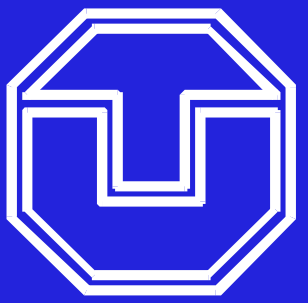
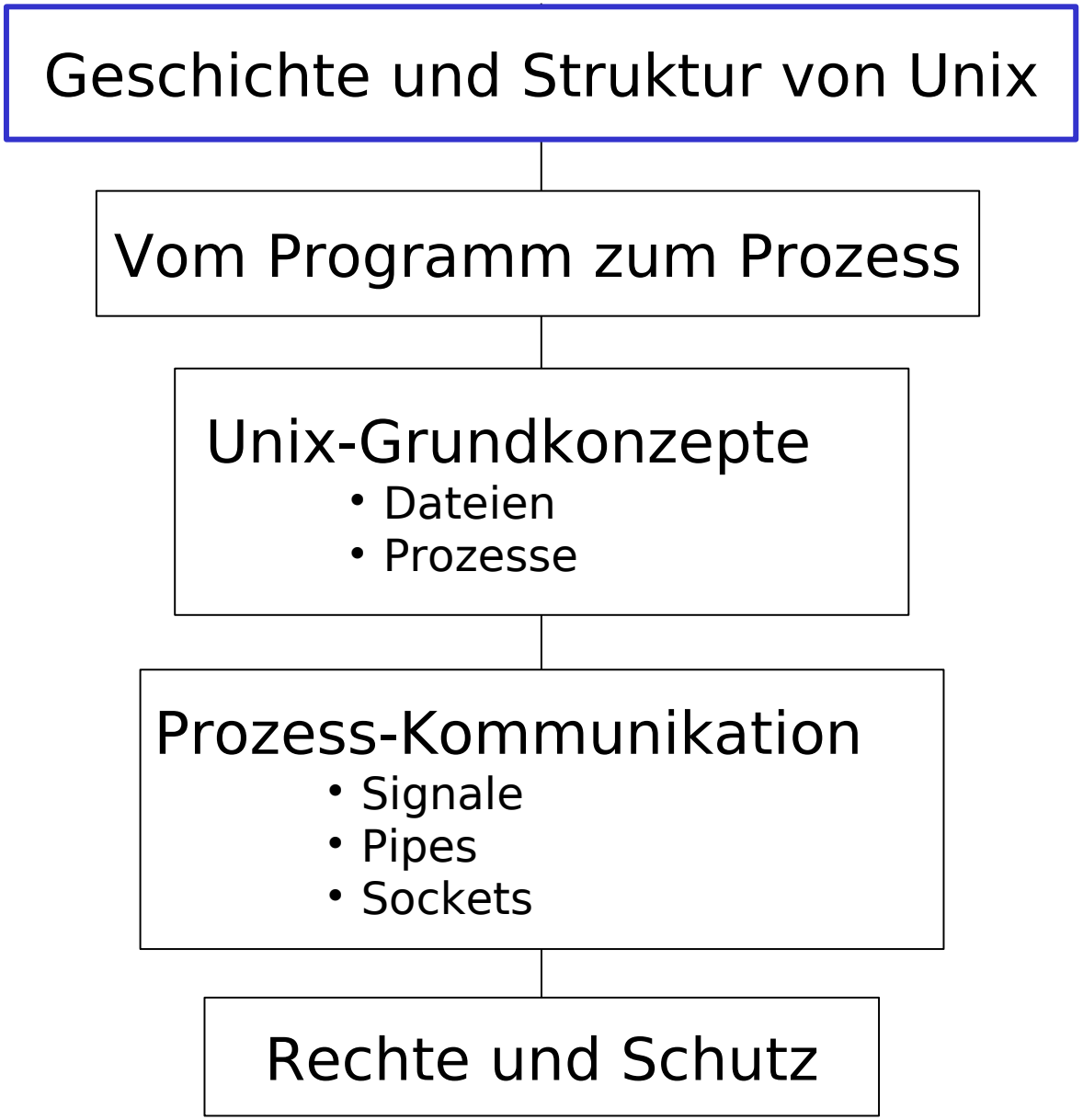
Fallbeispiel Unix

Betriebssysteme

Hermann Härtig
TU Dresden



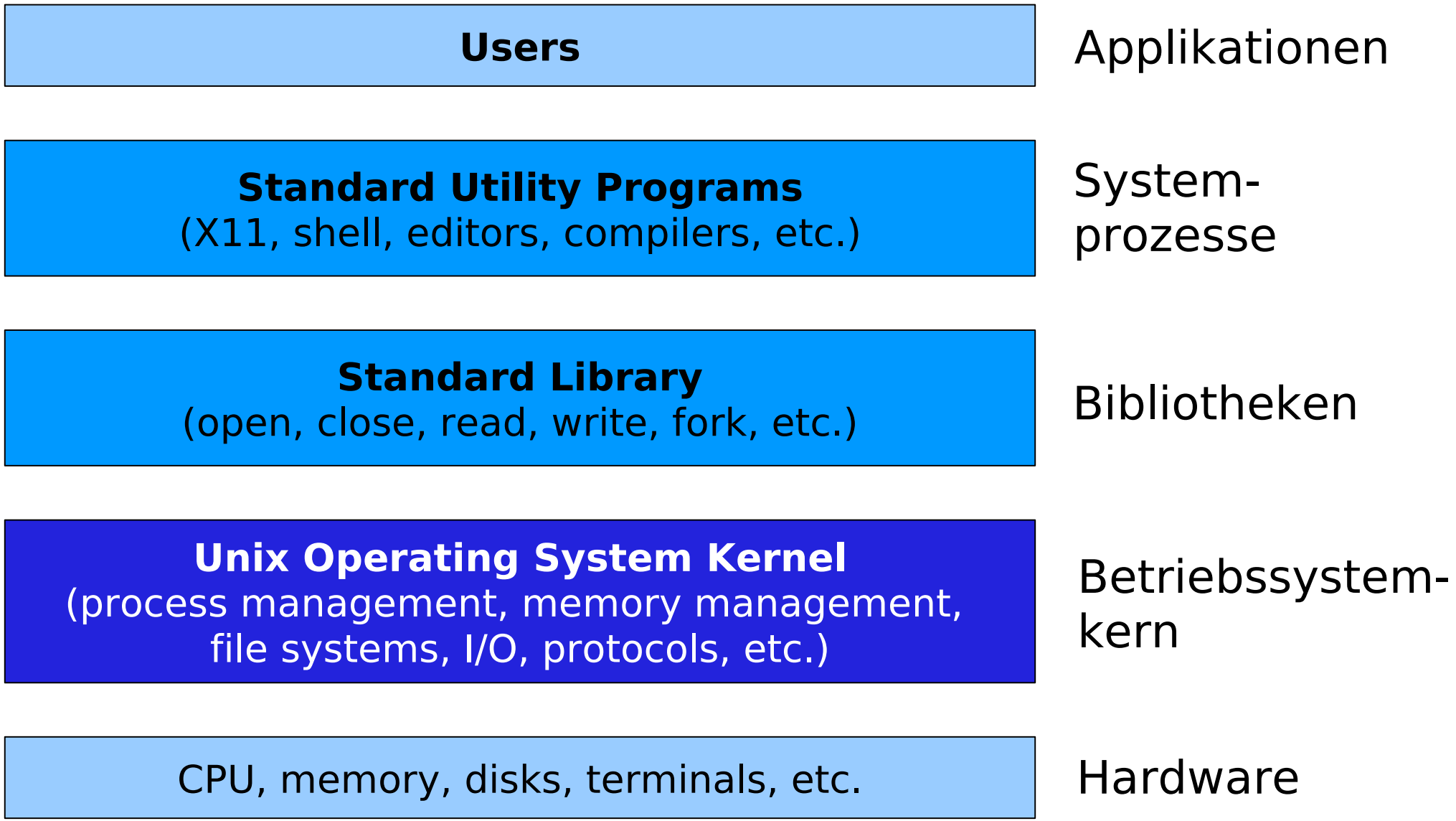
Wegweiser


Geschichte und Struktur von Unix
Vom Programm zum Prozess
Unix-Grundkonzepte
• Dateien
• Prozesse
Prozess-Kommunikation
• Signale
• Pipes
• Sockets
Rechte und Schutz

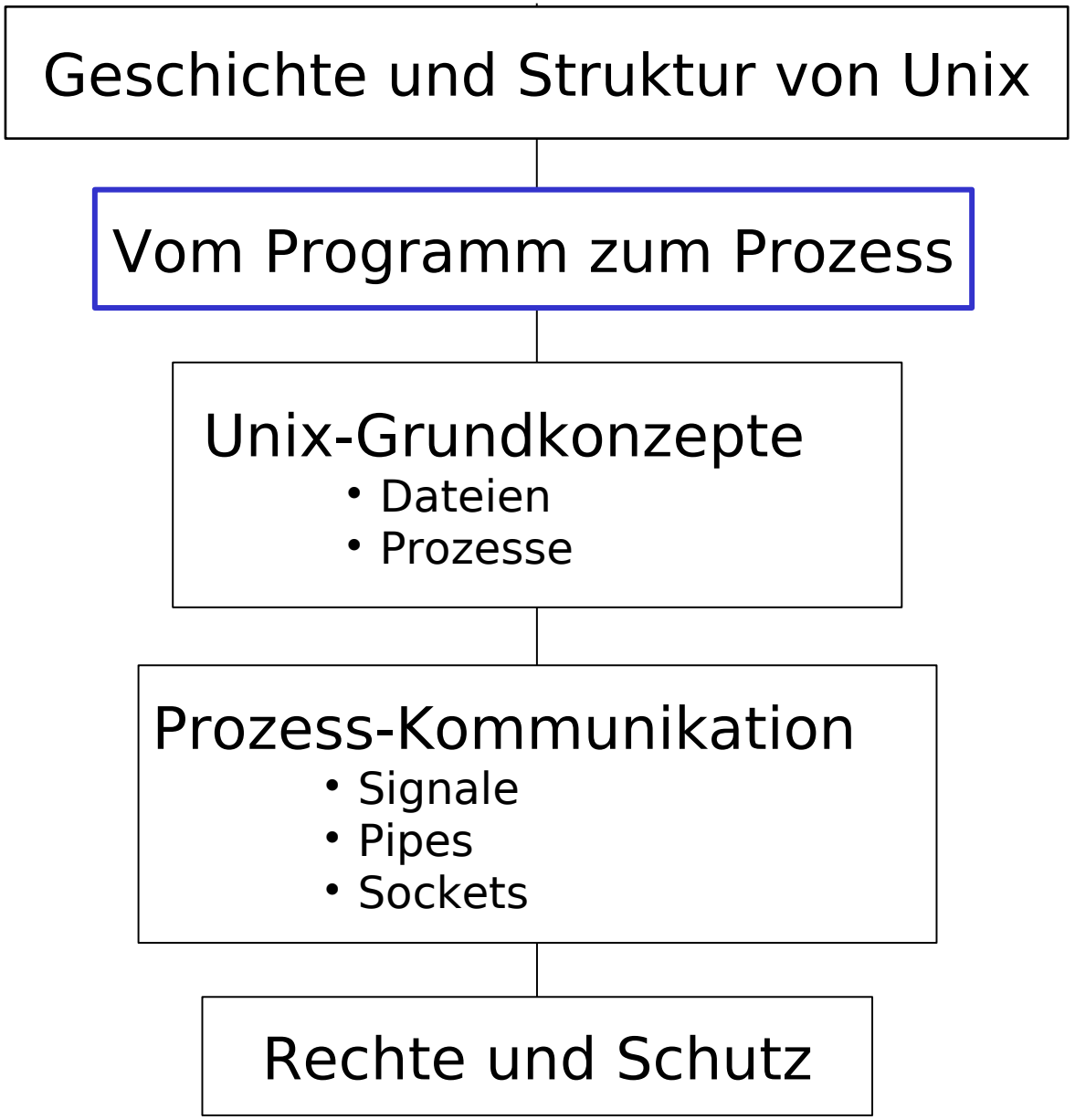
Unix-Story

196x	MULTICS (MIT)	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
1982	Randell et al.	Newcastle Connection
1985	Stallman	GNU / FSF
1986	IEEE	Posix
		“Unix wars“
199x	L. Torvalds et al.	Linux

Grobstruktur Unix



Wegweiser

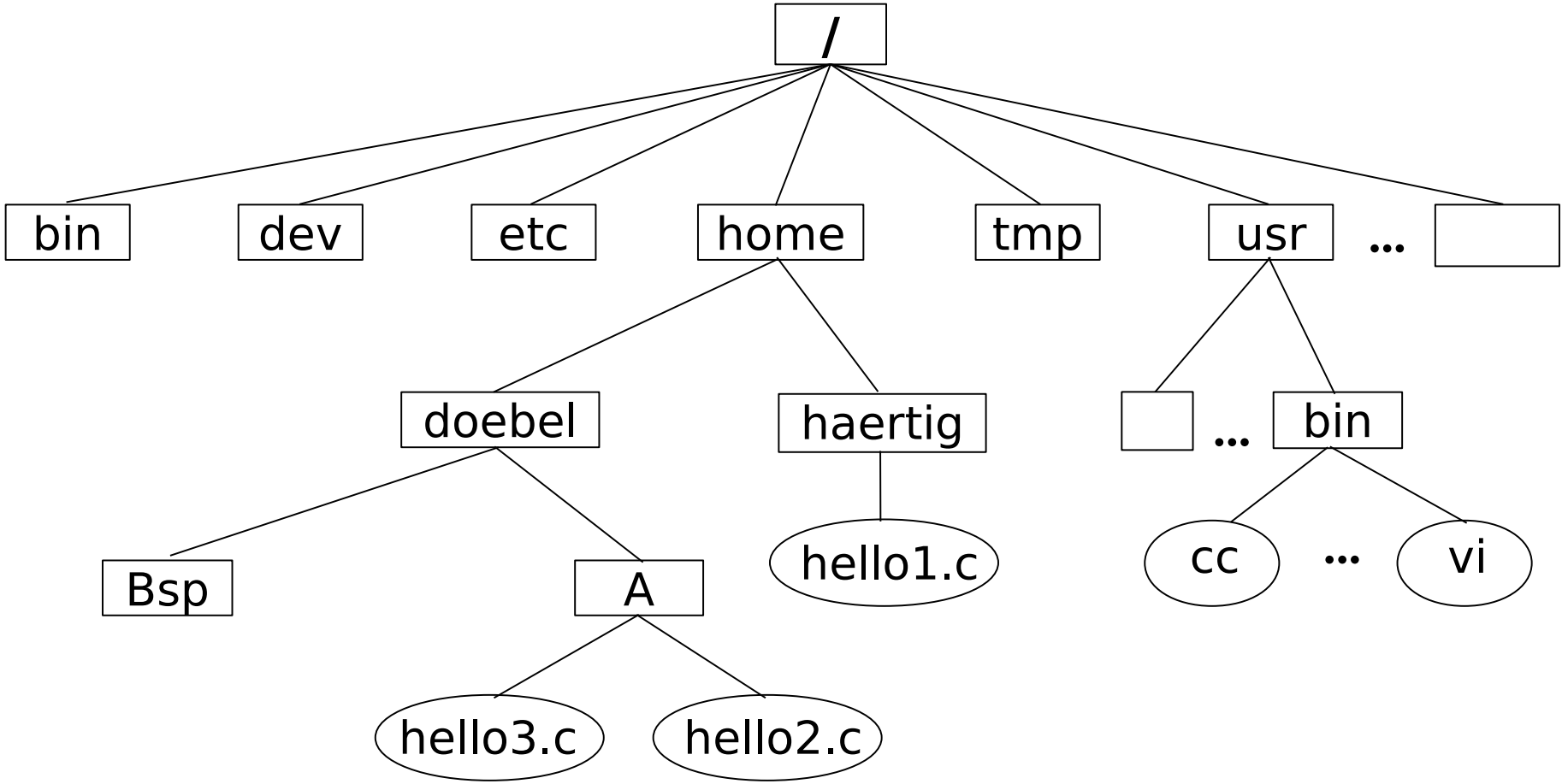

Geschichte und Struktur von Unix
Vom Programm zum Prozess
Unix-Grundkonzepte
• Dateien
• Prozesse
Prozess-Kommunikation
• Signale
• Pipes
• Sockets
Rechte und Schutz

