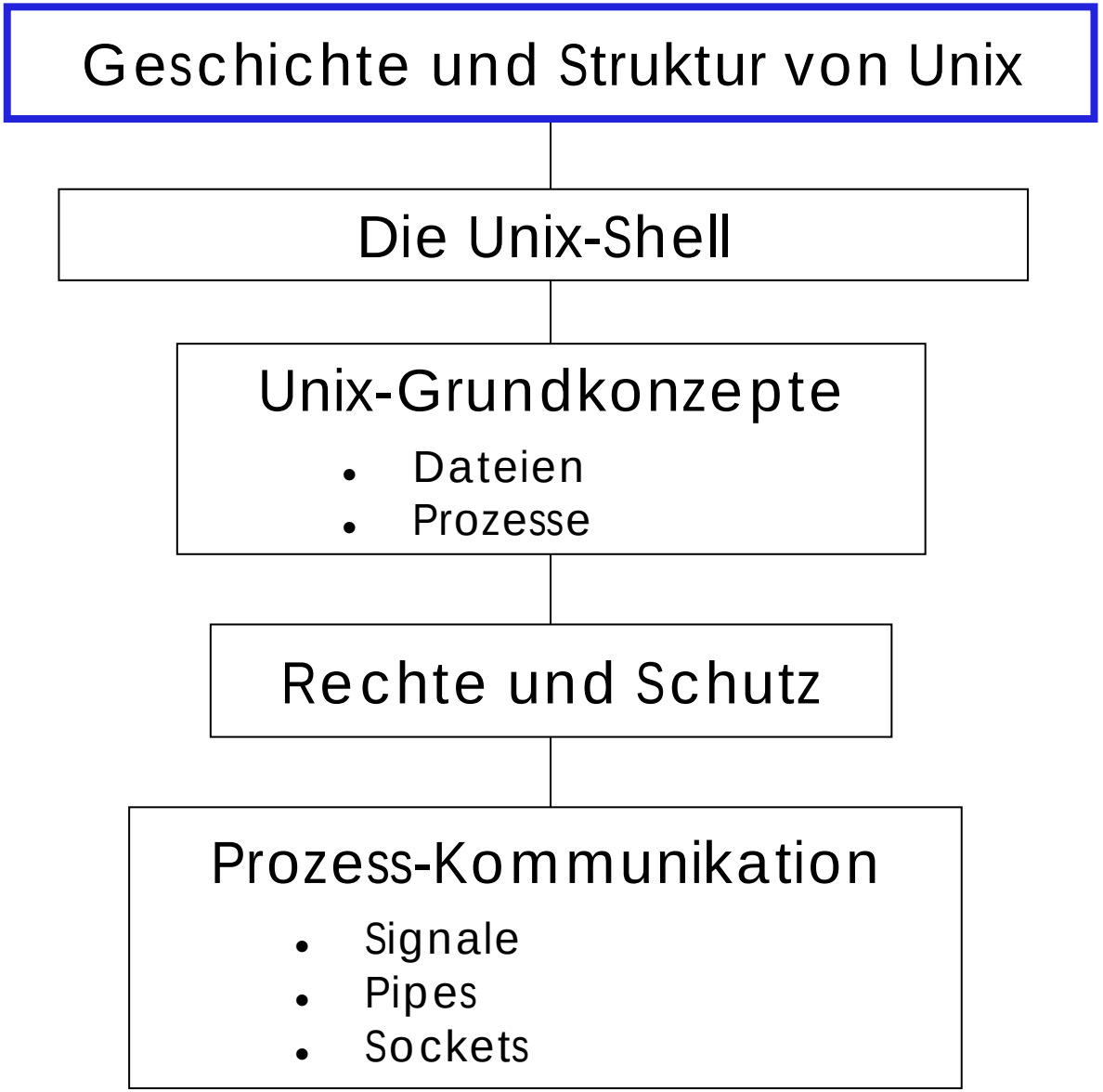
Fallbeispiel Unix

Betriebssysteme

Hermann Härtig
TU Dresden
WS 2007/08

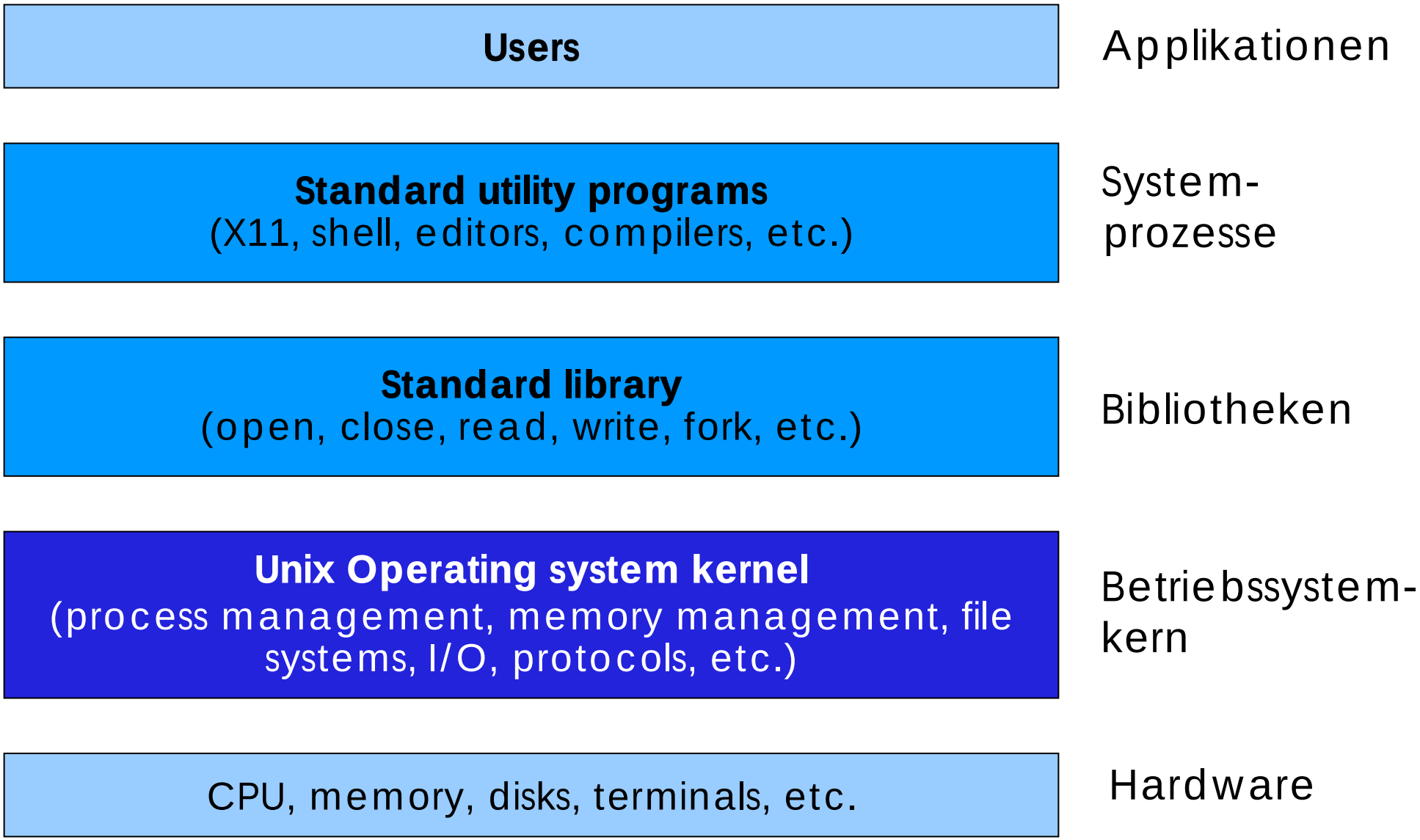
Wegweiser


Geschichte und Struktur von Unix
Die Unix-Shell
Unix-Grundkonzepte
Dateien
Prozesse
Rechte und Schutz
Prozess-Kommunikation
Signale
Pipes
Sockets

