrupt auslösen
- 3) Über ein PIN wird der CPU ein Interrupt signalisiert
- 4) Index des unterbrechenden Gerätes wird auf den Bus gelegt

Computer Hardware / Interrupt



1) Interrupt Handler

- Sichern von Registern und Programmzähler
- Sprung in den Kernel

2) Auswählen des Handlers

- Anhand des Geräte Index

3) Rücksprung

- Wiederherstellen von Registern und Programmzähler

Interrupt versus Trap

- Gleich
 - Beide ermöglichen Wechsel in den Kernel Mode
 - (Manchmal) beides vom User Mode aus möglich
 - Beides nützlich für Systemaufrufe
- Unterschiedlich
 - Interrupts werden von Controllern ausgelöst
 - Interrupts werden über Hardware (Interrupt Leitung) von außerhalb der CPU signalisiert
 - Traps werden von Software ausgelöst
 - Behandlung eines Traps erfolgt innerhalb der CPU

Computer Hardware / BIOS

- BIOS = Basic Input Output System
- Zuständig für die Zuteilung von Interrupts
 - Zuteilung bei Systemstart
 - 1. Schritt: Legacy Systeme
 - Interrupt der Jumper eingestellt
 - 2. Schritt: Plug and Play Geräte
 - Zuweisung dynamisch
- BIOS startet das Betriebssystem
 - 1. Schritt: „Boot device“ finden
 - 2. Schritt: Ersten Sektor laden und ausführen
 - 3. Schritt: Erster Sektor enthält den „Loader“