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) files
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 options arguments**

Verzeichnisstruktur



Programmentwicklung

hello1.c

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    printf(„Hello World\n“);
    return 0;
}
```

Präprozessor-Direktive

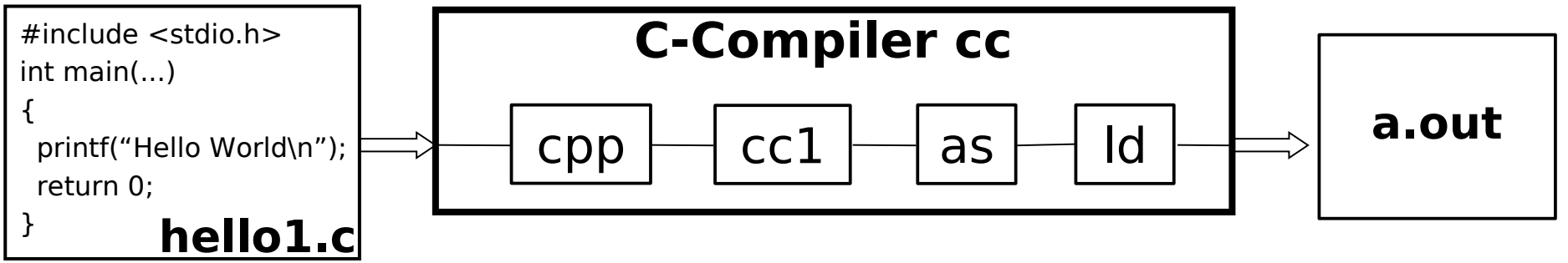
Eintrittspunkt

exit-Status
(0: erfolgreich)

```
mkdir Bsp
cp /home/haertig/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()
{
      // setup
      status = main(argc, argv);
```

```
int main(int argc, char **argv)
{
      printf(„Hello World\n“);
```

```
int printf(...)  → int vprintf(...)
{
      write(stdout, „Hello world\n“, 12);
            EAX = 2
            EBX = 1
            ECX = „Hello World\n“
            EDX = 12

            INT 0x80
}
```

/lib/libc.so.6

```
      return 0;
}
```

a.out

```
      exit(status);
}
```

crt1.o

Schnittstelle zum Betriebssystem

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

Kernel 2.2/2.4/2.6/2.6.28-15: 190 ⇒237 ⇒273 ⇒331 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 ulimitold olduname 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 vserver mbind get_mempolicy set_mempolicy mq_open mq_unlink mq_timedsend mq_timedreceive mq_notify mq_getsetattr kexec_load waitid add_key request_key keyctl ioprio_set ioprio_get inotify_init inotify_add_watch inotify_rm_watch migrate_pages openat mkdirat mknodat fchownat futimesat fstatat64 unlinkat renameat linkat symlinkat readlinkat fchmodat faccessat pselect6 ppoll unshare set_robust_list get_robust_list splice sync_file_range tee vmsplice move_pages getcpu epoll_pwait utimensat signalfd timerfd_create eventfd fallocate timerfd_settime timerfd_gettime signalfd4 eventfd2 epoll_create1 dup3 pipe2 inotify_init1

Programmentwicklung

hello2.c

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

cc hello2.c
hello2

Systemaufrufe unter Linux

- Parameter werden in Registern übergeben
  eax = Nummer des Systemaufrufs (0 – ...)
- 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 0x80“
- 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

Prozess-Struktur

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;
}
```

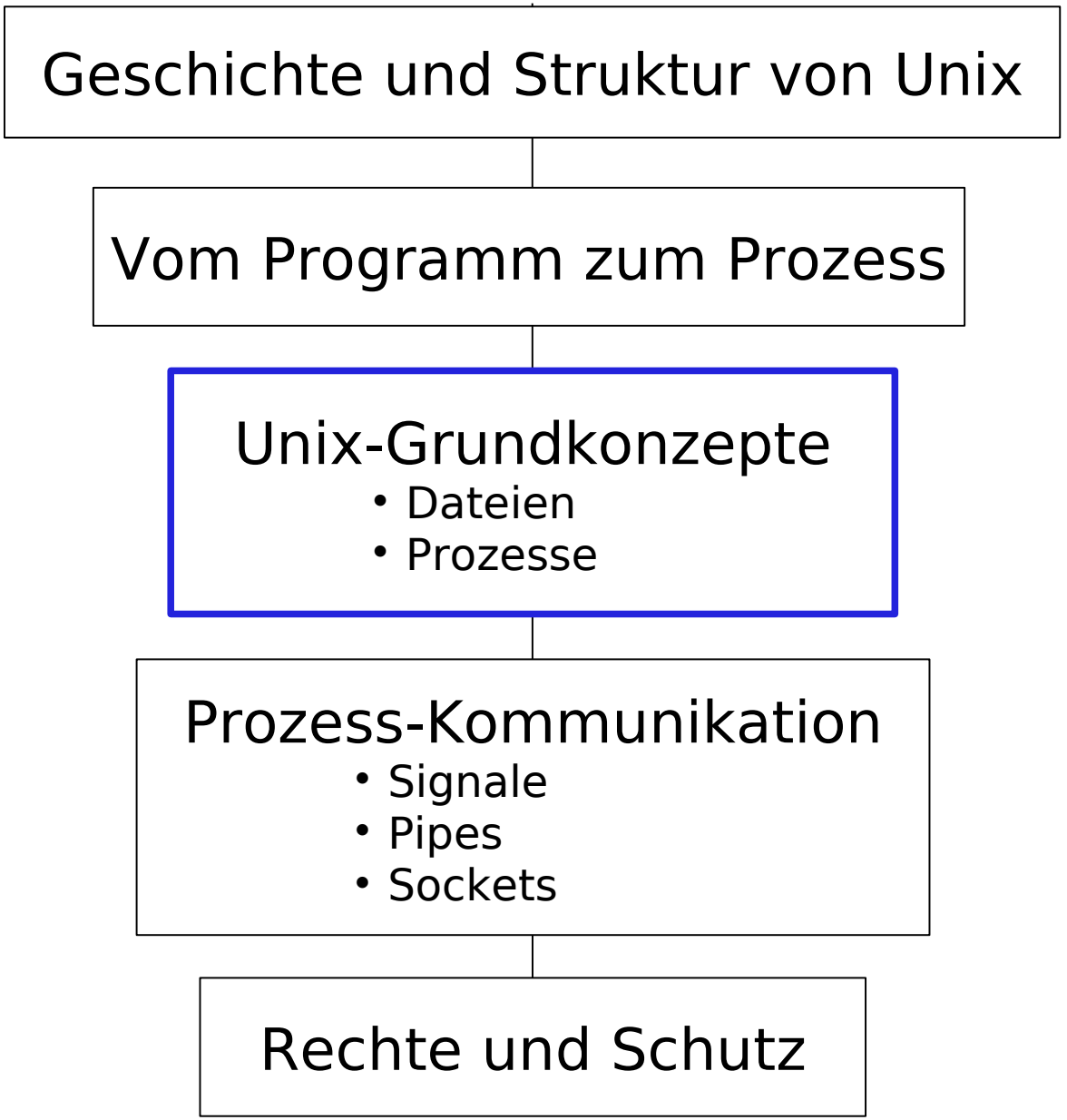
```
hello3 &
ps -l
```

Prozess-Struktur

```
>$ hello3 &
[1] 433
>$ Bitte Enter-Taste drücken...
ps -l
```

UID	PID	PPID	PRI	CMD
1026	331	330	75	bash
1026	433	331	76	hello3
1026	453	331	77	ps

Wegweiser


Geschichte und Struktur von Unix
Vom Programm zum Prozess
Unix-Grundkonzepte
• Dateien
• Prozesse
Prozess-Kommunikation
• Signale
• Pipes
• Sockets
Rechte und Schutz

Shell-Ebene: Dateien

Pfadnamen

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

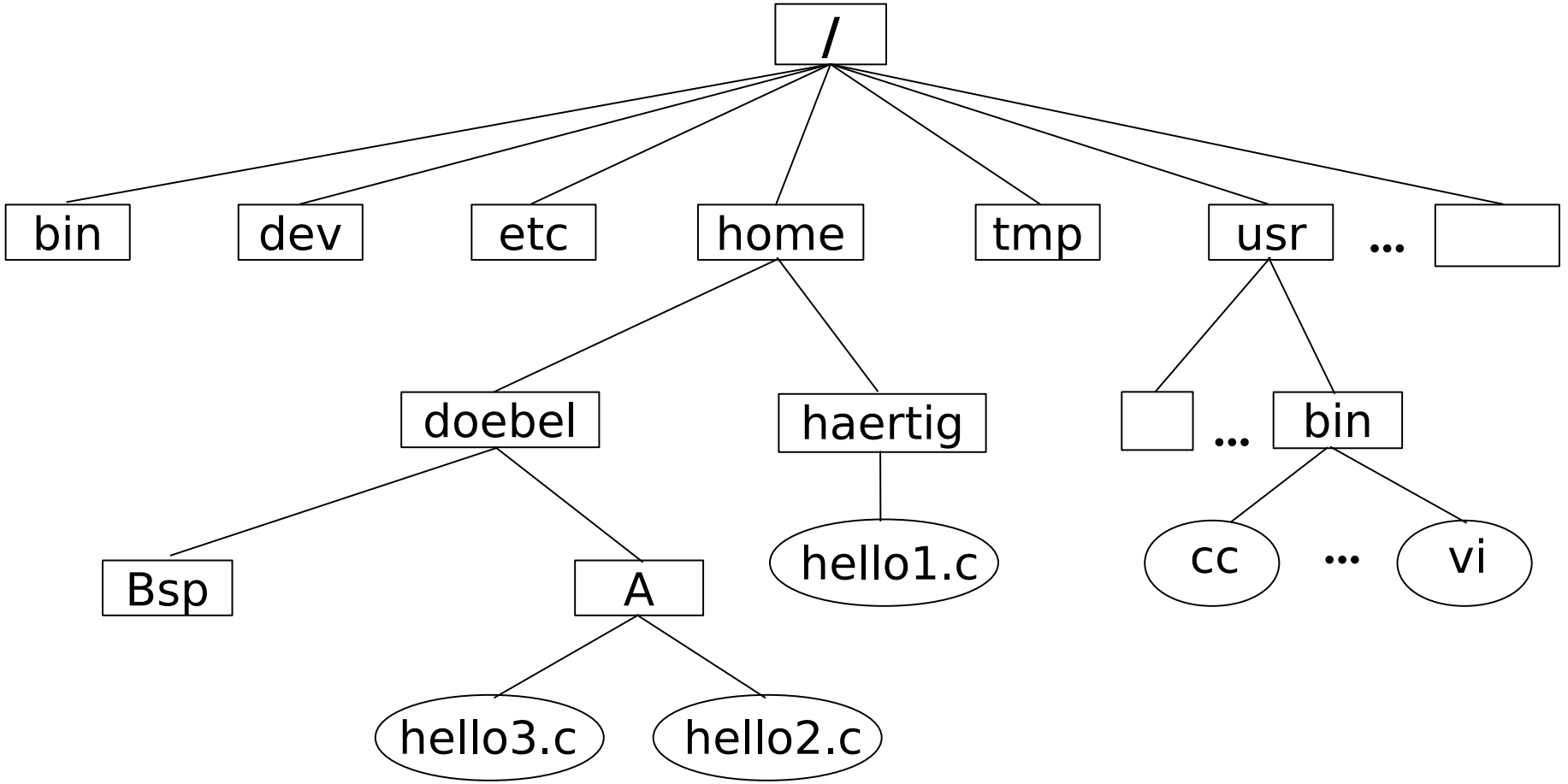
- Pfadnamen häufig als Parameter: `cp src dest`
- wichtig: viele Konventionen bzgl. Dateinamen (`/bin`, `/etc`, ...)