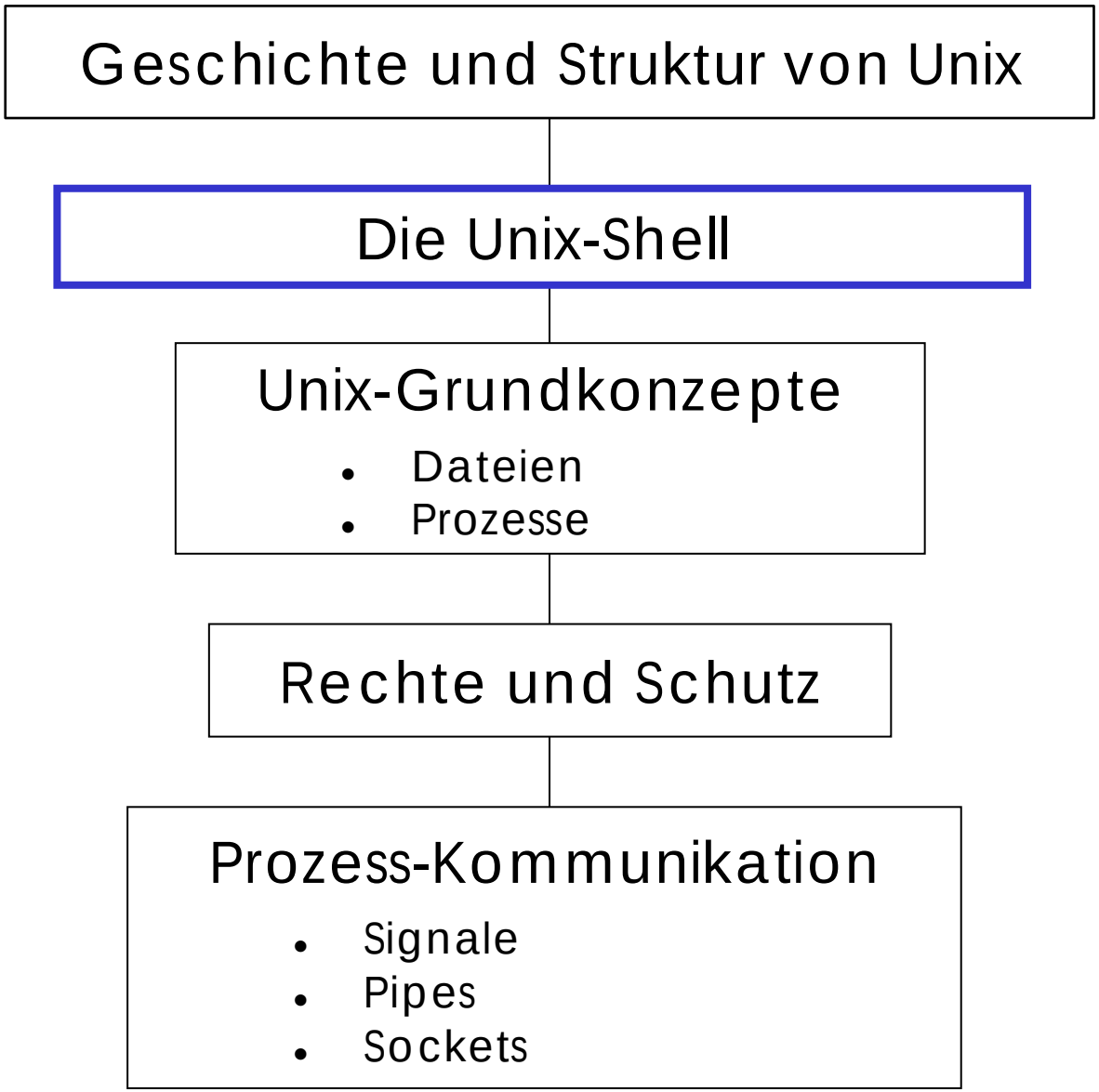
Unix - Story

196x MULTICS (MIT)	viele wichtige Ideen, aber „Fehlschlag“
1971 Ken Thompson	"UNICS" auf PDP-7 (First Edition)
1973 Dennis Ritchie + KT	C, rewrite in C
1974 TR74	The Unix Time-Sharing System
1975	Sixth Edition, weite Verbreitung
1977 Richards	Portierung auf Interdata (32 Bit)
1979	Bourne-Shell, PCC
1980 Bill Joy, et. al.	Berkeley SD 4, "vi"
198x	virtueller Speicher, Netzwerke
1986 IEEE	Posix
1983 Randell, et. al.	Newcastle Connection

Grobstruktur Unix



Wegweiser

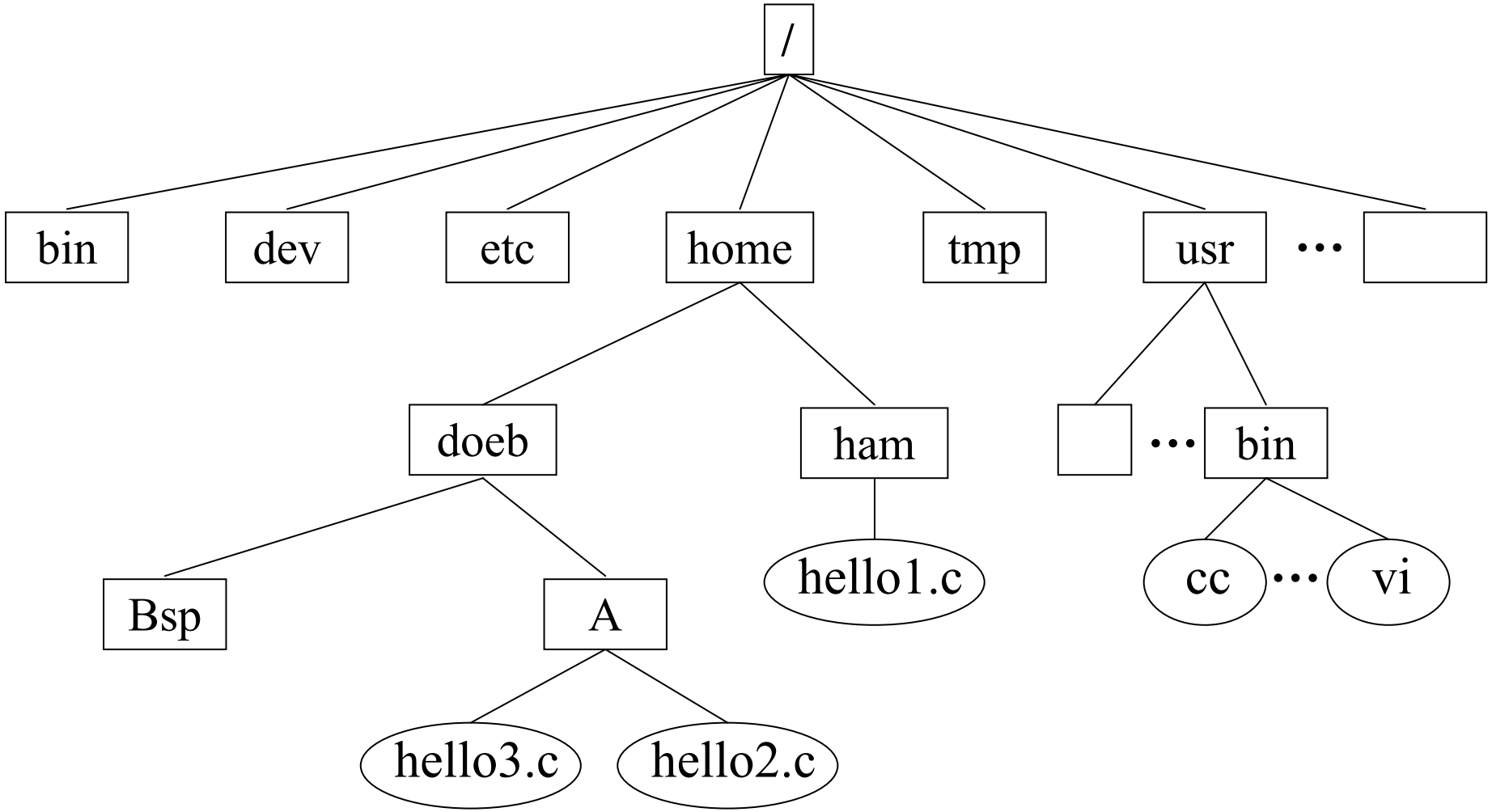

Geschichte und Struktur von Unix
Die Unix-Shell
Unix-Grundkonzepte
Dateien
Prozesse
Rechte und Schutz
Prozess-Kommunikation
Signale
Pipes
Sockets