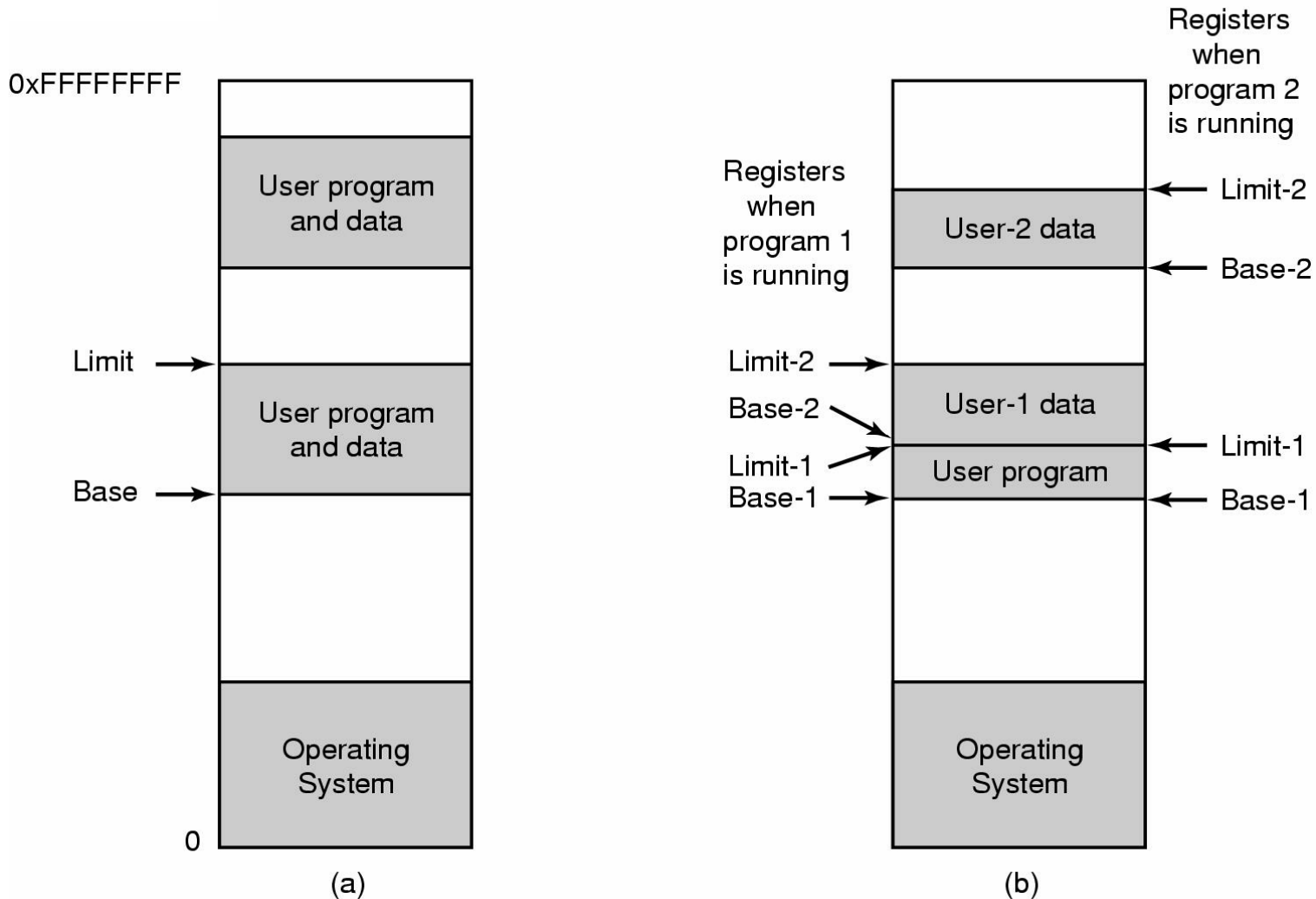
Computer Hardware / DMA

- DMA = Direct Memory Access
 - Geräte können direkt mit dem Speicher arbeiten
 - Entlastung der CPU
- Funktionsweise
 - DMA regelt Datenfluss zwischen Speicher und Controllern
 - CPU teilt dem DMA Speicheradresse, Richtung (Lesen/Schreiben), Länge und Gerät mit
 - Interrupt wenn Aufgabe beendet wurde
- Beispiel
 - Soundkarte

Computer Hardware / MMU

- MMU = Memory Management Unit
- MMU hat 3 wichtige Aufgaben
 - Schutz des Kernel Speichers vor Anwendungen
 - Schutz der Anwendungen untereinander
 - Virtueller Speicher
- Laden eines Programms
 - a) Reloziieren aller absoluten Adressen
 - Sehr rechenaufwändig
 - b) Virtueller Speicher mit Basis und Limit Registern
 - Benötigt spezielle Hardware, d.h. eine MMU