`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`

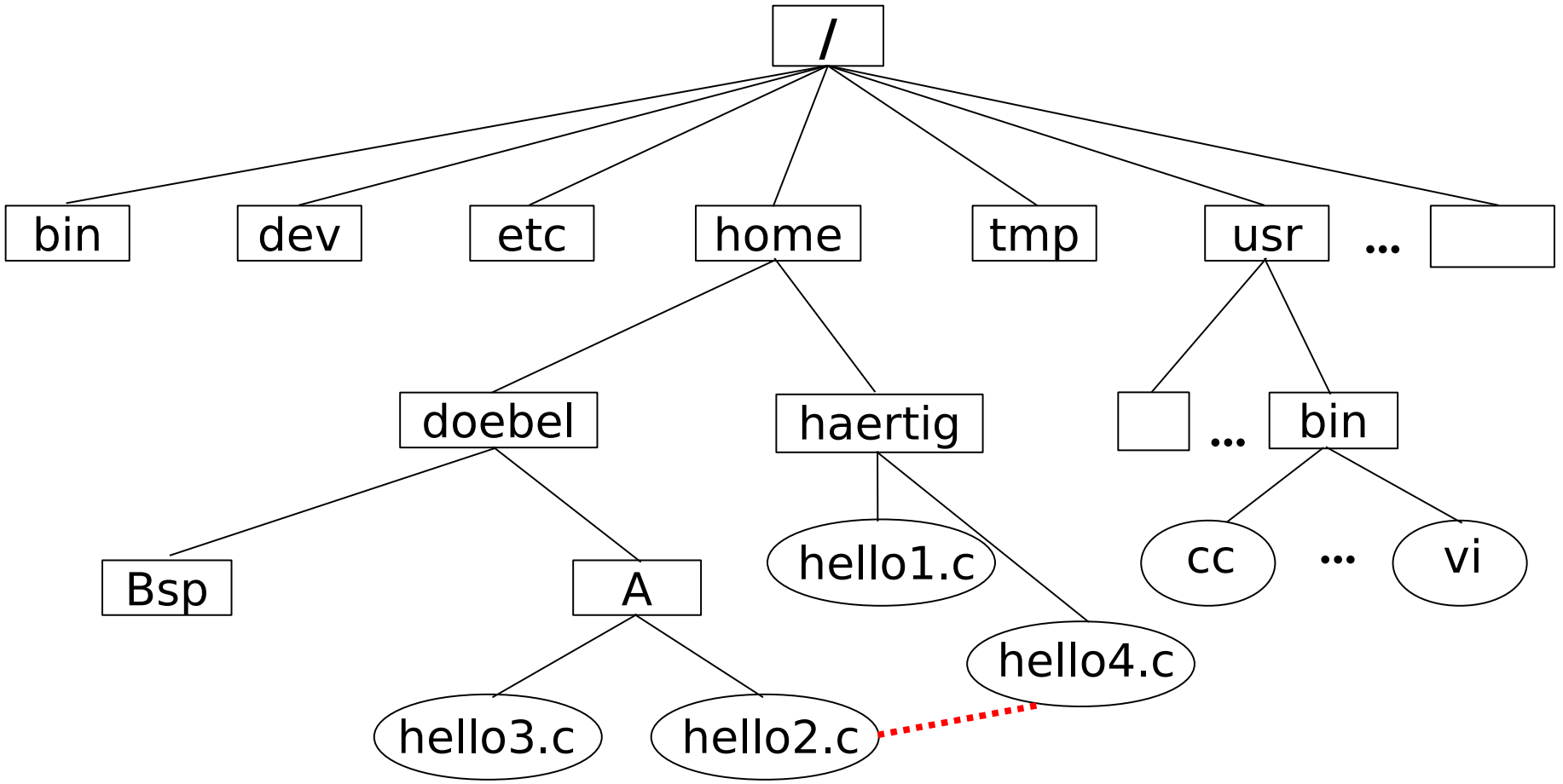
`rm /etc/xy`
keine Name mehr für Datei vorhanden → gelöscht

Verzeichnisstruktur



`ln hello2.c /home/haertig/hello4.c`

Verzeichnisstruktur



`ln hello2.c /home/haertig/hello4.c`

Dateien: Kernaufrufe (System-Calls)

`fd = open(name, flags, mode)`

- offen, liefert „Datei-Descriptor“ (0,1,2 konfigurierte Werte)
- Flags: O_RDONLY, O_RDWR, ...
- mit Flag O_CREAT wird Datei erzeugt wenn nicht da
- `mode` bestimmt Zugriffsrecht der Datei bei Erzeugung
- Datei-Deskriptor Konvention:
  - 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)  write`

`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

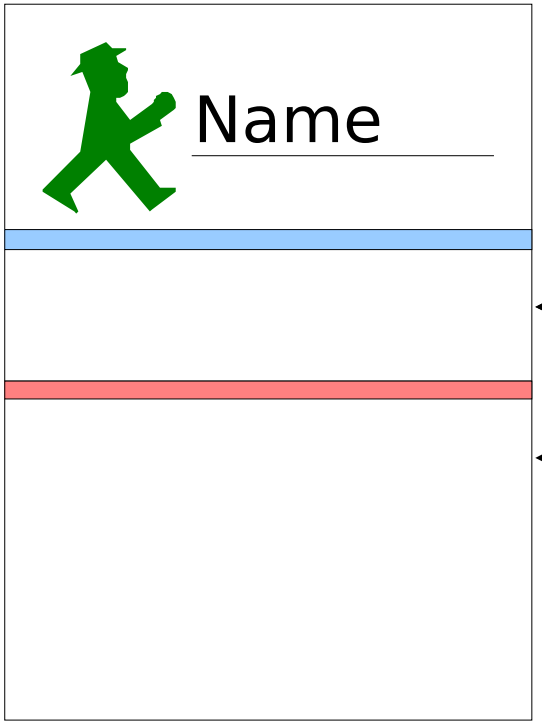
Unix-Prozess

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

Viele Prozesse pro Station

- Benutzerprozesse
- Hintergrund-Systemprozesse („daemons“)

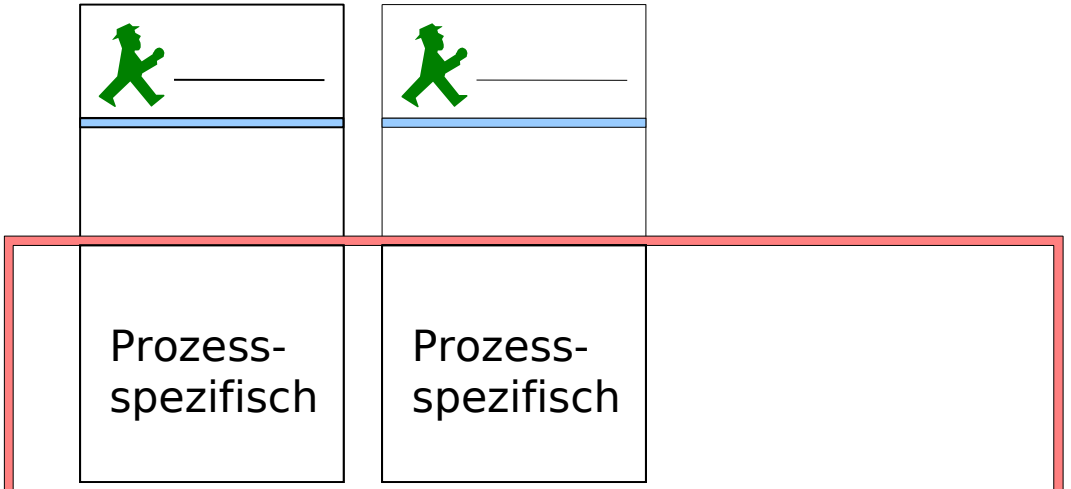
Darstellung Unix-Prozesse im Folgenden



User-Mode Daten

dem Prozess zugeordnete Betriebsmittel, werden nur im Kern manipuliert

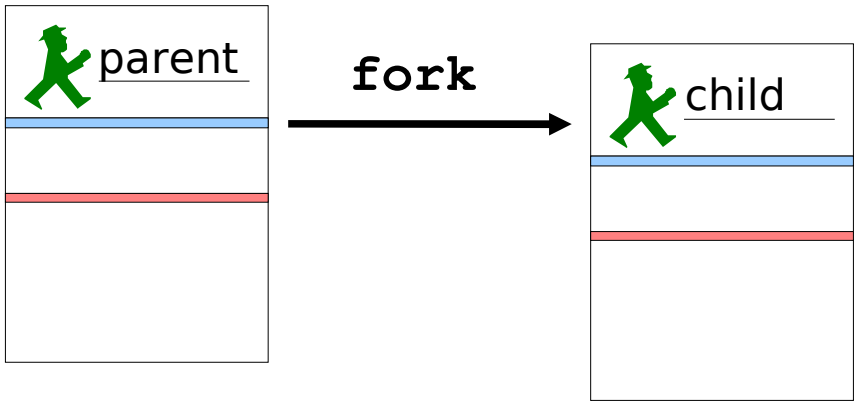
Kern-Adressraum



- 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 Dateideskriptoren ...
```




```
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)`

- existiert noch (als „Zombie“), bis Eltern-Prozess `wait` ausführt
- 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)
// Kind erhält fd des Eltern-Prozesses
// beide Prozesse setzen 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)
// Kind erhält fd des Eltern-Prozesses
// beide Prozesse setzen 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)
// Kind erhält fd des Eltern-Prozesses
// beide Prozesse setzen 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);
}
```

Kernaufrufe (System-Calls)

Beispiel

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

- Ergebnis: Anzahl der gelesenen Bytes
- Konvention: -1 $\rightarrow$ Fehler
- Meldung der Fehlerursache: **`errno`**

Kernaufruf im Detail

Benutzerprozess

```
read(...) {

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

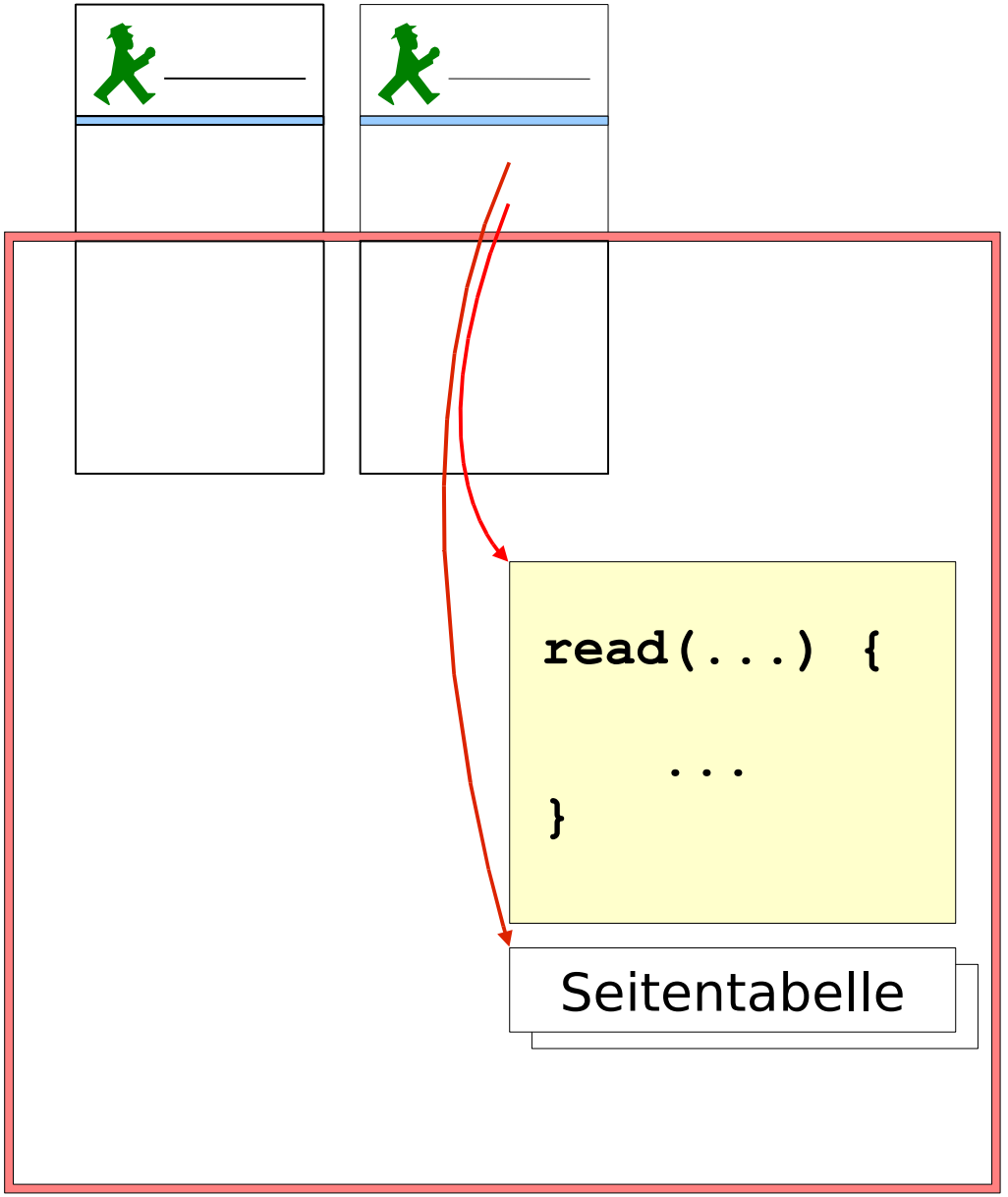
```
  //weiter geht's
  ...
}
```

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

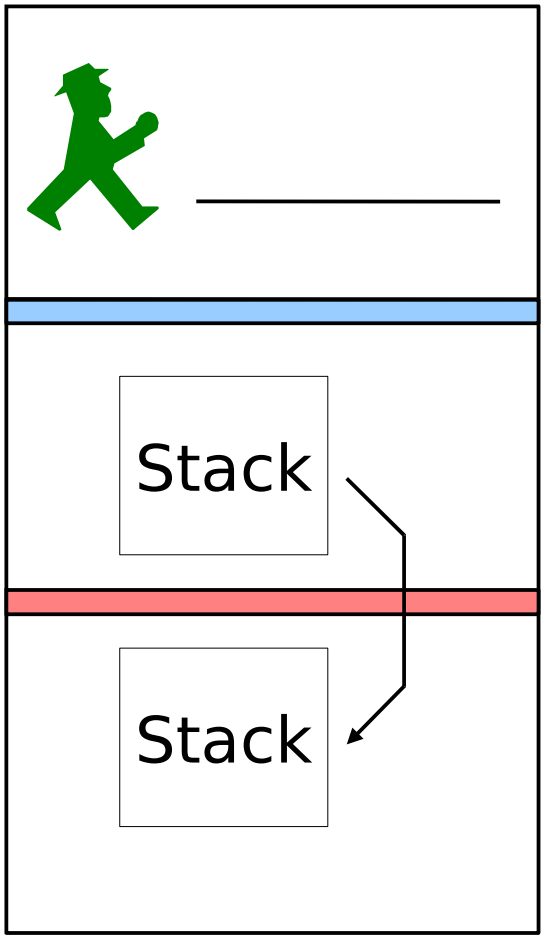
Schutz des Kerns?



- Wie wird der Kern sicher aufgerufen?
- Wie wird der Kern-Adressraum geschützt?
  Schutz z. B. der Dateideskriptoren
- Bei multithreaded Prozessen (nicht klassisches Unix):

  Wie wird verhindert, dass ein Thread den Keller des anderen manipuliert, während er gerade im Kern ist?