Ausgewählte Shell-Kommandos

pwd	print name of working directory
ls	list a directory
mkdir	make a directory
cd	change directory
mv	move (rename) files
rm	remove (delete)
chmod	change file access permissions
cp	copy
ln	make links between files
cat	concatenate files and print on standard output
less	opposite of **more**
ps	process list
man	browse manual pages

Syntax: name optionen argumente

Verzeichnisstruktur



Programmentwicklung

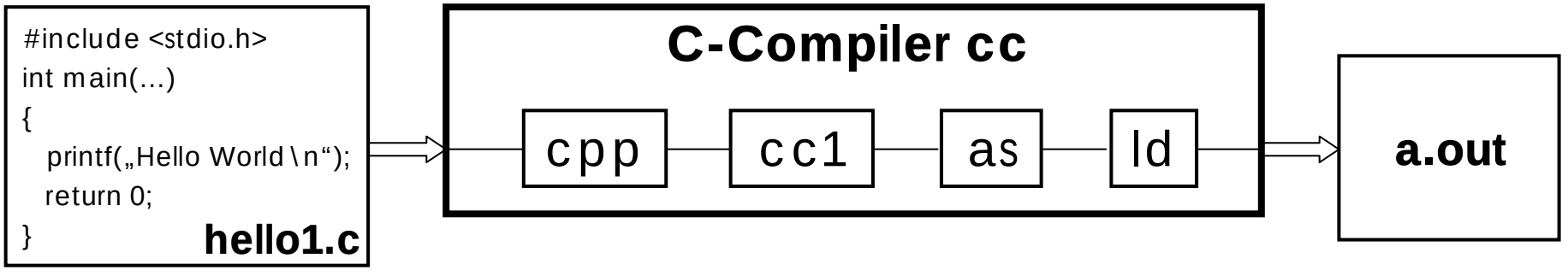
hello1.c

```
#include <stdio.h>                          Präprozessor-Direktive
int main(int argc, char *argv[])            Eintrittspunkt
{
    printf(„Hello World\n“);
    return 0;                               exit-Status (0: erfolgreich)
}
```

```
mkdir Bsp
cp /home/ham/hello1.c Bsp
cd Bsp
ls –l
chmod g+r hello1.c
cat hello1.c
```

Schritte des C-Compilers

- cc ist ein „Frontend“ für folgende Teile:
  - Präprozessor cpp: .c ⇒ .i C ohne Makros
  - Compiler cc1: (.i) .c ⇒ .s Assemblerquelltext
  - Assembler as: .s ⇒ .o Objektdatei
  - Linker ld: .o ⇒ a.out Programm
- .c, .i und .s sind Text ⇒ Texteditor, z. B. vi
- .o ist Maschinencode ⇒ nm, objdump, objcopy
- a.out ist ausführbar ⇒ ldd, nm, readelf, gdb



Schritte des C-Compilers

- Präprozessor:

  cc -E -o hello1.i hello1.c

  `less hello1.i`

- Compiler:

  cc -S -o hello1.s hello1.i

- Assembler:

  cc -c -o hello1.o hello1.s

  `objdump -Sdl hello1.o`

- Linker:

  cc -o hello1 hello1.o

hello1

ls -l h*

Rolle des Betriebssystems

```
void _start(void) {   . . .
    status = main(argc, argv);
        int main(int argc, char *argv[ ]) {
            printf("Hello World\n");
                printf(    ) {
                    vfprintf(stdout, "Hello World\n");   . . .
                        write(stdout, "Hello World\n", 12);
                            eax=4,
                            ebx=1,
                            ecx="Hello World\n",
                            edx=12,
                            int 0x80; /vmlinuz
                }           /usr/lib/libc.a bzw. /lib/libc.so.6
            return 0;
        }           hello1.o
    exit(status);
}               crt1.o
```

Schnittstelle zum Betriebssystem

273 Systemaufrufe unter Linux *(siehe /usr/include/asm/unistd.h)*

Kernel 2.2/2.4/2.6: 190 ⇒ 237 ⇒ 273 Systemaufrufe

restart_syscall **exit** **fork** read **write** open close waitpid creat link unlink execve chdir time mknod chmod lchown break oldstat lseek getpid mount umount setuid getuid stime ptrace alarm oldfstat pause utime stty gtty access nice ftime sync kill rename mkdir rmdir dup pipe times prof brk setgid getgid signal geteuid getegid acct umount2 lock ioctl fcntl mpx setpgid ulimit oldolduname umask chroot ustat dup2 getppid getpgrp setsid sigaction sgetmask ssetmask setreuid setregid sigsuspend sigpending sethostname setrlimit getrlimit getrusage gettimeofday settimeofday getgroups setgroups select symlink oldlstat readlink uselib swapon reboot readdir mmap munmap truncate ftruncate fchmod fchown getpriority setpriority profil statfs fstatfs ioperm socketcall syslog setitimer getitimer stat lstat fstat olduname iopl vhangup idle vm86old wait4 swapoff sysinfo ipc fsync sigreturn clone setdomainname uname modify_ldt adjtimex mprotect sigprocmask create_module init_module delete_module get_kernel_syms quotactl getpgid fchdir bdflush sysfs personality afs_syscall setfsuid setfsgid _llseek getdents _newselect flock msync readv writev getsid fdatasync _sysctl mlock munlock mlockall munlockall sched_setparam sched_getparam sched_setscheduler sched_getscheduler sched_yield sched_get_priority_max sched_get_priority_min sched_rr_get_interval nanosleep mremap setresuid getresuid vm86 query_module poll nfsservctl setresgid getresgid prctl rt_sigreturn rt_sigaction rt_sigprocmask rt_sigpending rt_sigtimedwait rt_sigqueueinfo rt_sigsuspend pread64 pwrite64 chown getcwd capget capset sigaltstack sendfile getpmsg putpmsg vfork ugetrlimit mmap2 truncate64 ftruncate64 stat64 lstat64 fstat64 lchown32 getuid32 getgid32 geteuid32 getegid32 setreuid32 setregid32 getgroups32 setgroups32 fchown32 setresuid32 getresuid32 setresgid32 getresgid32 chown32 setuid32 setgid32 setfsuid32 setfsgid32 pivot_root mincore madvise madvise1 getdents64 fcntl64 gettid readahead setxattr lsetxattr fsetxattr getxattr lgetxattr fgetxattr listxattr llistxattr flistxattr removexattr lremovexattr fremovexattr tkill sendfile64 futex sched_setaffinity sched_getaffinity set_thread_area get_thread_area io_setup io_destroy io_getevents io_submit io_cancel fadvise64 exit_group lookup_dcookie epoll_create epoll_ctl epoll_wait remap_file_pages set_tid_address timer_create timer_settime timer_gettime timer_getoverrun timer_delete clock_settime clock_gettime clock_getres clock_nanosleep statfs64 fstatfs64 tgkill utimes fadvise64_64