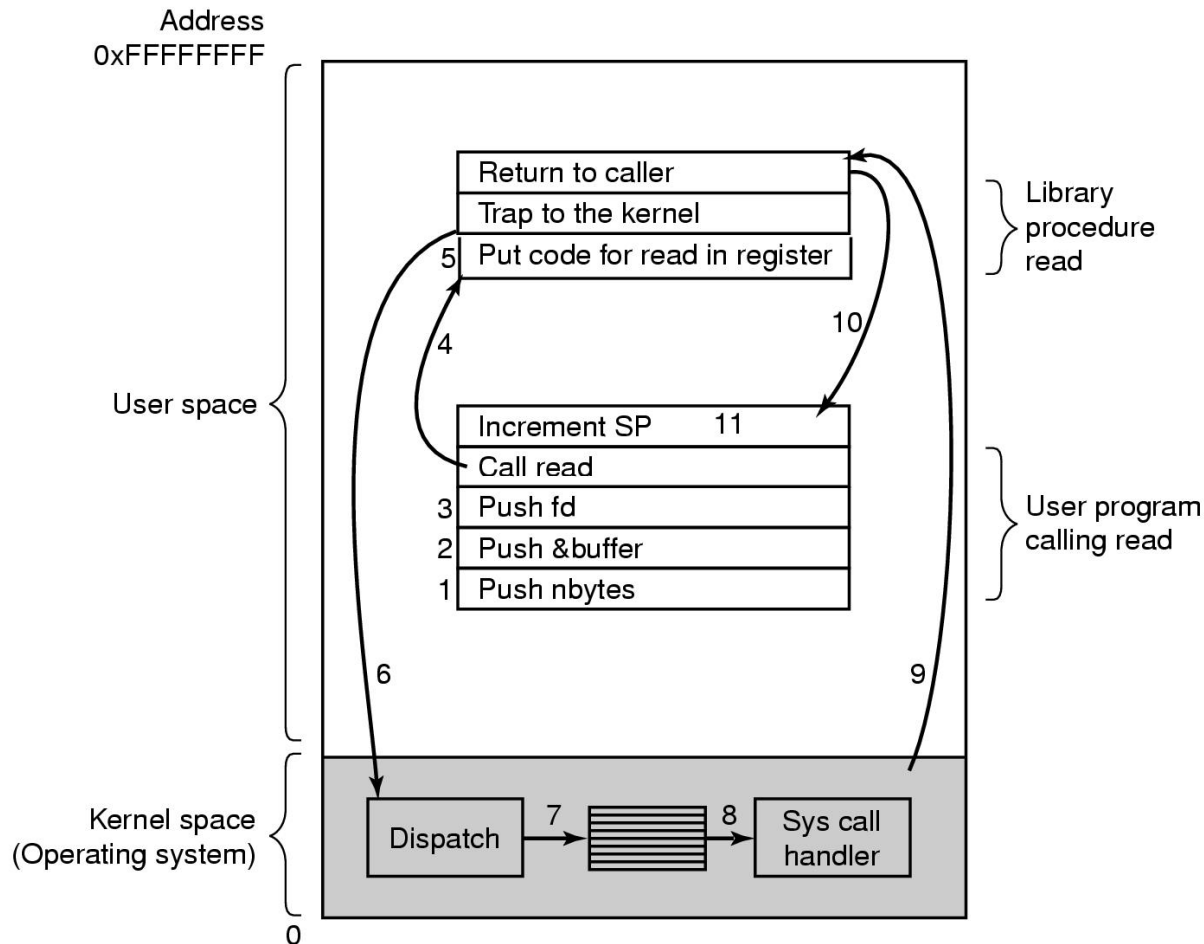
Computer Hardware / MMU



Computer Hardware & Prozesswechsel

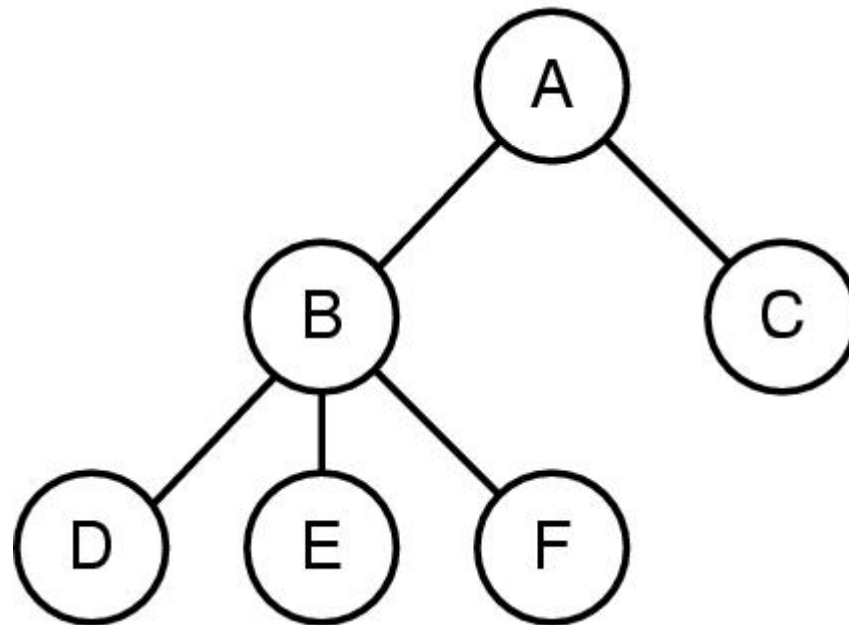
- Register müssen gesichert werden
 - CPU Register
 - MMU Register
 - Je mehr Register desto schlimmer
- Cache muss neu gefüllt werden
 - Beim Wechsel ist der Cache voll ...
 - ... aber mit den falschen Daten
 - Viele Cache-Misses
- Prozesswechsel werden teuer
 - Je komplexer CPU, MMU und Cache desto schlimmer
 - Widersprüchliche Optimierungsziele

Systemaufruf



Systemfunktionen für Prozesse

- Prozesse können eine Hierarchie bilden
 - Beispiel: UNIX
 - Gegenbeispiel: Windows
- Funktionen
 - Starten
 - Warten
 - Beenden