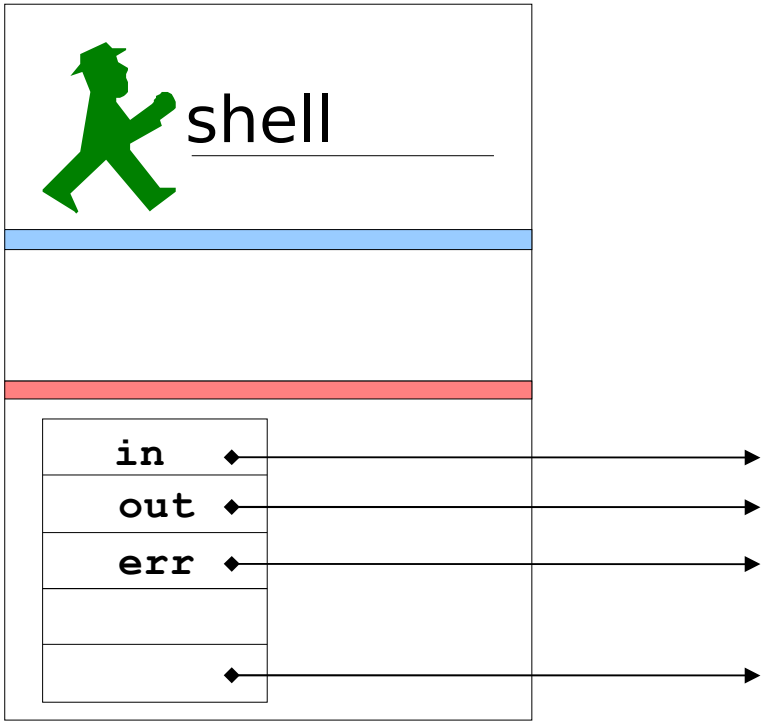
Zwei Keller pro Prozess: User, Kernel



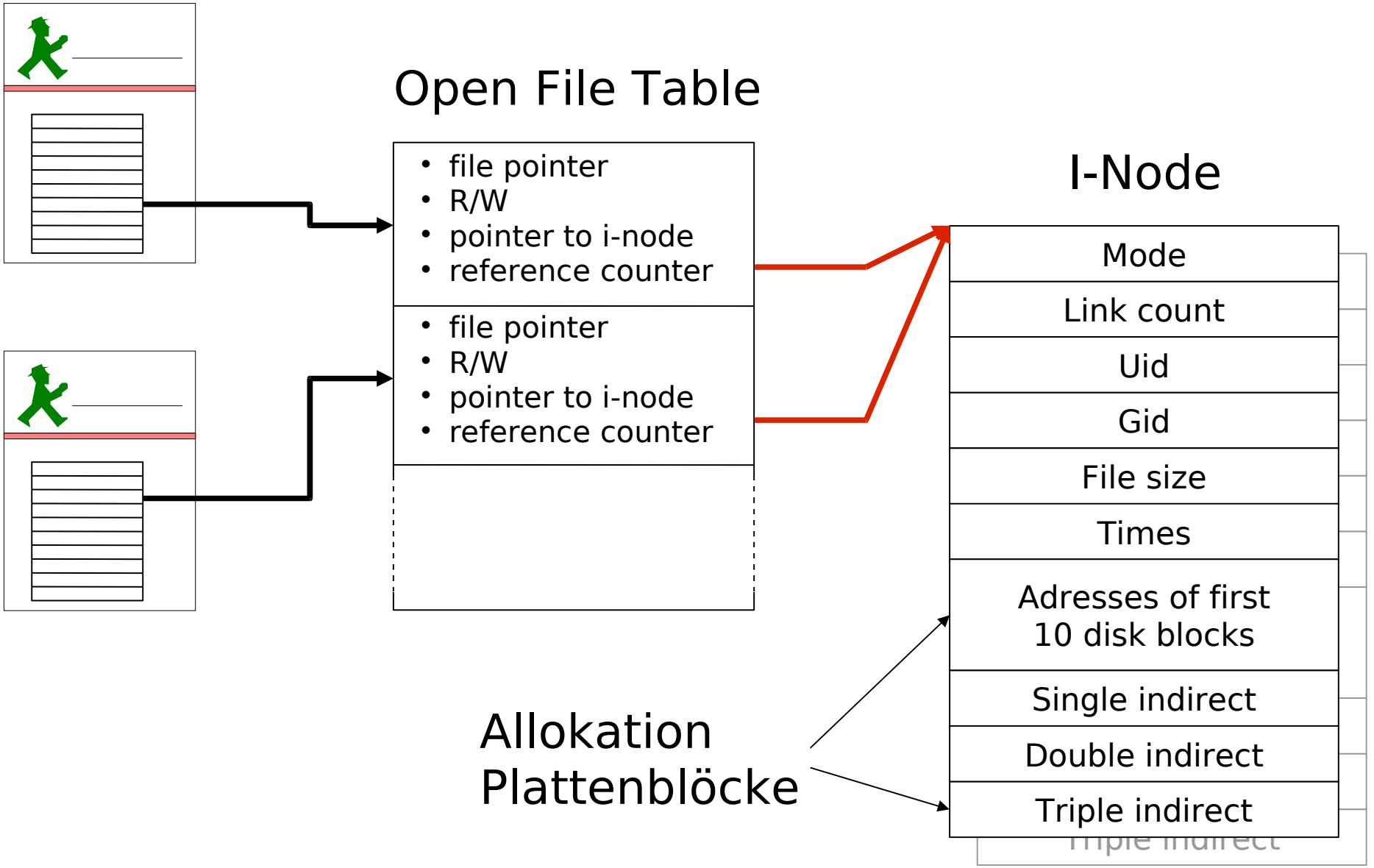
Bei Systemcalls wird

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

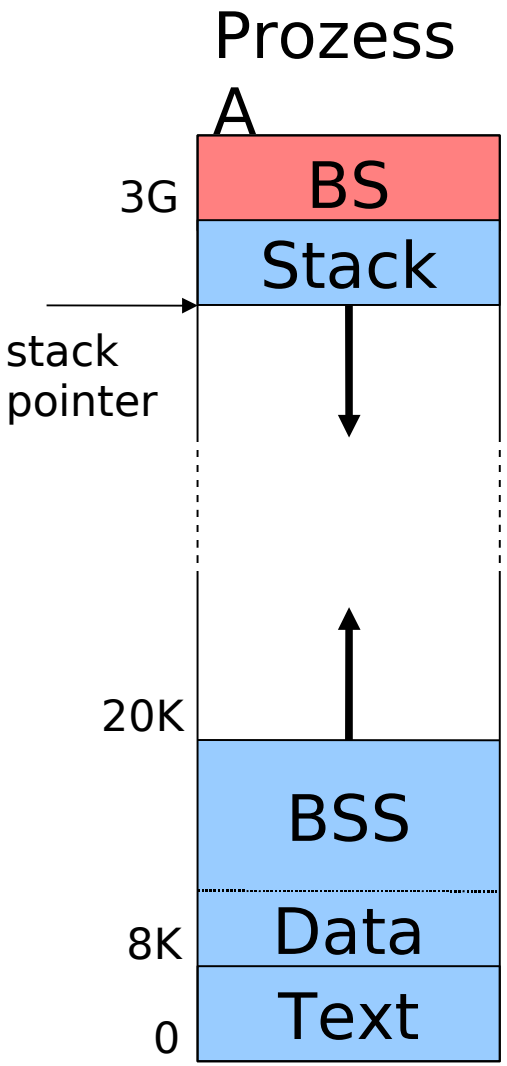
Kernschnittstelle: Datei-Deskriptoren



Kerninterne Datenstrukturen



Das Unix-Speichermodell



Daten-Segment

- globale Daten eines Programms
  - **Data**: initialisierte Daten
  - **BSS**: per Konvention mit 0 initialisiert
    erweiterbar durch Systemaufruf

    `brk`

Textsegment

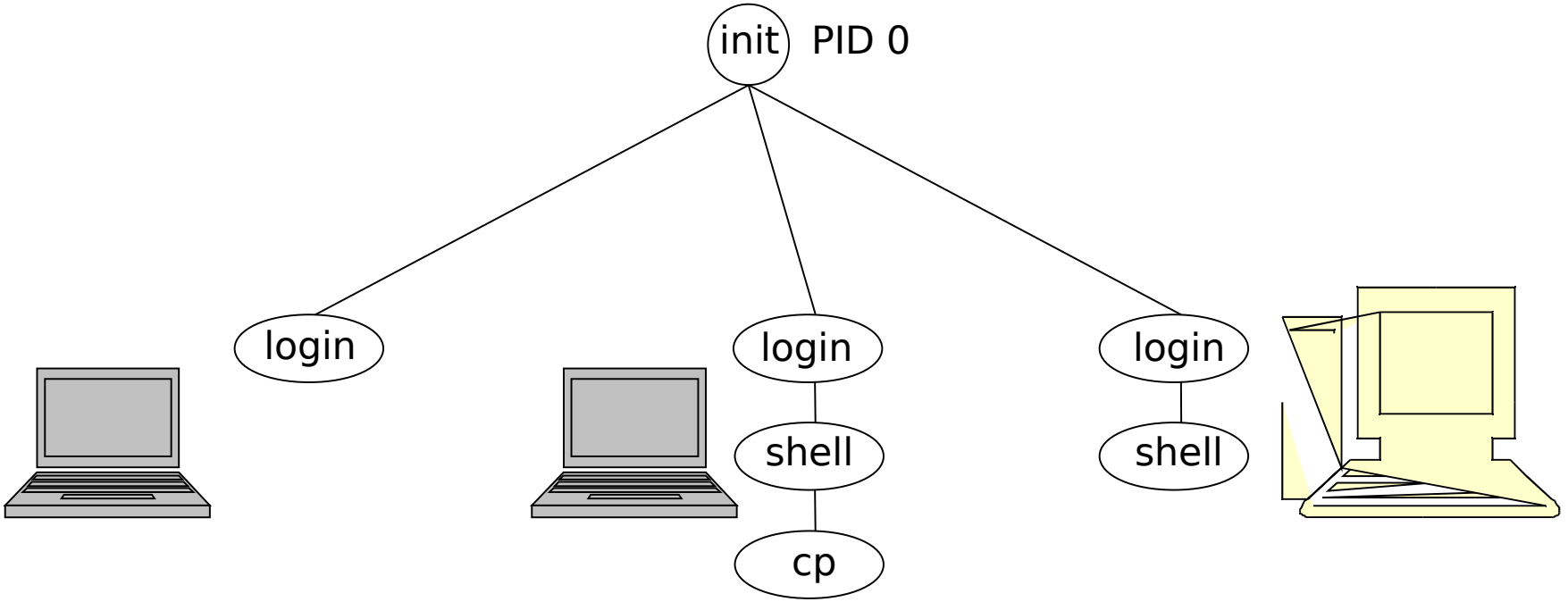
- enthält Maschinencode
- read only
- erste Seite frei zum Entdecken nicht-initialisierter Pointer
- "shared text"

Keller-Segment

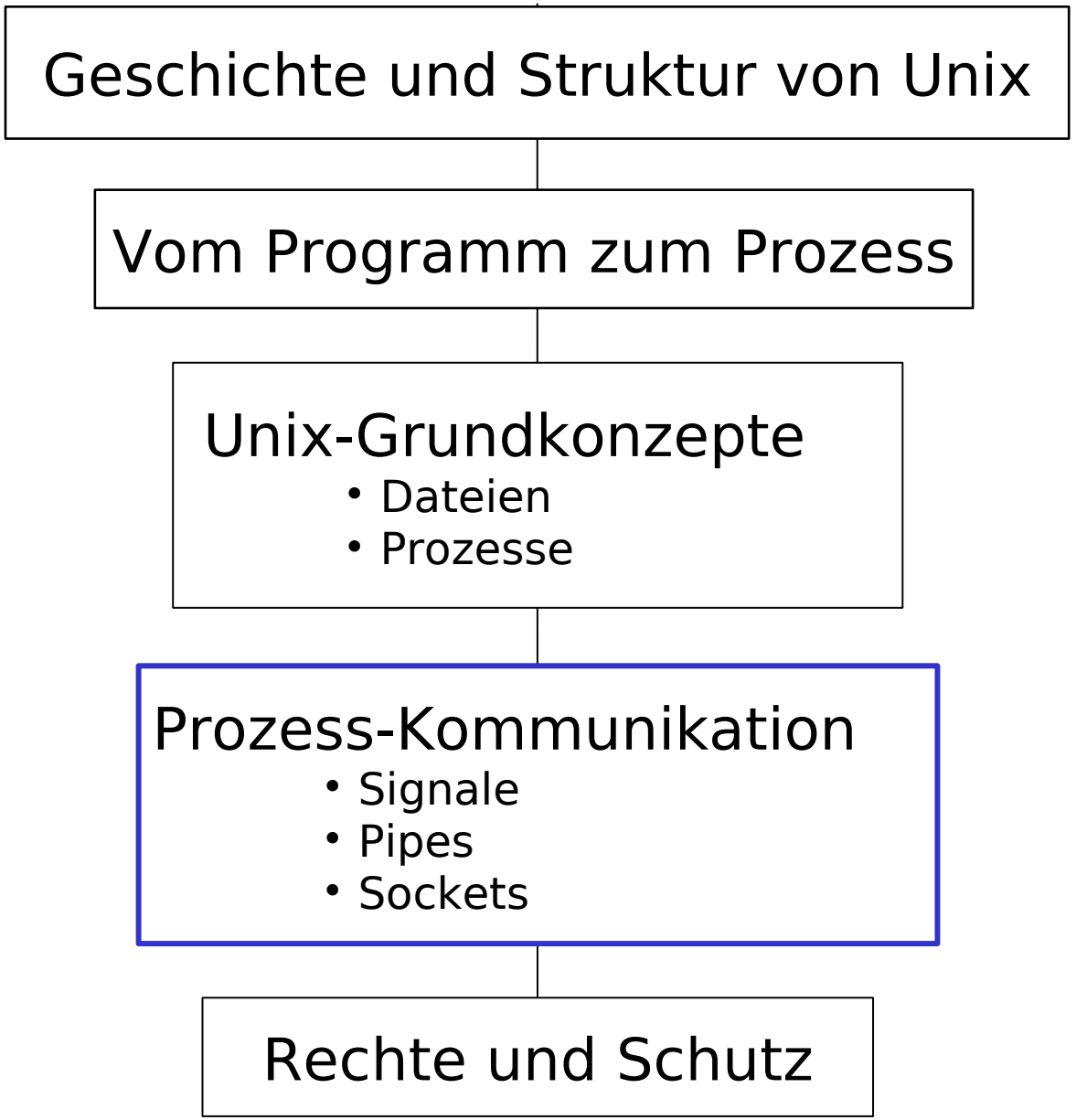
- Keller (Stack)
- enthält Parameter und Kontext (environment)

Systemstart und Login

- **`init`** – der erste Prozess



Wegweiser


Geschichte und Struktur von Unix
Vom Programm zum Prozess
Unix-Grundkonzepte
• Dateien
• Prozesse
Prozess-Kommunikation
• Signale
• Pipes
• Sockets
Rechte und Schutz

Prozesskommunikation: Signal, Pipe und Socket

Signale

- Senden von Signalen: z. B.
  - kill-Kernaufruf (`kill(pid, SigNo)`)
  - Terminaltreiber

- Disponieren:
  - gar nichts: Default-Verhalten, z. B. Abbruch
  - ignorieren: Signal verpufft
  - blockieren: Signal wird später zugestellt (nach unblock)
  - zustellen: Signalhandler wird aufgerufen