Programmentwicklung

hello2.c

```
#include <unistd.h>
int main(int argc, char *argv[])
{
    write(1, „Hello World\n“, 12)
          ↑                    ↑
      STDOUT_FILENO       HELLO_LENGHT
    return 0;
}
```

cc hello2.c
hello2

Systemaufrufe unter Linux

- Parameter werden in Registern übergeben
  eax = Nummer des Systemaufrufs (1–273)
- Systemaufrufe haben 0 bis 5 Parameter
  ebx = 1. Parameter
  ecx = 2. Parameter
  edx = 3. Parameter
  esi = 4. Parameter
  edi = 5. Parameter
- Übergang in den Kern durch Softwareinterrupt „int 80h“
- Rückgabewert wird in Register eax übergeben.
- Bibliothek libc enthält Hüllfunktionen („Wrapper“)

  z. B.: #include <unistd.h>

  ssize_t **write**(int fd, const void *buf, size_t count);
  ebx ecx edx

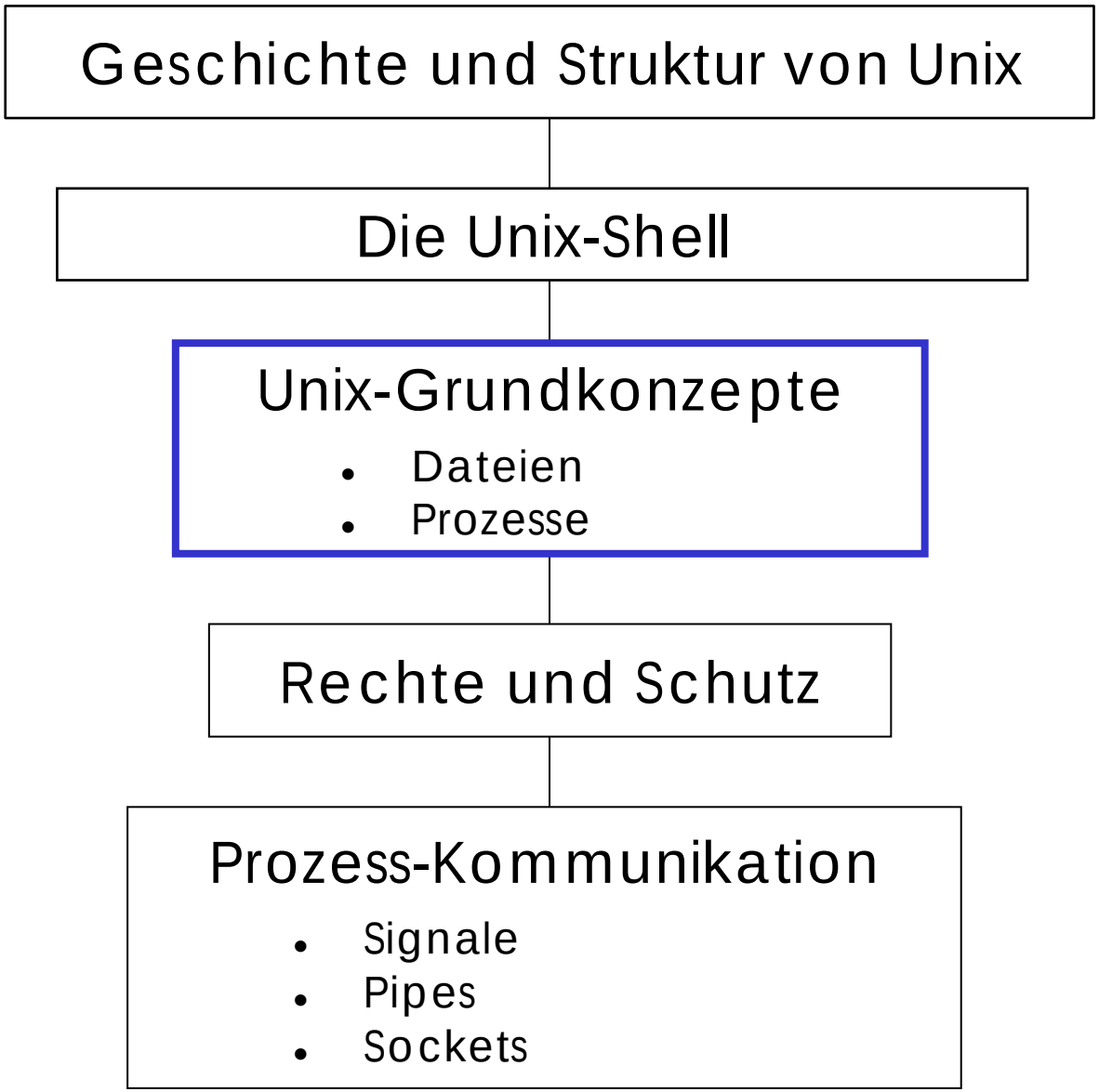
Prozeßstruktur

hello3.c

```
#include <stdio.h>
int main(void)
{
   int err = 0;
   printf("Bitte Enter-Taste drücken...\n");
   err = getchar();
   if (err < 0)
         perror("getchar");
   return 0;
}
```

```
hello3 &
ps -l
```

Wegweiser


Geschichte und Struktur von Unix
Die Unix-Shell
Unix-Grundkonzepte
Dateien
Prozesse
Rechte und Schutz
Prozess-Kommunikation
Signale
Pipes
Sockets

Shell-Ebene: Dateien

Pfadnamen

- **bla/xy** (relativ zu Prozess-Kontext)
  /bla/xy (absolut)

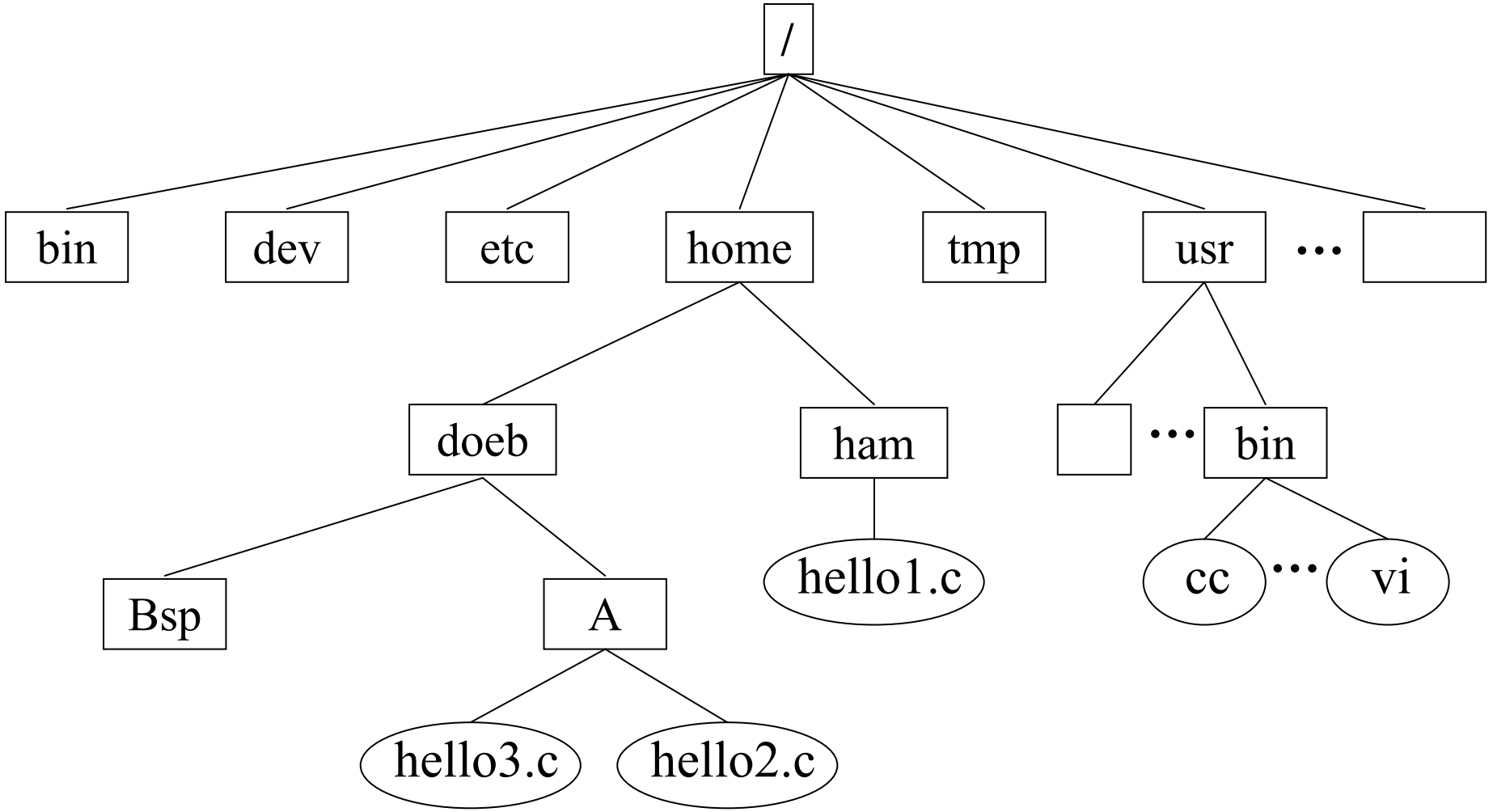
➔ Pfadnamen häufig als Parameter : **cp src dest**