Systemfunktionen für Prozesse

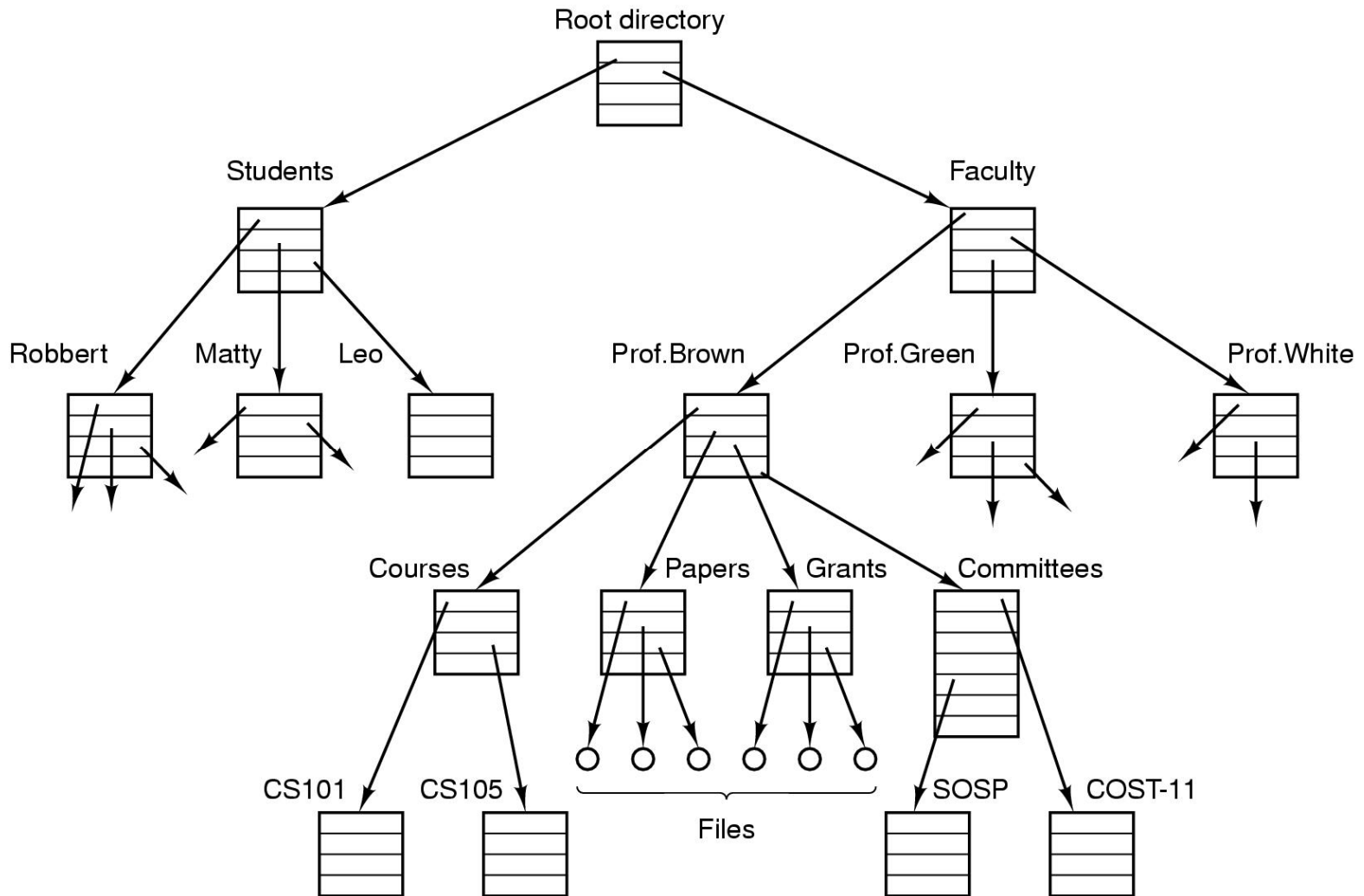
Process management

Call	Description
<code>pid = fork()</code>	Create a child process identical to the parent
<code>pid = waitpid(pid, &statloc, options)</code>	Wait for a child to terminate
<code>s = execve(name, argv, environp)</code>	Replace a process' core image
<code>exit(status)</code>	Terminate process execution and return status

Beispiel: Eine einfache Shell

```
while (TRUE) {                               /*repeat forever */
    type_prompt( );                           /*display prompt */
    /*read from terminal */
    read_command (command, parameters)
    /*fork off child process */
    if (fork() != 0) {
        /* Parent code */
        /* wait for child to exit */
        waitpid( -1, &status, 0);
    } else {
        /* Child code */
        /* execute command */
        execve (command, parameters, 0);
    }
}
```