Signale

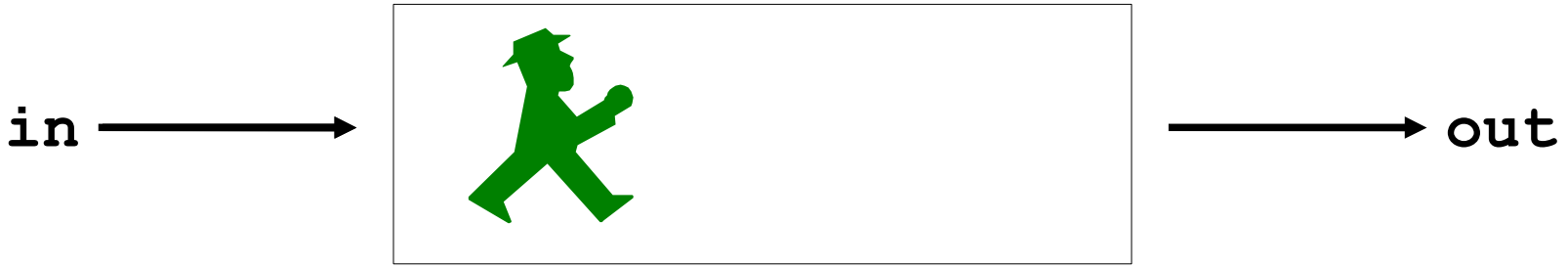
Signal	Cause
SIGABRT	Sent to abort process and force a core dump
SIGALARM	The alarm clock has gone off
SIGFPE	A floating point error has occurred (e.g., division by 0)
SIGHUP	The phone line the process was using has been hung up
SIGILL	The process has executed an illegal machine instruction
SIGINT	The user has hit the DEL key to interrupt the process
SIGOUIT	The user has hit the ley requesting a core dump
SIGKILL	Sent to kill a process (cannot be caught or ignored)
SIGPIPE	The process has written on a pipe with no readers
SIGSEGV	The process has referenced an invalid memory address
SIGTERM	Used to request that a process terminate gracefully
SIGUSR1	Available for application-defined purposes
SIGUSR2	Available for application-defined purposes

Pipes und Filterketten

- Programme lesen von STDIN und schreiben nach STDOUT
- Kein Unterschied, ob lesen/schreiben von/in Datei oder über pipe zu einem anderen Prozess.

```
cat a > b

cat < a  > b

cat a | lpr

cat a | sort | lpr
```

Shell-Ebene: Prozesse als Filter



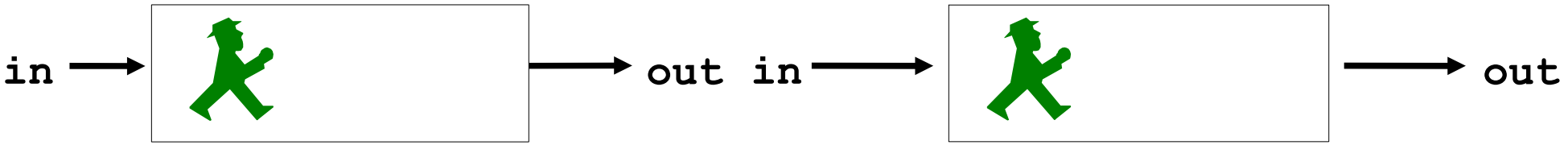
`cat < src.txt | sort`

`sort <in >out`

- Datenstrom als durchgängiges Konzept,
  spezielle Dateien per Konvention (`stdin`, `stdout`)
- Normalfall:
  - Tastatur als "standard in"
  - Terminal als "standard out"

➔ Ein/Ausgabe als Spezialfall von Dateien

Shell-Ebene: Prozesse als Filter




```
cat < src.txt | sort
```

```
sort <mylist.txt | lpr
```

```
cat | sort | lpr
```

“Pipes” als spezielle Datenstrom-Dateien

➔ Datenstrom als durchgängiges Konzept !

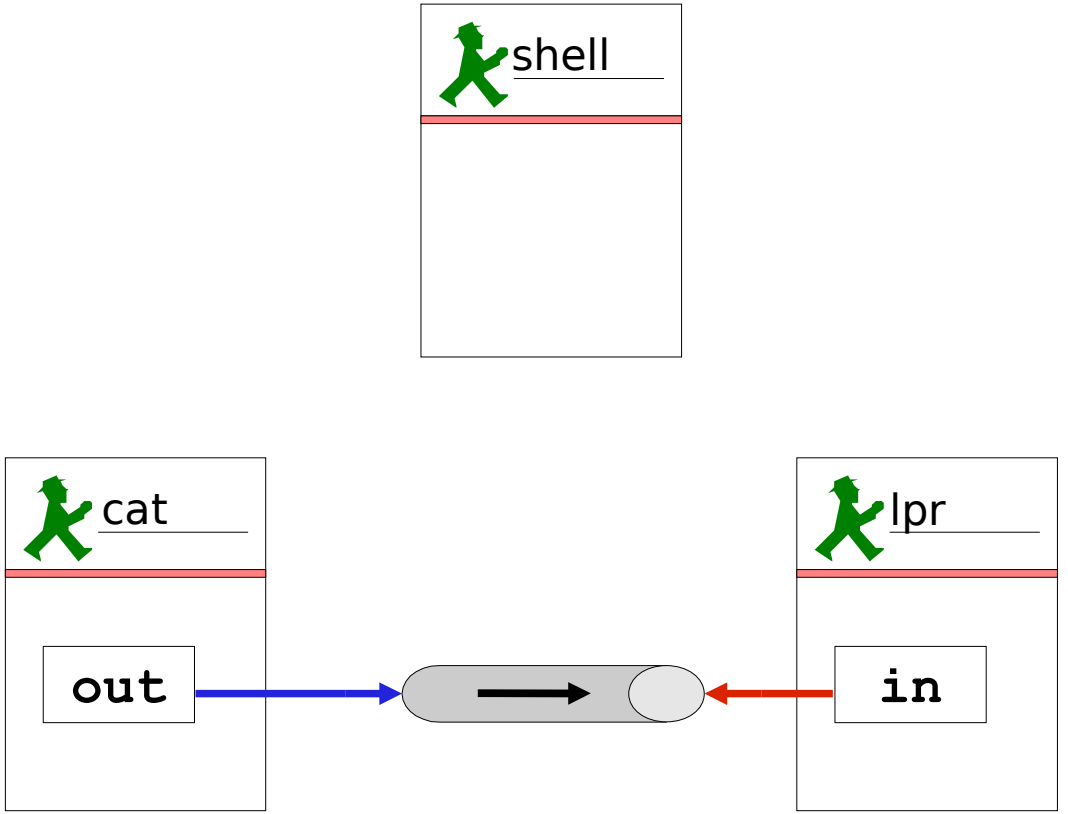
Beispiel: cat <in >out

```
read (command, params);

pid = fork();

  if (pid < 0) {
     // Fehlerbehandlung
  } else if (pid != 0) {

      waitpid(pid, &status, 0); // warte auf Kind-Prozess
  } else {

      close(0); open(in, ...);  // ersetze vorh. fd
      close(1); open(out, ...); // durch in/out

     exec(command, params, env);
  }
```

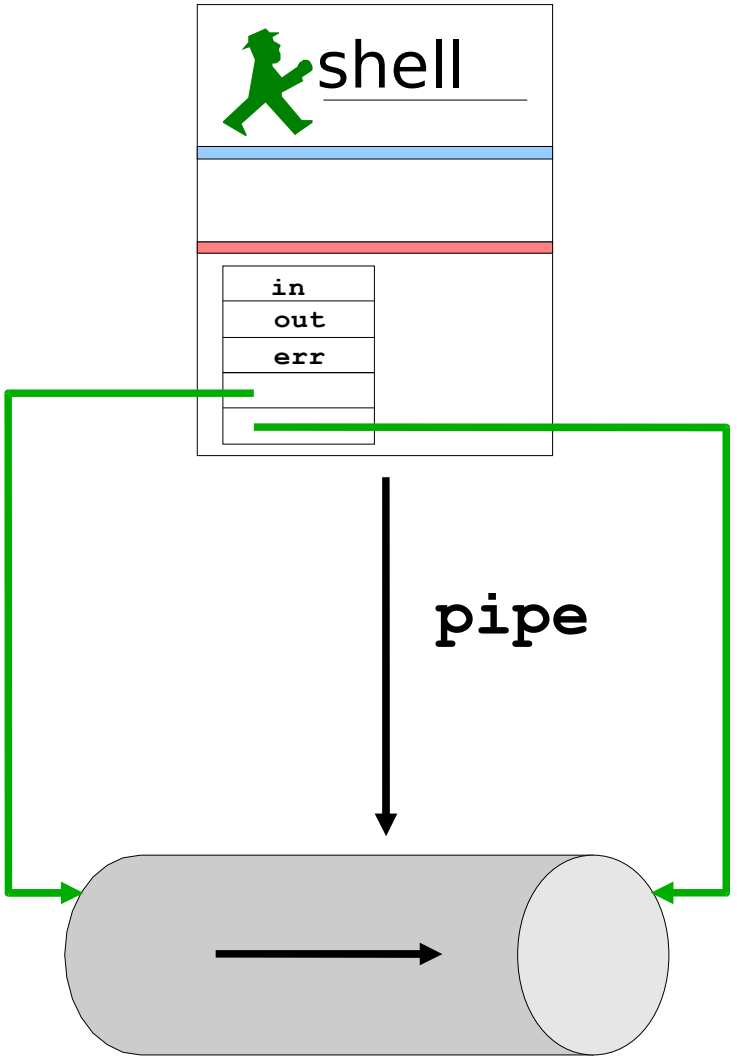
Pipes und Filterketten

z. B. **`cat | lpr`**

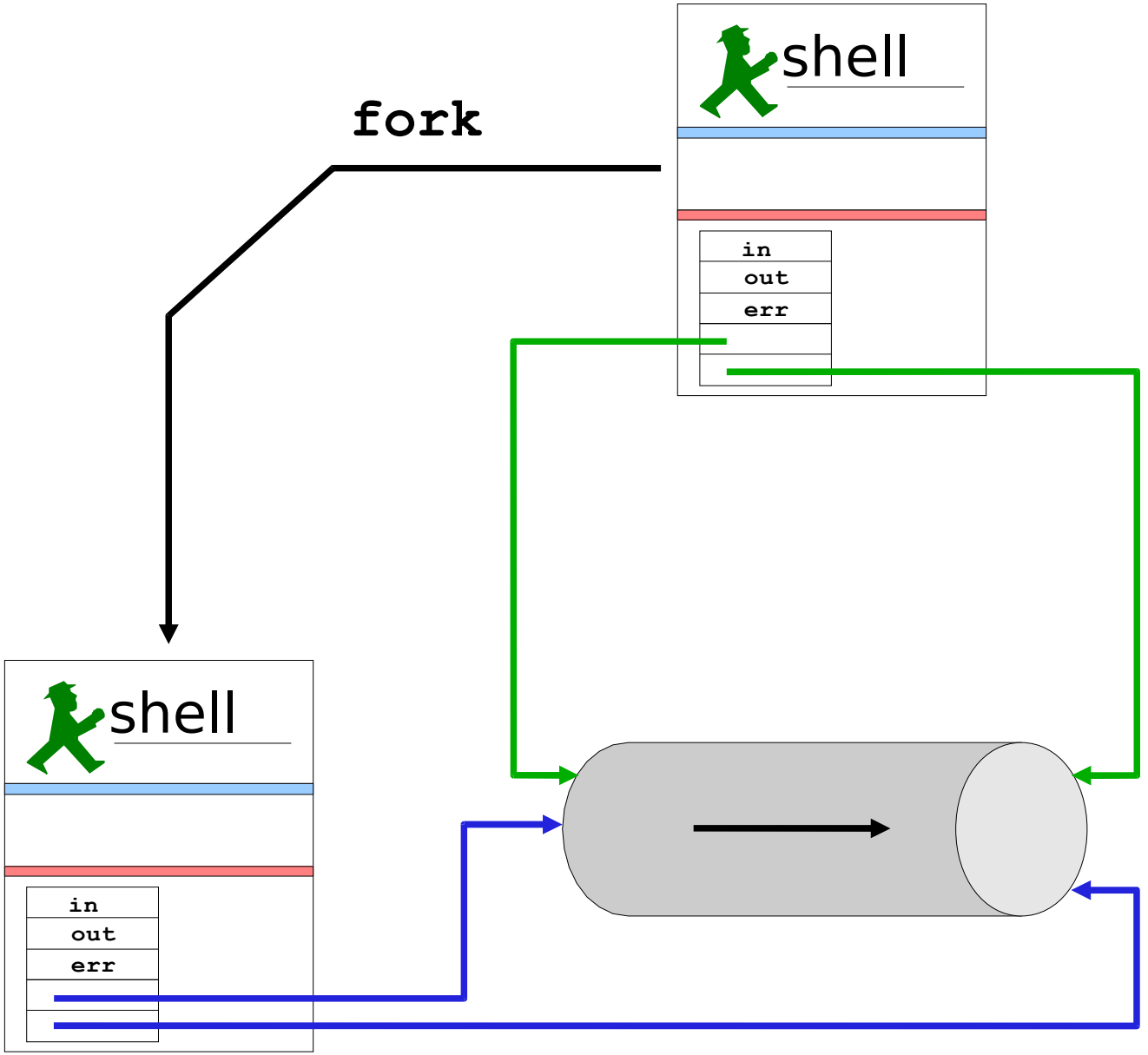
- **`pipe`**
  erzeugt pipe mit 2 fd (fd1, fd2)
- **`fork`**: Kind1 für **`cat`**
- **`fork`**: Kind2 für **`lpr`**
- Elternteil (shell):
  schliesst fd1, fd2
- Kind1:
  schließt fd2
  schließt stdout
  fd1 → stdout
  schließt fd1
  `exec cat`
- Kind2: spiegelbildlich