➔ wichtig: viele Konventionen bzgl. Dateinamen

Links

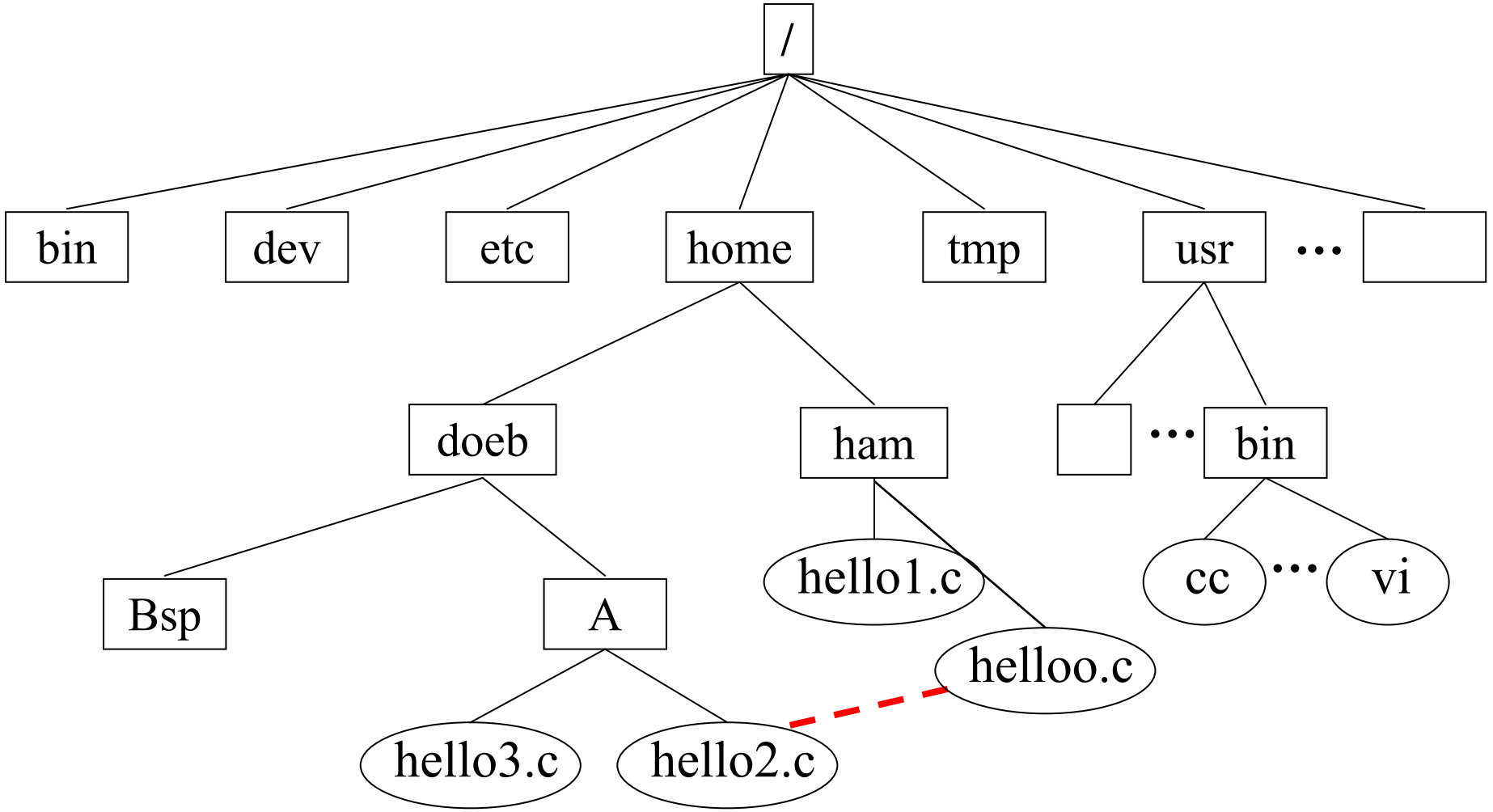
- **ln a /etc/xy**
  Datei **a** auch unter dem Namen **/etc/xy** zugreifbar
- **rm a**
  Datei nicht mehr unter dem Namen **a** zugreifbar
  aber nach wie vor unter **/etc/xy**
- **rm /etc/xy**
  kein Name mehr für Datei vorhanden → gelöscht

Links



`ln hello2.c /home/ham/helloo.c`

Links



`ln hello2.c /home/ham/helloo.c`

Dateien: Kernaufrufe (System Calls)

- **fd = open(name, flags, mode)**

  Öffnen der Datei **name**, liefert „file descriptor“

  Flags: O_RDONLY, O_RDWR, ...

  O_CREAT: Datei wird ggf. erzeugt

  mode bestimmt Zugriffsrecht der Datei bei Erzeugung

  Konvention für File-Descriptor :

  0 standard in
  1 standard out
  2 standard error

- **bytes = read(fd, buf, size)**

  liest max. **size** Bytes von **fd** nach **buf**

- **bytes = write(fd, buf, size)**

- **lseek(fd, offset,** absolut/relativ**)**

Ein-/Ausgabe

- Integration in Dateisystem: special files
  - Verzeichnis: /dev
  - die gleiche Schnittstelle: **`cat file > /dev/lp`**
  - automatische Übernahme des Schutzkonzeptes

➔ Block special files vs. Character special files

- zahlreiche zusätzliche Systemaufrufe zur Einstellung/Manipulation

Prozesse in Unix

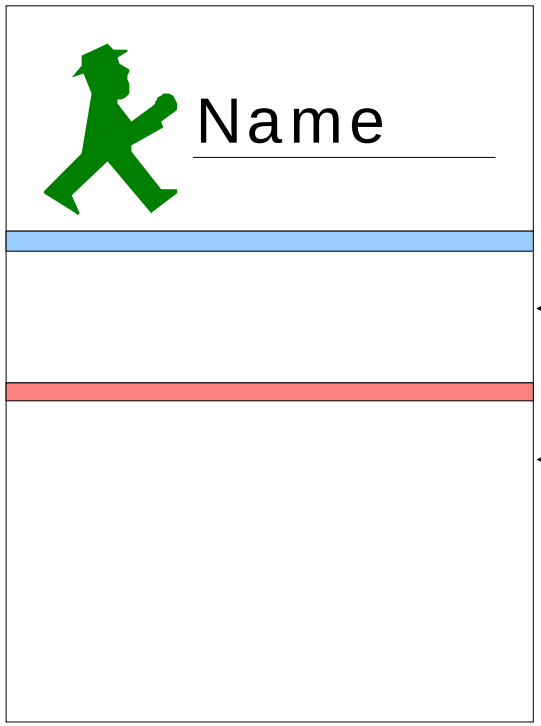
Unix-Prozess:

- ein Programm
- ein Thread
- ein Adressraum
- “is a program in execution“
- "Besitzer" aller Betriebsmittel (Speicher, Dateien, ...)
- repräsentiert Prinzipale (durch Uld/Gld)

Viele Prozesse pro Station

- Benutzerprozesse
- Hintergrund-System-Prozesse (daemon)

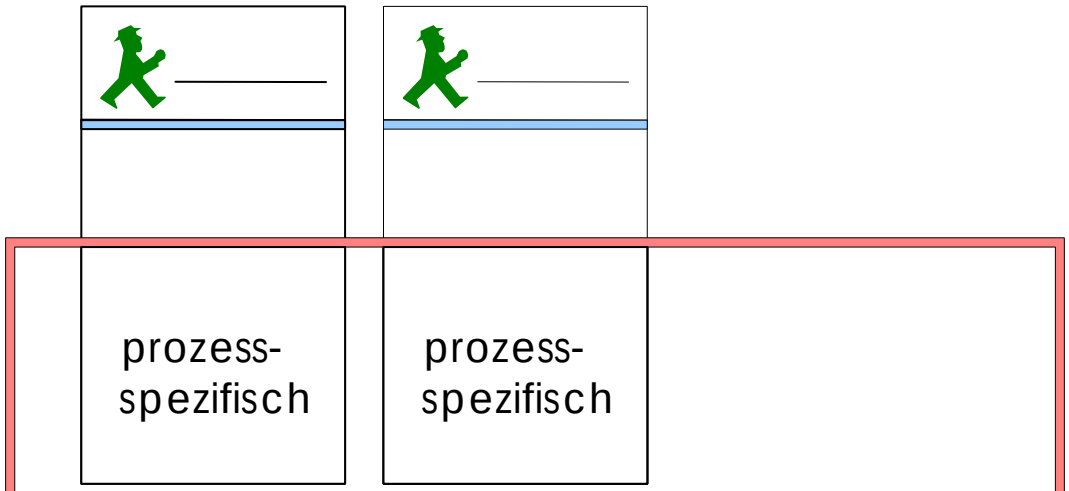
Darstellung Unix–Prozesse



User-Mode-Daten

dem Prozess zugeordnete Betriebsmittel, werden nur im Kern manipuliert

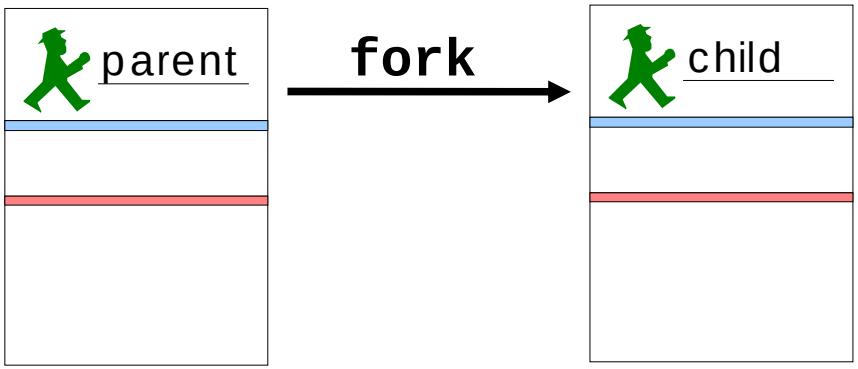
Kern-Adressraum



- weitere Datenstrukturen des Kerns - z.B. Tabelle der offenen Dateien
- Kern-Code

Kernaufruf: Erzeugung von Prozessen

```
pid = fork();
//Erstellen einer exakten Kopie des Aufrufers
//inklusive Adressraum, aller Filedeskriptoren ...
```




```
if (pid == 0) {
    //child code
} else {
    printf(„new child: PID = %d\n“, pid); //parent code
}
```

Weitere Kernaufrufe

`s = exec(file, argument, environment)`

- ersetzt Speicherinhalt durch Inhalt von `file` und führt `file` aus
- schreibt Argumente und Umgebungsvariablen an Anfang des Kellersegments

`exit(status)`

- wartet bis Eltern-Prozess `wait` ausführt (zombie)
- überträgt Ergebnis zum Eltern-Prozess

`s = waitpid(pid, status,` block or run**`)`**

- wartet auf Ende des Kindprozesses `pid`
  bei `pid` = -1 auf irgendein Kind
- Ergebnis des Kindprozesses in `status`

Beispiel: Shell mittels fork/exec

```
read (command, params);

pid = fork();
// erzeugt Kopie des Aufrufers (d.h. der Shell)
// Kinder erhalten fd's des Eltern-Prozesses
// Kind-Prozess setzt die Abarbeitung hinter fork fort

if (pid < 0){
     // Fehlerbehandlung
} else if (pid != 0) {
     // Eltern-Prozess
     waitpid (pid, &status, 0);
     // warte auf Kind-Prozess
} else {
     // Kind
     exec (command, params, env);
}
```

Beispiel: Shell mittels fork/exec

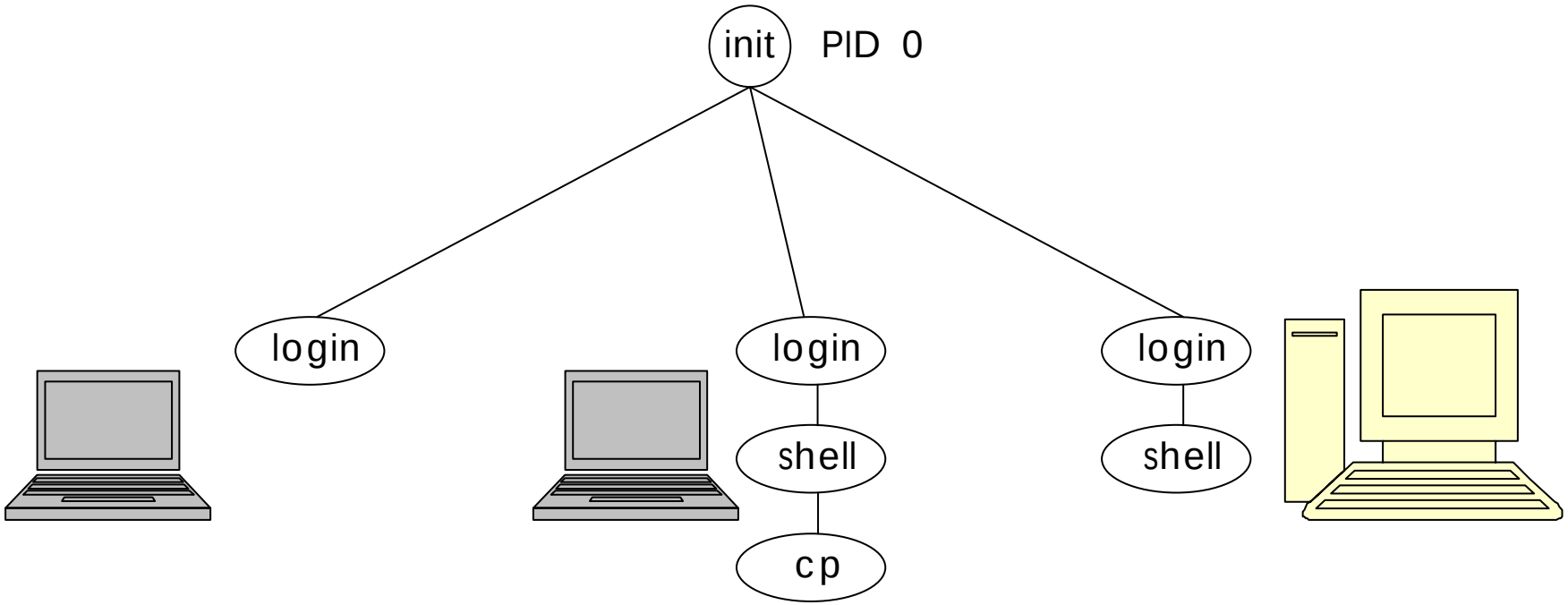
```
read (command, params);

pid = fork();
// erzeugt Kopie des Aufrufers (d.h. der Shell)
// Kinder erhalten fds des Eltern-Prozesses
// Kind-Prozess setzt die Abarbeitung hinter fork fort

if (pid < 0){
     // Fehlerbehandlung
} else if (pid != 0) {
     // Eltern-Prozess
     waitpid (pid, &status, 0);
     // warte auf Kind-Prozess
} else {
     // Kind
     exec (command, params, env);
}
```