Systemfunktionen für Dateien

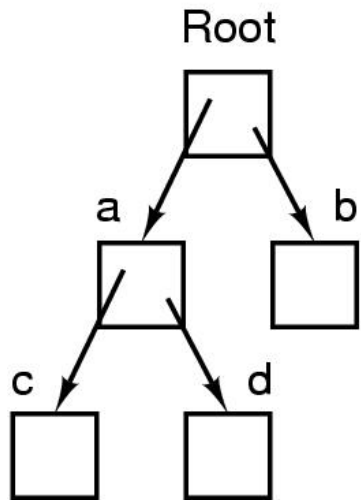


Systemfunktionen für Dateien

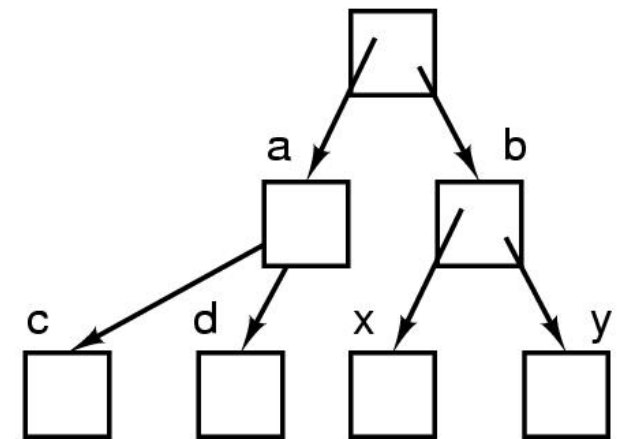
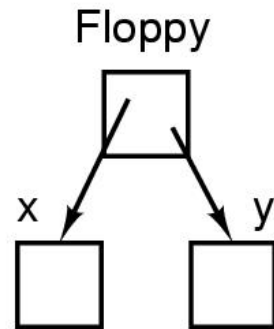
File management

Call	Description
<code>fd = open(file, how, ...)</code>	Open a file for reading, writing or both
<code>s = close(fd)</code>	Close an open file
<code>n = read(fd, buffer, nbytes)</code>	Read data from a file into a buffer
<code>n = write(fd, buffer, nbytes)</code>	Write data from a buffer into a file
<code>position = lseek(fd, offset, whence)</code>	Move the file pointer
<code>s = stat(name, &buf)</code>	Get a file's status information

Systemfunktionen für Verzeichnisse



(a)



(b)

(a) Vor dem mounten des Floopy Laufwerks

(b) Nach dem mounten

Systemfunktionen für Verzeichnisse

Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Systemfunktionen für Verzeichnisse

/usr/ast		/usr/jim		/usr/ast		/usr/jim	
16	mail	31	bin	16	mail	31	bin
81	games	70	memo	81	games	70	memo
40	test	59	f.c.	40	test	59	f.c.
		38	prog1	70	note	38	prog1

(a) (b)

(a) Vor dem linken

(b) Nach dem linken

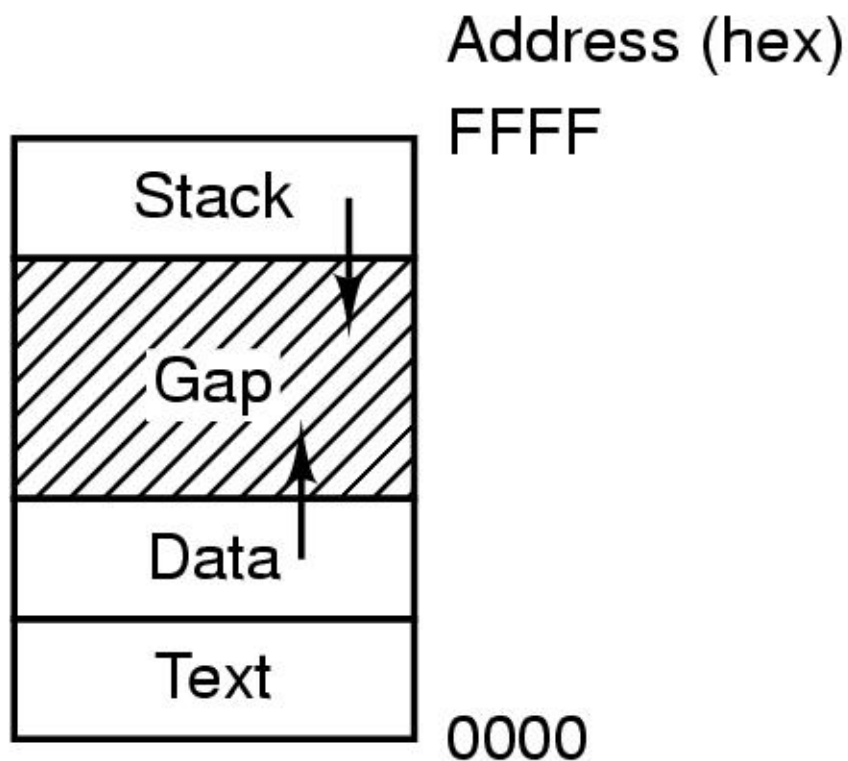
Weitere Systemfunktionen

Miscellaneous

Call	Description
<code>s = chdir(dirname)</code>	Change the working directory
<code>s = chmod(name, mode)</code>	Change a file's protection bits
<code>s = kill(pid, signal)</code>	Send a signal to a process
<code>seconds = time(&seconds)</code>	Get the elapsed time since Jan. 1, 1970

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

Funktionsaufrufe



- Ein Programm hat 3 Speicherbereiche
 - Stack
 - Daten (Heap)
 - Text (read-only)
- Stack und Daten wachsen aufeinander zu
- Out-of-memory
 - ... lange bevor Stack und Daten einander überlappen