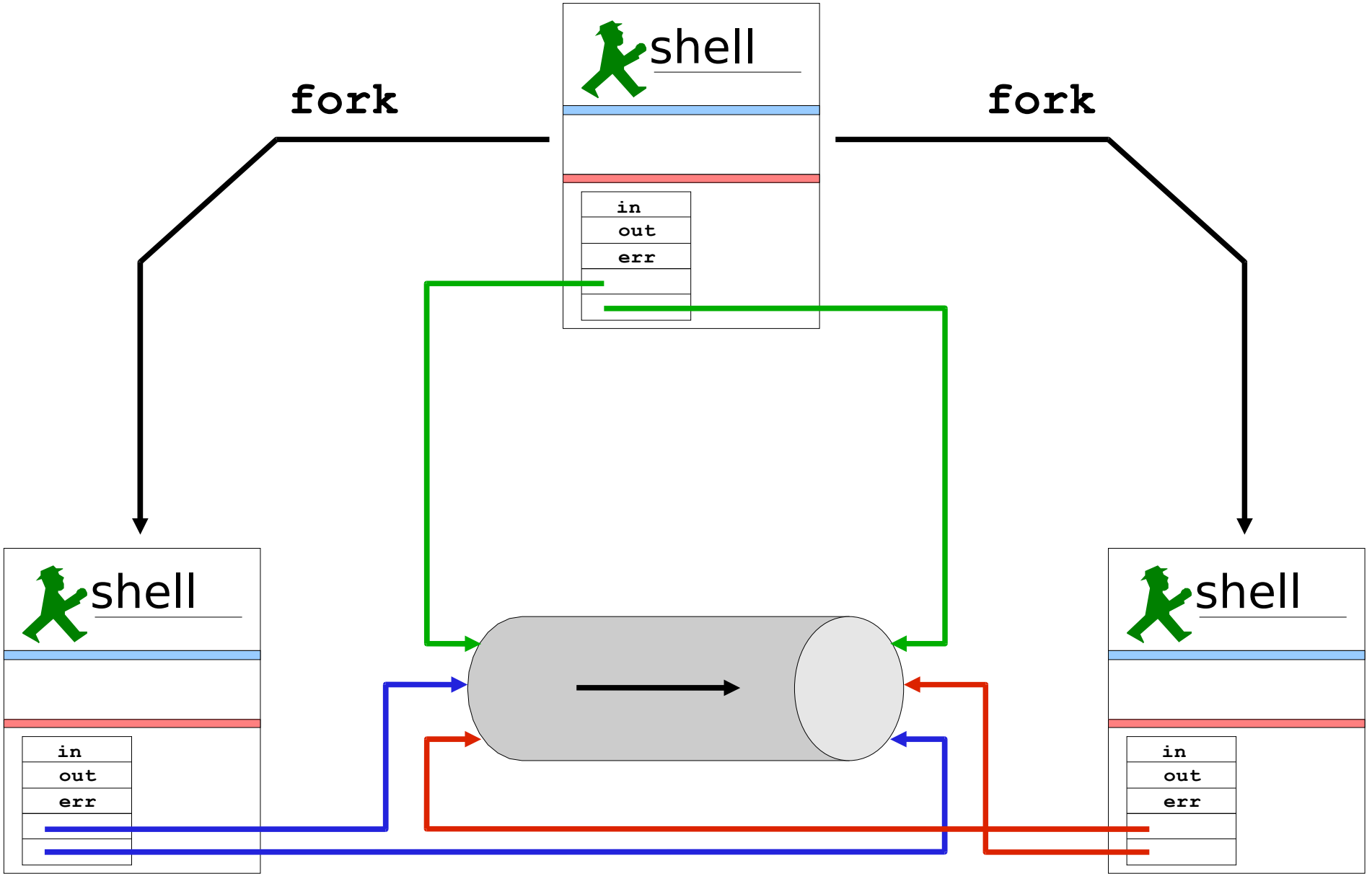
Aufbau einer Filterkette mit Pipes



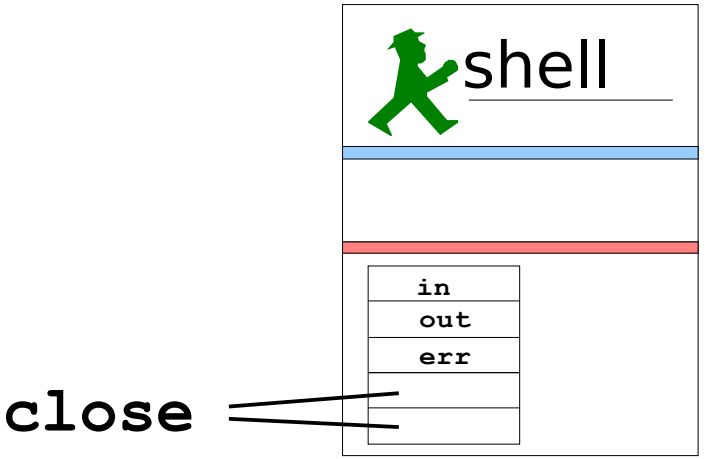
Aufbau einer Filterkette mit Pipes

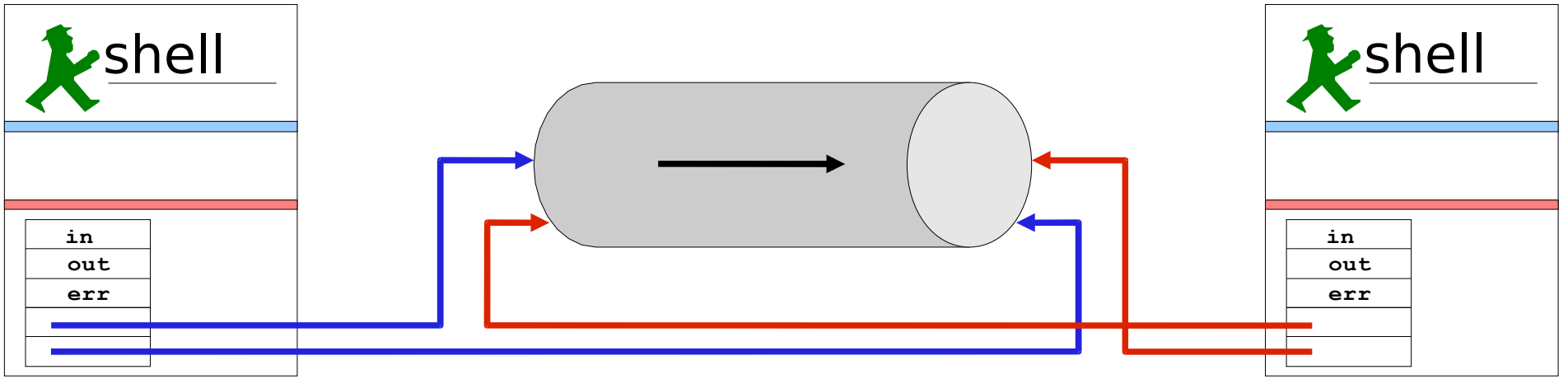


Aufbau einer Filterkette mit Pipes

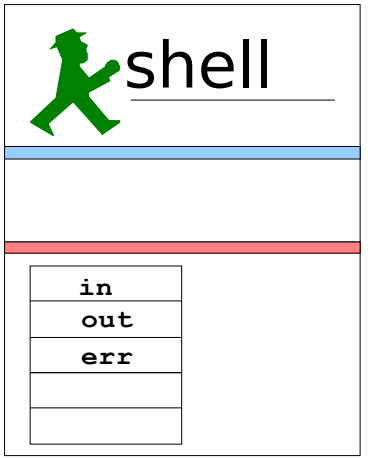


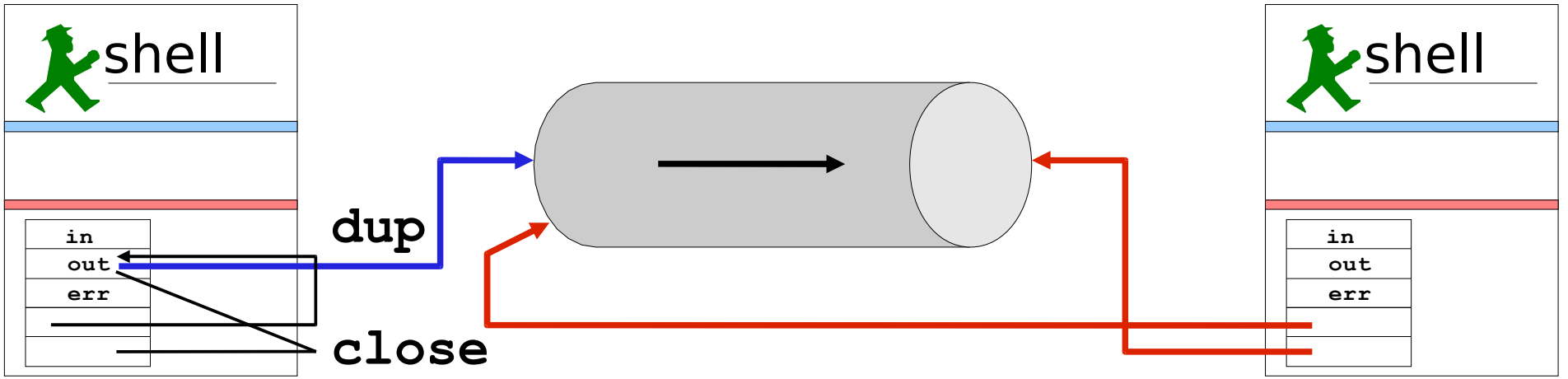
Aufbau einer Filterkette mit Pipes



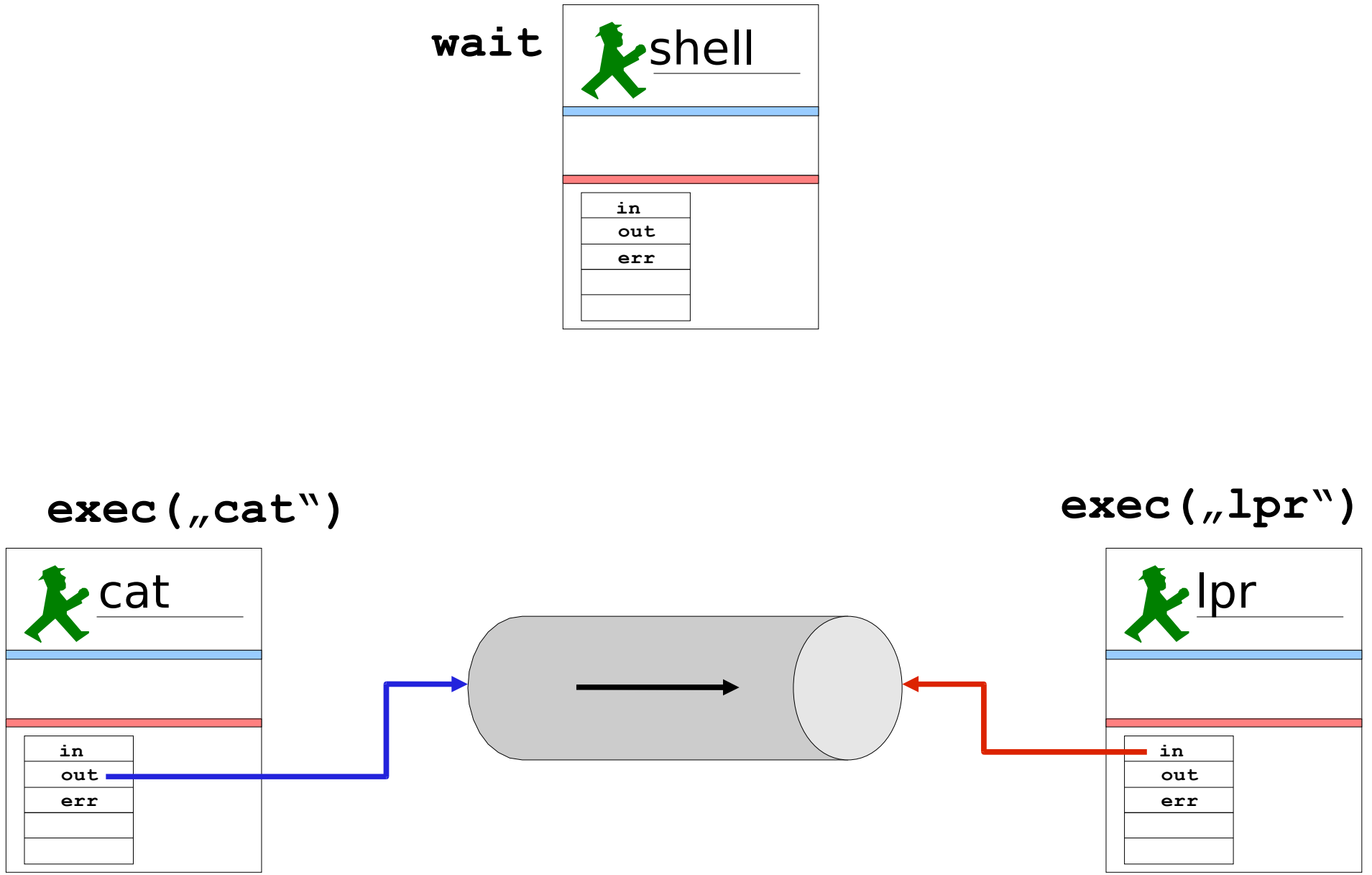


Aufbau einer Filterkette mit Pipes

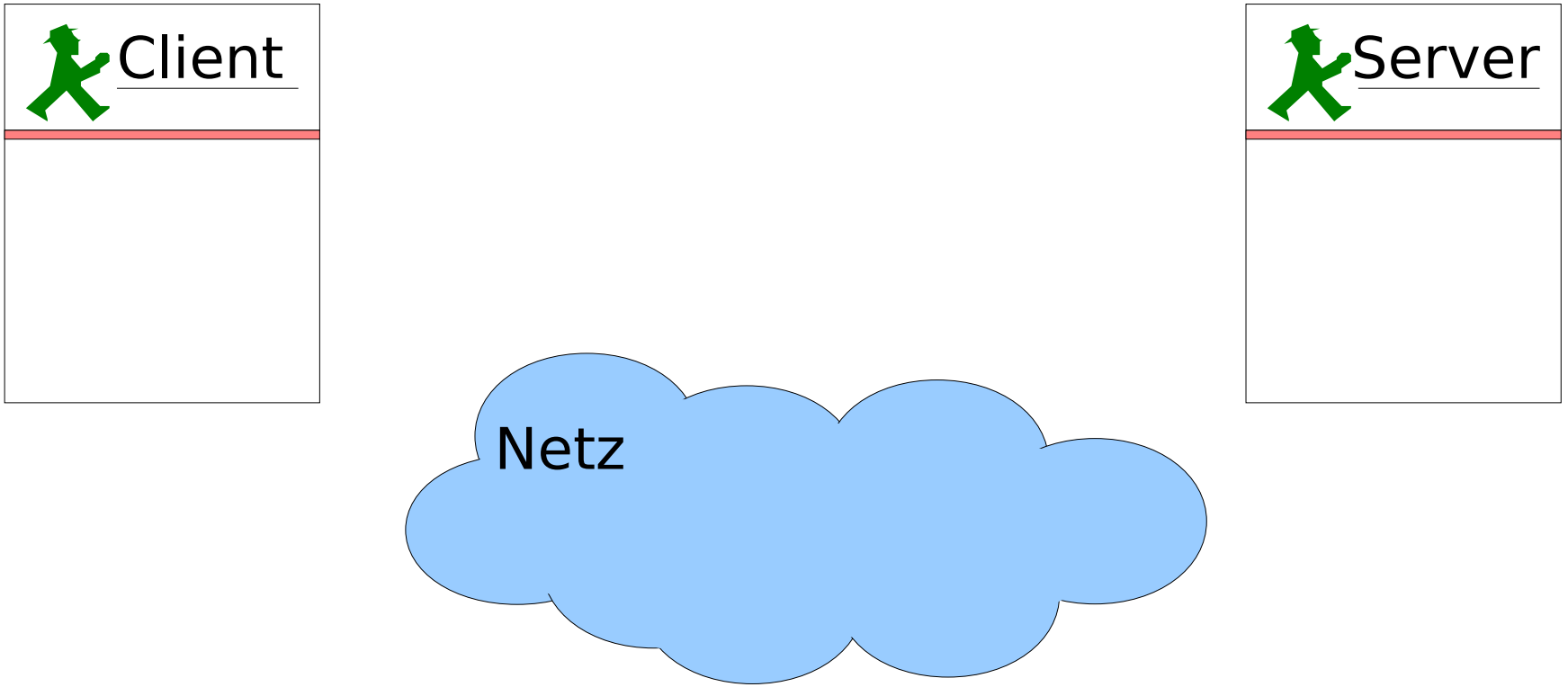




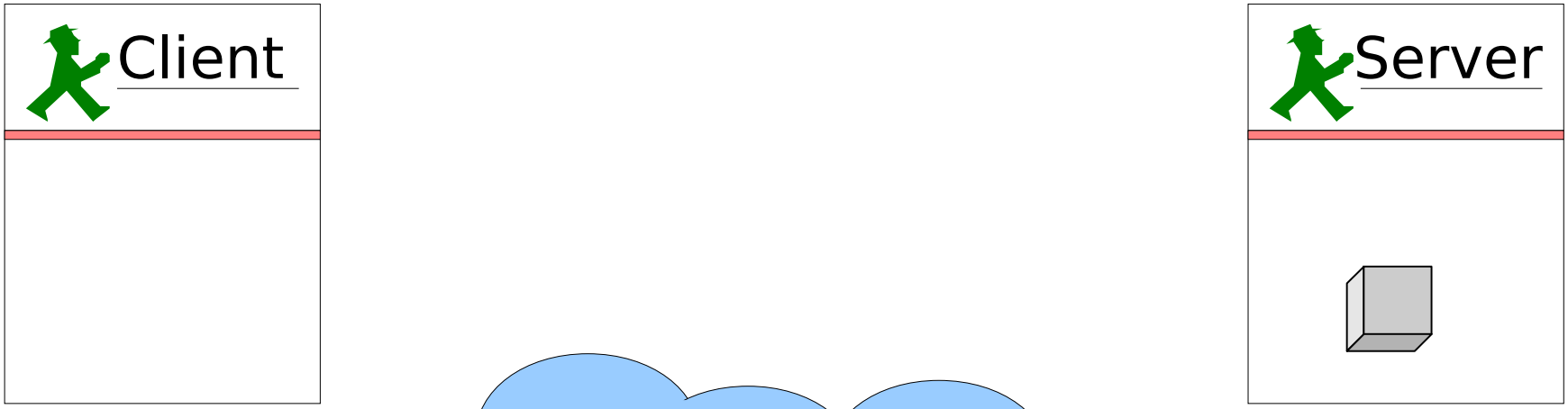
Aufbau einer Filterkette mit Pipes



Sockets



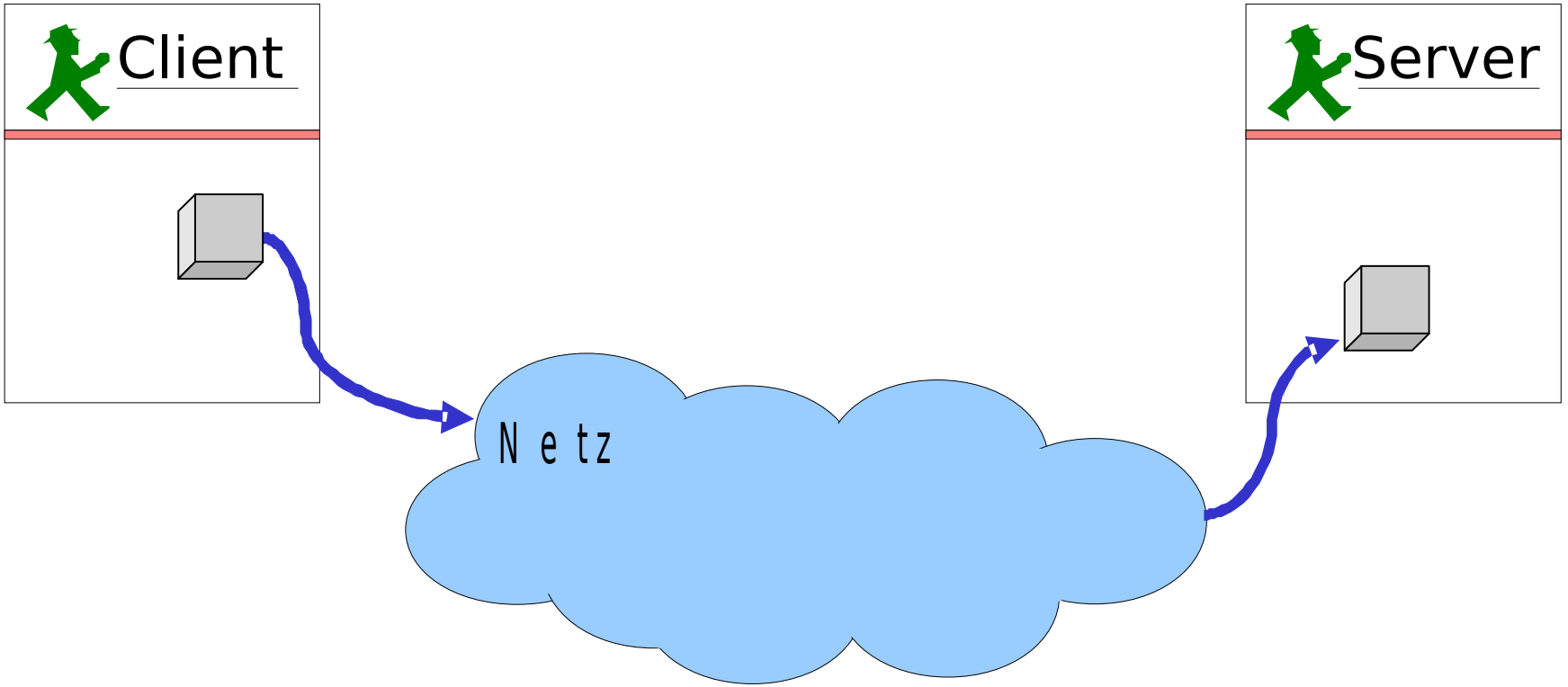
Sockets



Server

```
create sock(protocol type)
bind(27)
listen
accept
```

Sockets



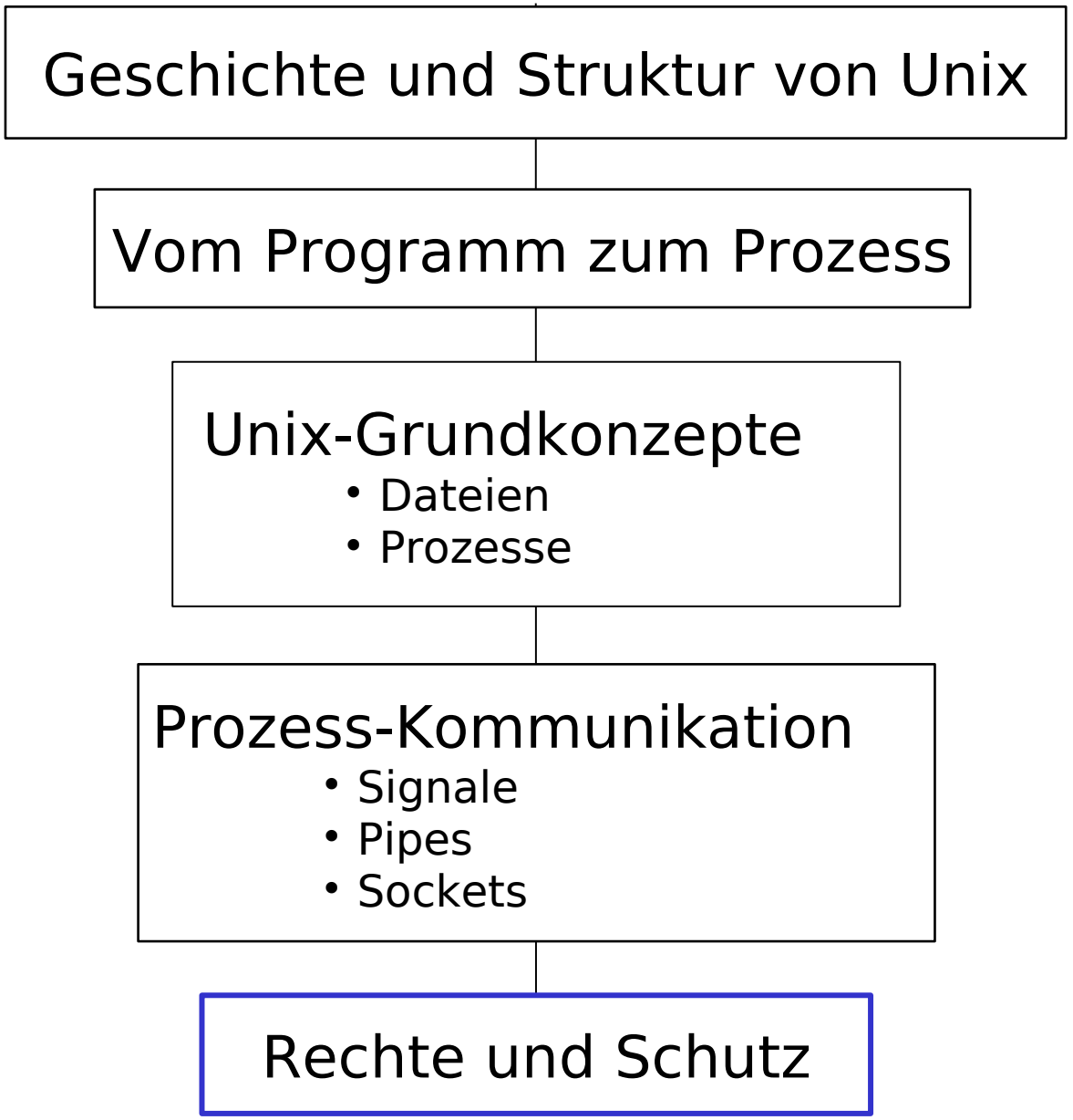
Client

```
create sock(protocol type)
connect(27)
```

Server

```
create sock(protocol type)
bind(27)
listen
accept
```

Wegweiser


Geschichte und Struktur von Unix
Vom Programm zum Prozess
Unix-Grundkonzepte
• Dateien
• Prozesse
Prozess-Kommunikation
• Signale
• Pipes
• Sockets
Rechte und Schutz

Rechte: Benutzer

- In Unix werden Benutzer (Prinzipale) dargestellt durch UId (User-Id) und GId (Group-Id)
- Zuordnung (klassisch): /etc/passwd
  (jeder kann zugreifen, verschlüsselt)