Systemstart und Login

`init` – der erste Prozeß



Kernaufrufe (System Calls)

Beispiel

```
bytes = read (10, buf, anzahl);
```

- Ergebnis: Anzahl der gelesenen Bytes
- Konvention: -1 → Fehler
- Meldung der Fehlerursache: **errno**

Kernaufruf im Detail

Benutzerprozess

```
read(...) {

    //Parameteraufbereitung
        ...
    call = read;
    TRAP
```

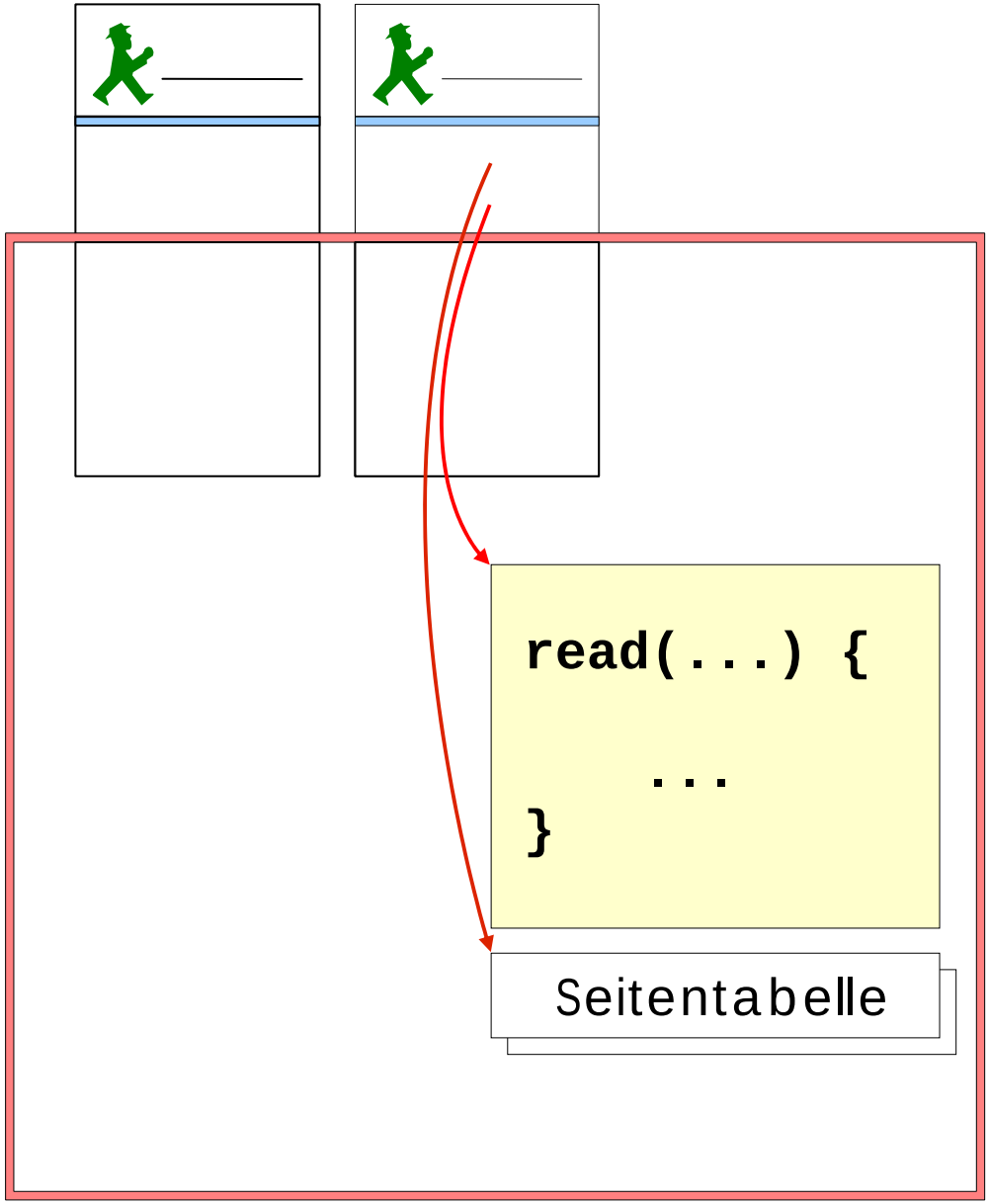
```
    //weiter geht's
    ...
}
```

Kern

```
//TRAP-Entry
    switch (call) {
      case read:
        ...
          return from trap
      case write: ...
```

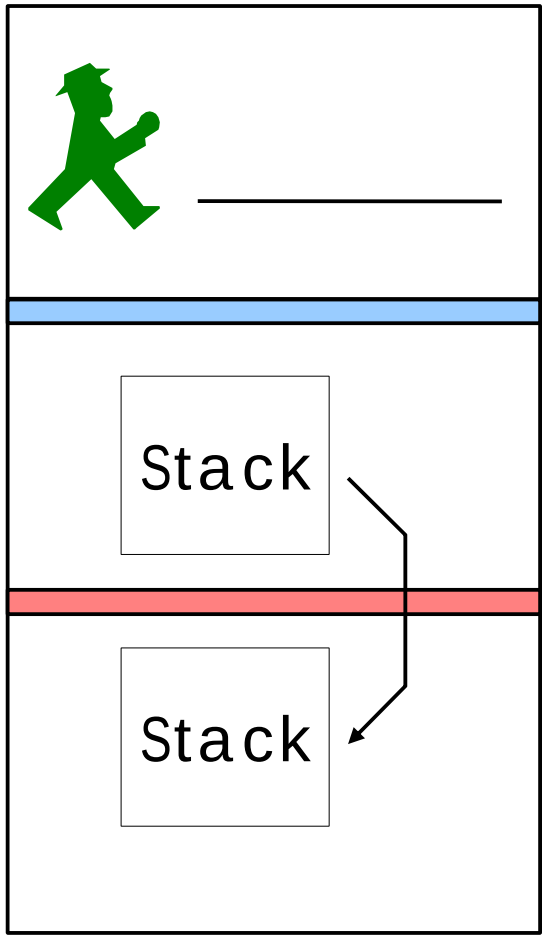
- User-Mode: kein Zugriff auf Kern-Adressraum
- Kernel-Mode: Zugriff auf Kern- und Benutzer-Adressraum

Schutz des Kerns ?



- Wie wird der Kern sicher aufgerufen ?
- Wie wird der Kern-Adressraum geschützt ?
  Schutz z.B. der File-Deskriptoren
- Bei multithreaded Prozessen (nicht klassisches Unix) :
  Wie wird verhindert, dass ein Thread den Keller des anderen manipuliert, während er gerade im Kern ist?