Funktionsaufrufe

```
int foobar(int x, int y)
{
    int a=666;
    char* ptr=malloc(12);
    memcpy(ptr, "Hello World", 12);
    ...
    free(ptr);
    return x+y;
}
```

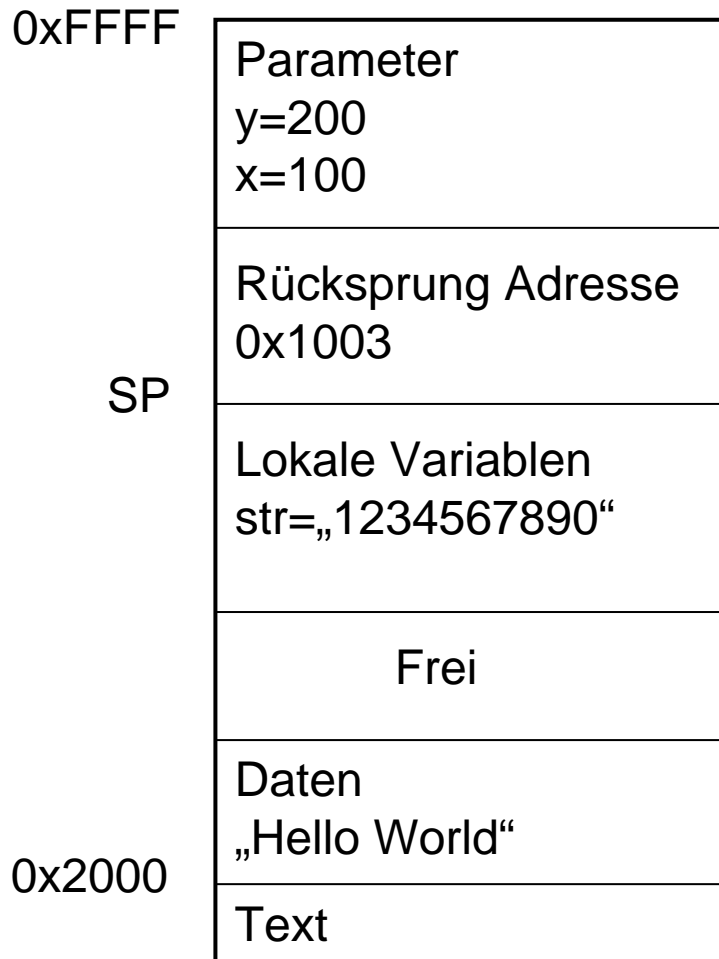
Funktionsaufrufe

0xFFFF	Parameter y=200 x=100
SP	Rücksprung Adresse 0x1003
	Lokale Variablen a=666 ptr=0x2000
	Frei
0x2000	Daten „Hello World“
	Text

```
// sp=0xFFFF
0x0F00: mv sp-2, 200
0x0F00: mv sp-4, 100
0x0F00: sub sp, 4
0x0F00: mv sp-2, 0x1003
0x1000: jmp 0x1500
0x1003: add sp, 4

0x1500: mv sp-2, 666
0x1506: ... // ax=x+y
0x1580: return
```