- Benutzer gehören zu einer (oder mehreren) Gruppen
  Zuordnung: /etc/group

Benutzer und Gruppen in Unix

/etc/passwd

```
root:x:0:0:Björn:/root:/bin/bash
sqrt:x:0:0:Mario:/root:/bin/tcsh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:100:sync:/bin:/bin/sync
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
irc:x:39:39:ircd:/var:/bin/sh
christiane:x:1000:100:Christiane:
     home/users/christiane:
     /bin/bash
johannes:x:1001:100:Johannes:
     /home/users/johannes:
     /bin/bash
hen:x:1002:100:Hendrik:
     /home/users/hen:
     /bin/tcsh
micha:x:1006:100:Michael:
     /home/users/micha:
     /bin/tcsh
...
```

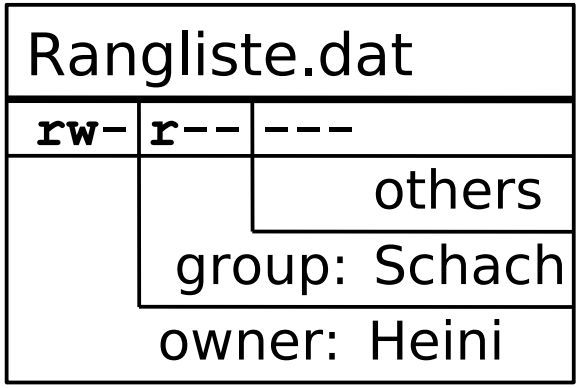
/etc/group

```
offline:x:102:ulli,iwer,veritaz

oea:x:103:veritaz,hen,bd1,iwer,
bjoern,robert,johnny,johannes,
ulli,nico

ese:!:104:iwer,veritaz,hen,chris,
benjamin,keiler,mario,ralf,bd1

www:!:105:chris,bjoern,iwer,
veritaz,hen,robert,anatol

ftp:x:106:chris,iwer

ifc:x:107:hen,reinhold,ulli,iwer,
micha
```

Rechte: Benutzer und Dateien

Zugriffsrechte zu Dateien festgelegt in Bezug auf Benutzer

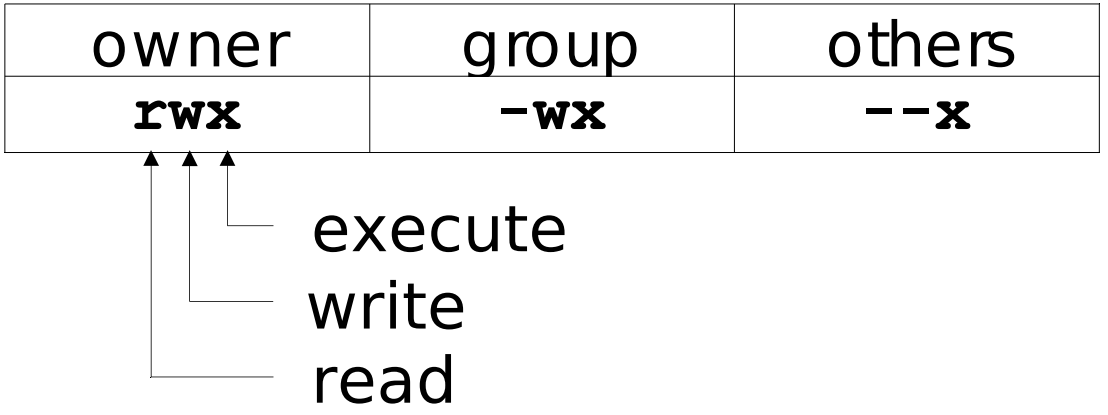
- jede Datei hat Attribute für Besitzer
  - owner: UId
  - group: Gid