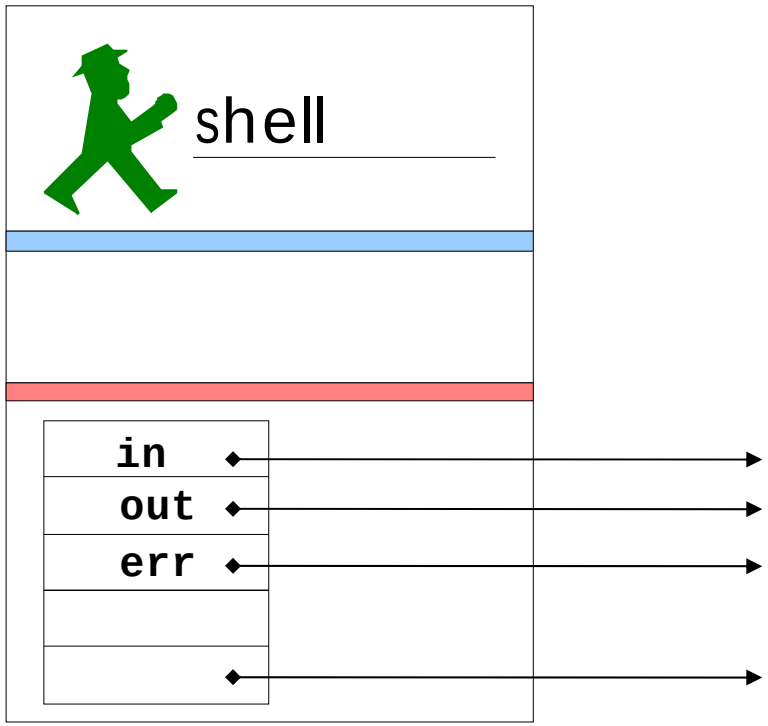
Zwei Keller pro Prozess: User, Kernel



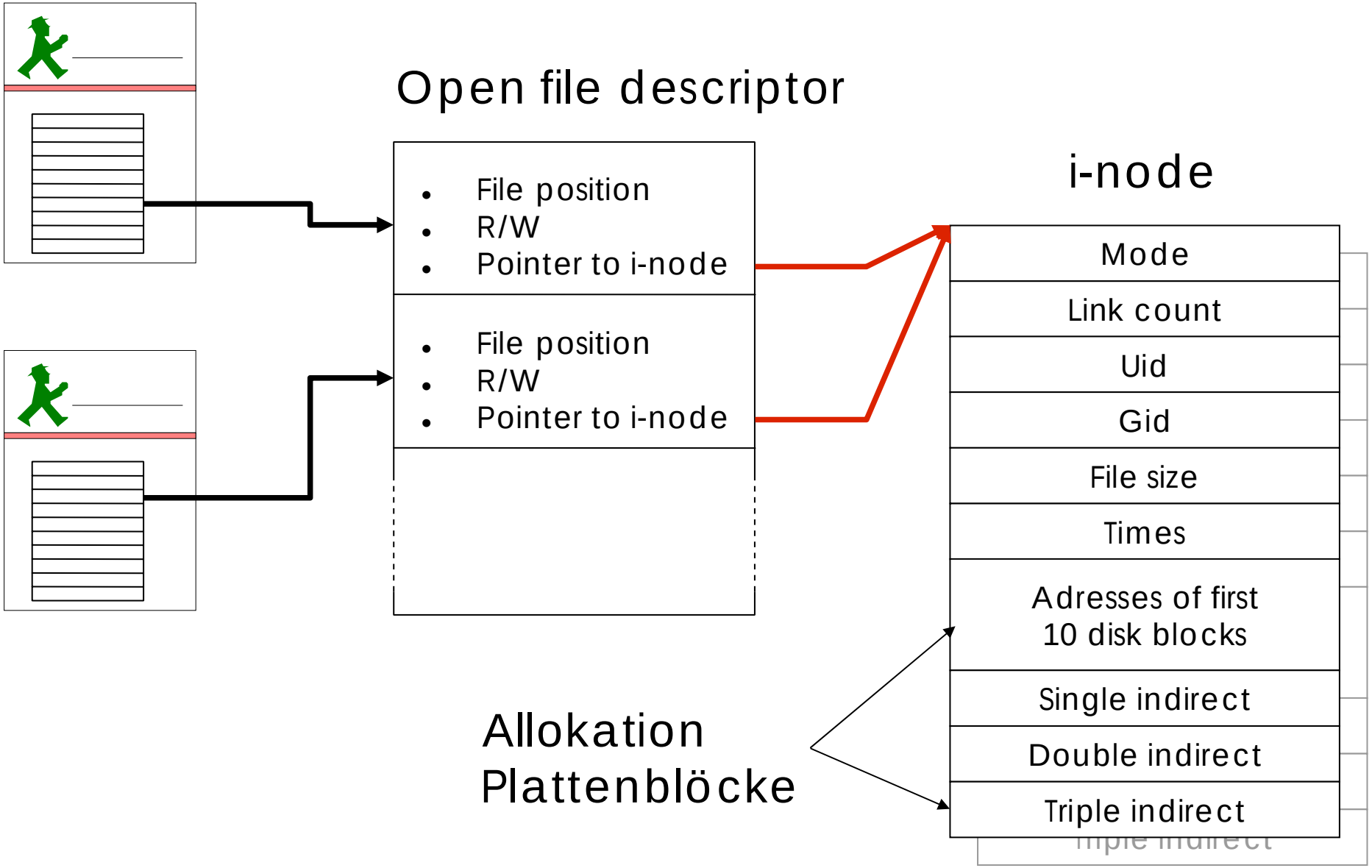
Bei Systemcalls

- wird auf den Kern-Keller geschaltet
- wird der Kernmodus eingeschaltet
- dadurch wird der Kern-Adressraum sichtbar
- und an eine feste Einsprungstelle gesprungen und von dort kontrolliert verzweigt

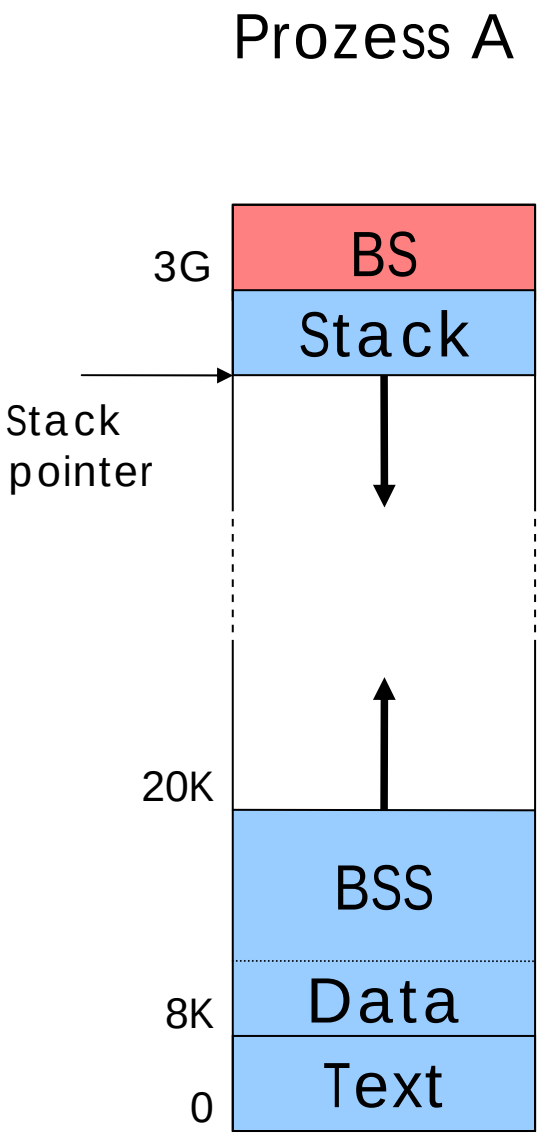
Kernschnittstelle: File-Deskriptoren



Kerninterne Datenstrukturen



Das Unix-Speichermodell



Daten-Segment

- globale Daten eines Programms
  - data: initialisierte Daten
  - bss: per Konvention mit 0 initialisiert erweiterbar duch **`brk`** Systemaufruf

Textsegment

- enthält Maschinencode
- execute only (auf X86 auch read Zugriff)
- erste Seite frei, um uninitialisierte Pointer zu entdecken
- "shared text"

Keller-Segment

- Keller (Stack)
- enthält Parameter und Kontext (environment)