Stack Overflow

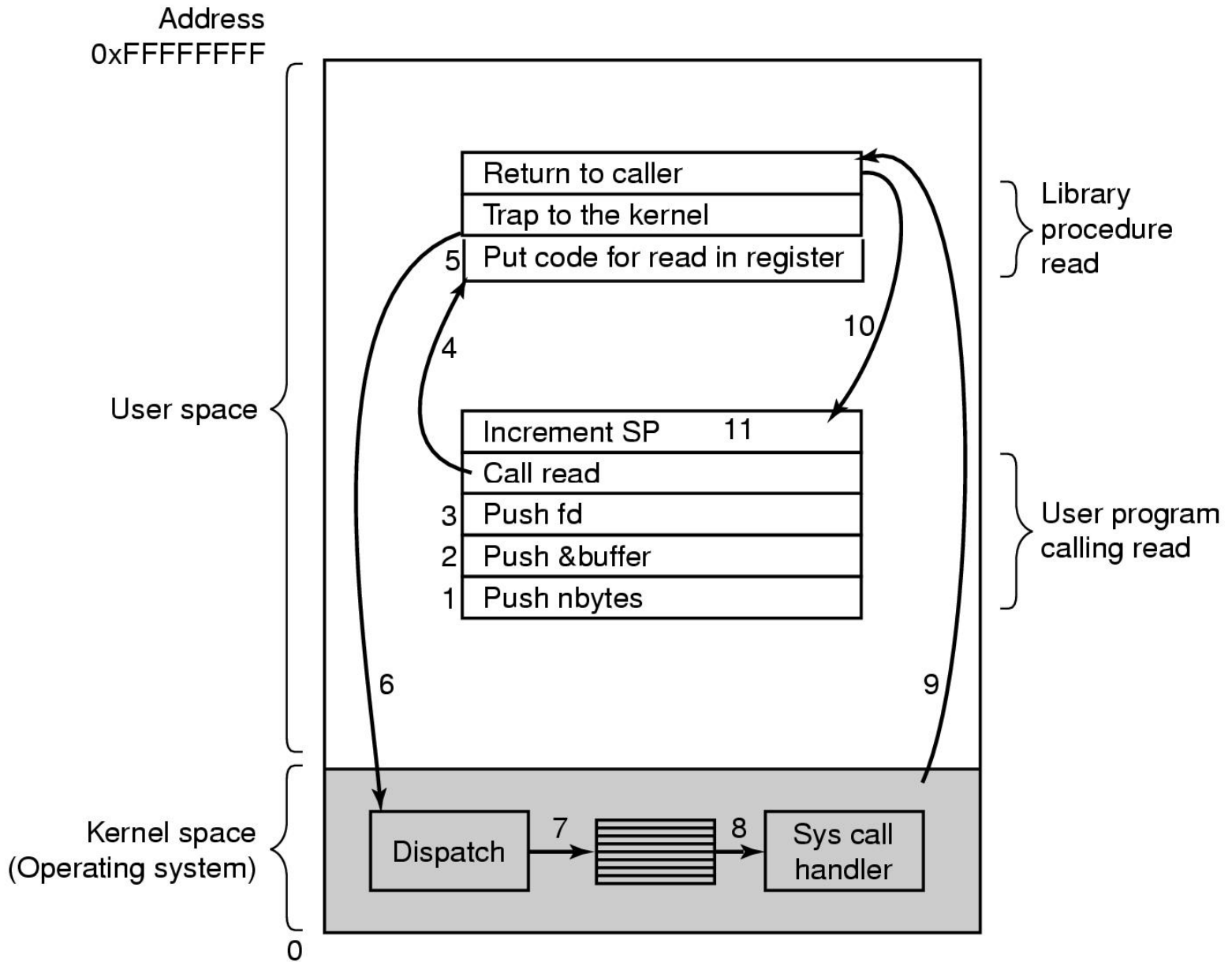


```
void foo()  
{  
    char str[10];  
    char* input = ...;  
    sprintf(  
        &str,  
        "Hallo %s",  
        input);  
    dosomething(str);  
}
```

Systemaufruf

- Unterschied zum Systemaufruf
 - Direkter jmp nicht möglich
 - Wechsel in Kernel Mode erforderlich
 - Trap oder Interrupt erforderlich
 - Aufruf über eine Bibliotheksfunktion
- Beispiel
 - `int read(int fd, char* buffer, int nbytes)`
 - Was passiert bei folgendem Aufruf
`count = read(fd, buffer, nbytes)`

... nächste Folie



Spezialdateien

- Geräte werden auch als Dateien angesprochen
 - /dev/lp
 - /dev/sound
 - /dev/mouse
 - /dev/hdd1
- Zeichenorientierte Geräte
 - Lesen/Schreiben aber keine Positionierung (seek)
- Blockorientierte Geräte
 - Lesen/Schreiben eines Blocks
 - Auswählen eines Blocks (seek)

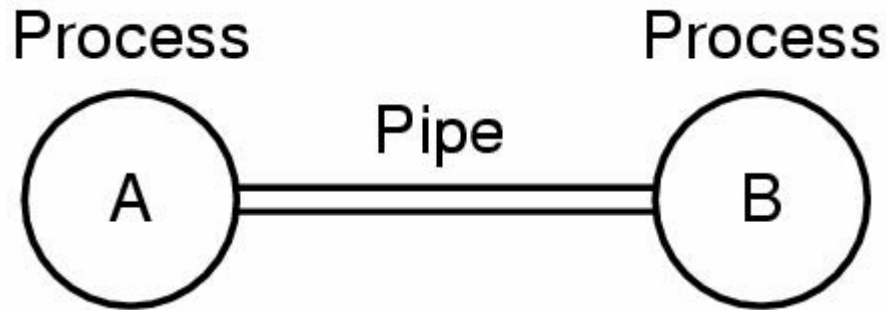
Ein-/Ausgabe

- Prozesse (zu Zeiten als UNIX modern war)
 - Ließt eine Eingabe
 - Schreibt eine Ausgabe
 - Erzeugt Fehlermeldungen
- Typischerweise 3 Standard „Dateien“
 - stdin
 - stdout
 - stderr
- Die Standard Dateien sind immer offen
 - stdin/stdout kann ein Terminal sein
 - stdin/stdout kann ein anderes Programm sein

Ein-/Ausgabe Umleitung

- Umlenken auf eine richtige Datei
 - `sort <infile >outfile`
 - `stdin = infile, stdout = outfile`
- Verbinden von Prozessen
 - `cat file1 file2 file3 | sort >/dev/lp`
 - `cat: stdin = Terminal, stdout = sort`
 - `sort: stdin = cat, stdout = /dev/lp`
- Ausführung im Hintergrund
 - `cat file1 file2 file3 | sort >/dev/lp &`
 - `cat: stdin = /dev/null, stdout = sort`
 - `sort: stdin = cat, stdout = /dev/lp`

Kommunikation über Pipes



- Zwei Prozesse kommunizieren über eine Pipe
- Pipes verhalten sich wie Dateien
- Es erfolgt aber keine Speicherung auf Platte

Kommunikation über Pipes

- Named Pipe
 - Erscheint im Dateisystem `/tmp/myapp/pipe1`
 - Security über Dateisystemsicherheit
 - Prozesse müssen nicht „verwandt“ sein
- Anonymous Pipe
 - Wird vom Elternprozess erzeugt
 - Kindprozess erbt die offenen Dateihandles
 - Ermöglicht komplexere Kommunikation zwischen Eltern- und Kindprozessen

Dateisystem-Sicherheit in UNIX

- Jede Datei gehört ...
 - 1) Einem User
 - 2) Einer Gruppe
- User Rechte
 - Read / Write / Execute
- Gruppen Rechte
 - Read / Write / Execute
- Rechte der Allgemeinheit
 - Read / Write / Execute
- Notation
 - r-xr-x---
 - rw-r--r--

Dateisystem-Sicherheit in UNIX

- Fast alles in UNIX ist eine Datei
 - Richtige Dateien
 - Geräte (/dev Verzeichnis)
 - Pipes
 - Terminal Ein-/Ausgabe
 - Systemkonfiguration (/etc Verzeichnis)
- Einheitliches Sicherheitskonzept
 - Wenn das Dateisystem sicher ist, dann auch der Rest
 - Wenn nicht, dann ... ☹
 - Beim Umgang mit Dateirechten Vorsicht!

Blockierende Ein-/Ausgabe

- Systemaufrufe können blockieren
 - Rückkehr nach Vollendung oder Fehler
- Beispiel
 - `int count = read(fd, &buffer, 100);`
 - Blockiert bis
 - a) 100 Bytes gelesen wurden
 - b) Alle ≤ 100 Bytes gelesen wurden (EOF)
 - c) Ein Fehler aufgetreten ist (errno auslesen)
- Vorteil
 - Einfache Programmierung
- Nachteil
 - Derweil keine Userinteraktion (außer CTRL+C)

Nicht-Blockierende Ein-/Ausgabe

- Dateien ermöglichen Zugriff ohne Blockieren
 - Sofortige Rückkehr
 - Schreiben/Lesen von so vielen Daten wie gewünscht und möglich
- Beispiel
 - `int count = read(fd, &buffer, 100);`
 - Nach Aufruf wurden
 - a) 100 Bytes gelesen
 - b) alle $0 \leq \text{count} \leq 100$ verfügbaren Bytes gelesen
 - c) ein Fehler signalisiert (errno auslesen)
- Nachteil
 - Polling: Man muss immer wieder versuchen zu lesen

Nicht-Blockierende Ein-/Ausgabe

- Es geht auch ohne Polling
 - Man wartet blockierend auf irgendeine Datei
 - Zwar blockierend aber ...
... keine Eingabe -> keine wartenden User

- Das `select` Kommando (vereinfacht)

```
fd1 = open(„file1“);
```

```
fd2 = open(“file2”);
```

```
add_read(fdset, fd1); add_read(fdset, fd2);
```

```
while( true ) {
```

```
    select(fdset); // Blockierender Aufruf
```

```
    if ( can_read(fdset, fd1) ) read(fd1) // non-blocking
```

```
    else read(fd2) // non-blocking
```

Zusammenfassung

- Computer Hardware
- Geräte-Treiber/-Controller
- Prozesse / Prozesswechsel
- Speicherverwaltung
- Funktionsaufrufe
- Systemaufrufe
- Ein-/Ausgabe
- Sicherheit