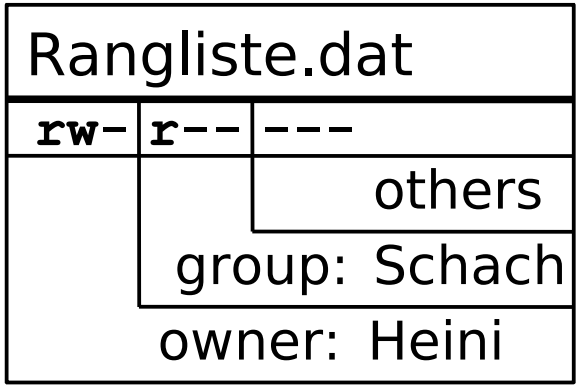


- Rechte an einer Datei werden festgelegt in Bezug auf
  - owner
  - group
  - others (= Rest der Welt)

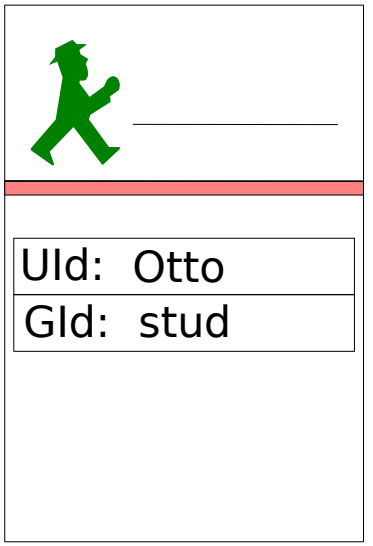
Benutzer und Prozesse

Rechte an Daten werden festgelegt in Bezug auf





- Jeder Prozess übernimmt UId und GId vom „Eltern“-Prozess, die Rechte eines Benutzers leiten sich von UId, GID ab



Zusammenfassung/Weiterführung

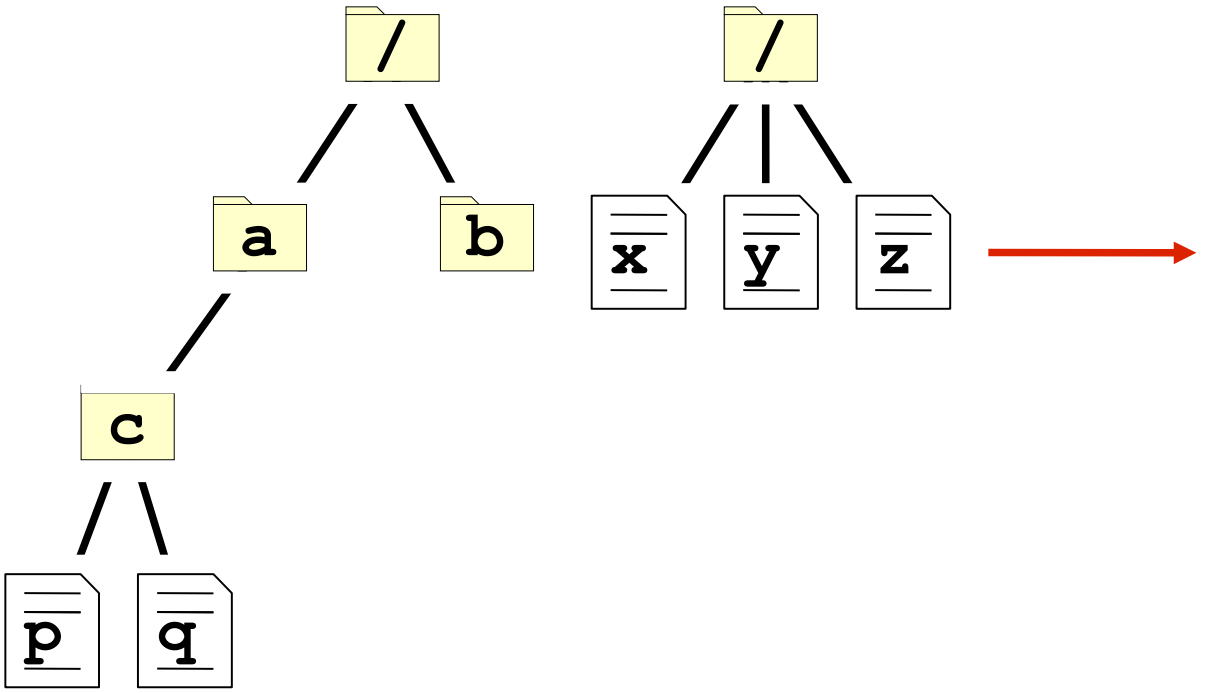
- Erfolgreiches Betriebssystem (akademisch, Workstations, Server)
  - ➔ aber zu viele Versionen ...

- Linux: Re-Implementierung von Unix
  - ➔ ernstzunehmende Konkurrenz zu W*

Shell-Ebene: Mounting

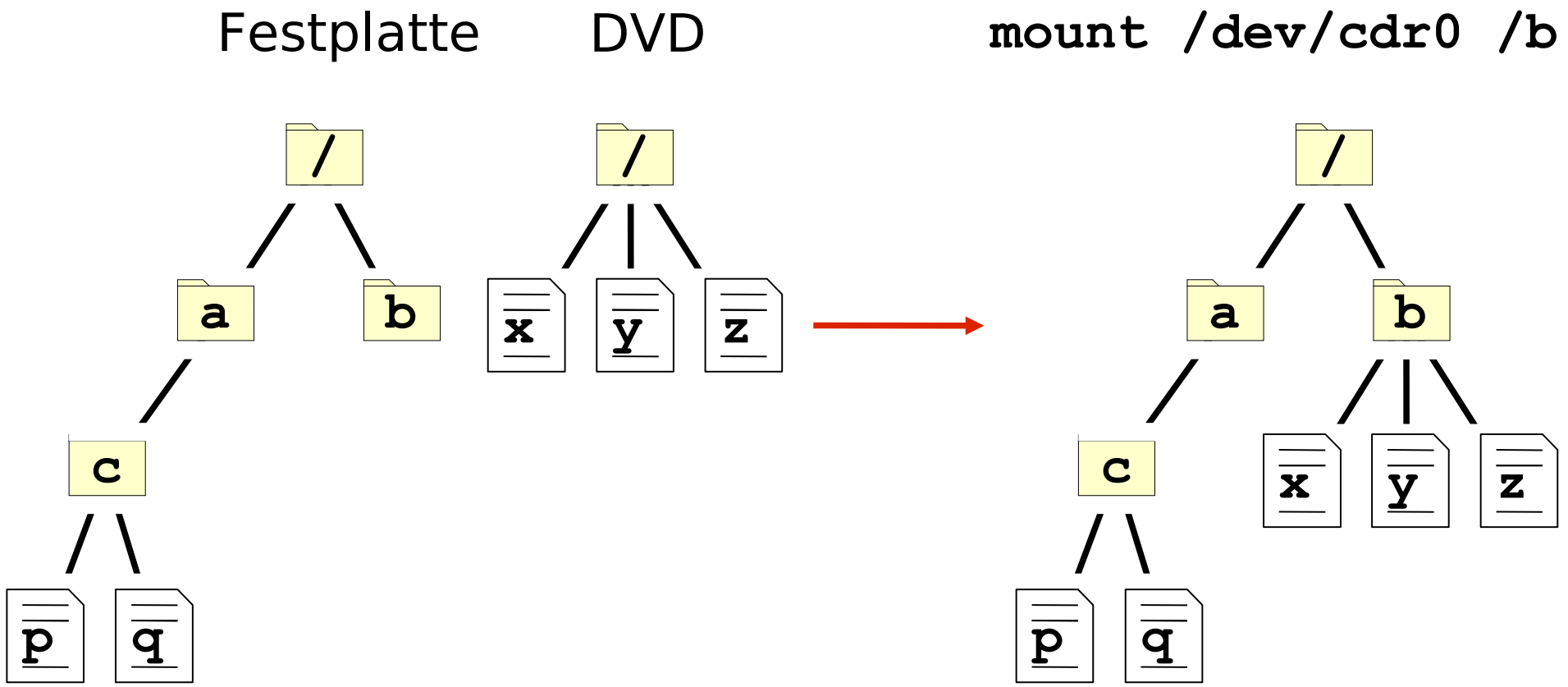
- **„Mounting“**

Festplatte DVD **`mount /dev/cdr0 /b`**



Shell-Ebene: Mounting

- **„Mounting“**



Prozesse auf Shell-Ebene

`cp src dest`

- ein neuer Prozess („Kind-Prozess“) wird erzeugt
  Parameter (`src`, `dest`) werden übergeben
  Eltern-Prozess (shell) wartet

`cp src dest &`

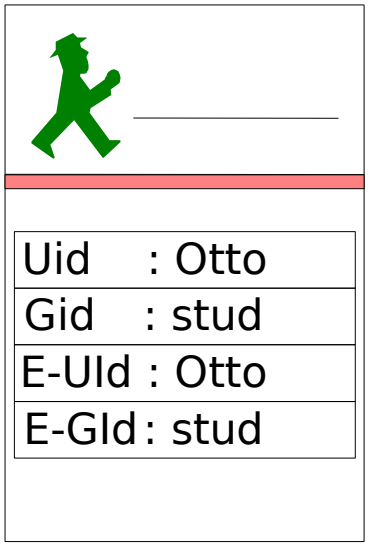
- Eltern-Prozess (shell) wartet nicht

Signale

- Prozessgruppen
  - sind einem Terminal zugeordnet
  - Terminal-Signale gehen an alle Prozesse der zugeordneten Gruppe
  - killpg
- Fehlerbehandlung

Benutzer und Prozesse

- Jeder Prozess repräsentiert einen Benutzer.
  Prozess-Attribute:
  - UId, GId
  - Effective-UId, Effective-GId



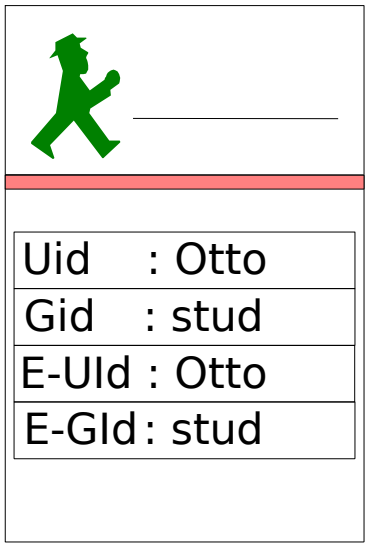
- Nur wenige hochprivilegierte Prozesse dürfen UId und GId manipulieren, z.B. Login-Prozess.
- Nach Überprüfung des Passwortes setzt Login-Prozess UId, GId, Eff-UId, Eff-GId.

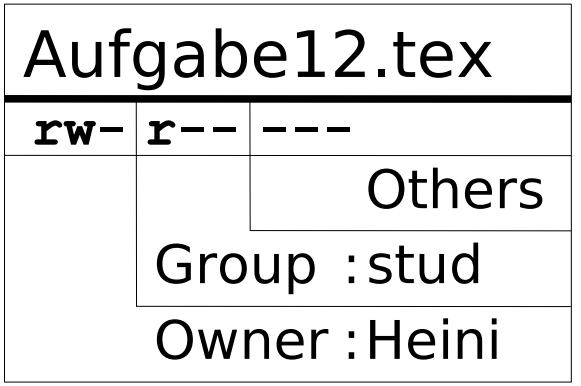
➔ Alle anderen Prozesse: Kinder des Login-Prozesses.

- Kinder erben Attribute von Eltern.

Prozesse und Dateien

Die Attribute E-UId und E-GId bestimmen beim Zugriff auf Dateien die Rechte eines Prozesses.



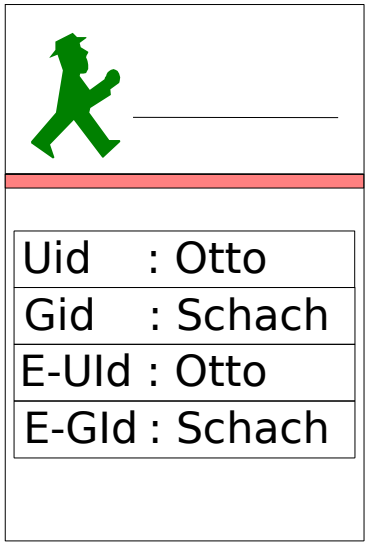


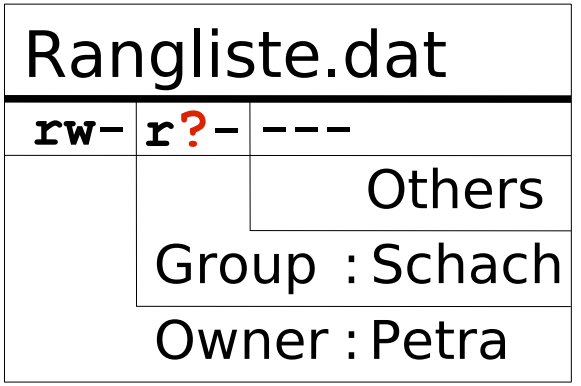
Problem: Rechteerweiterung

Beispiel: Schachrangliste

- Jeder Teilnehmer soll lesen können.
- Jeder Teilnehmer soll seine Ergebnisse schreiben können.
- Kein Teilnehmer soll darüber hinaus schreiben dürfen (Fälschung der Rangliste).

➔ Ziel: Jeder Teilnehmer soll nur seine korrekten Ergebnisse schreiben können.



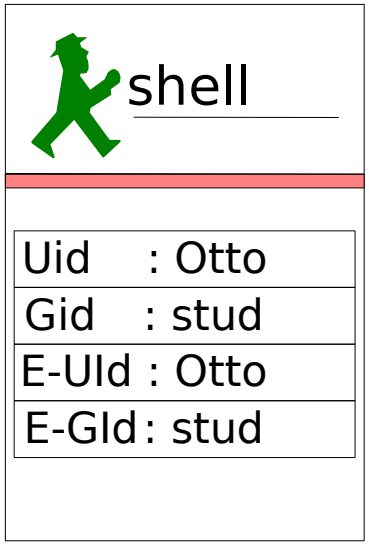


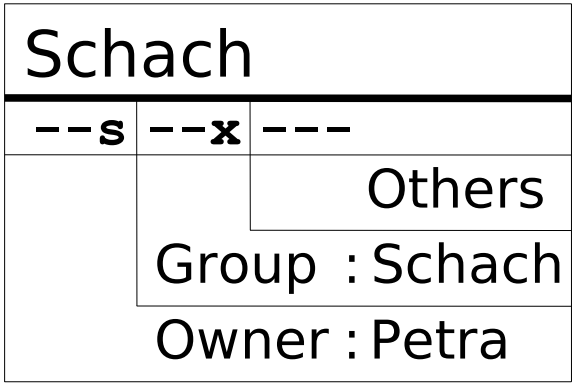
Unix-Lösung: SetUId-Mechanismus

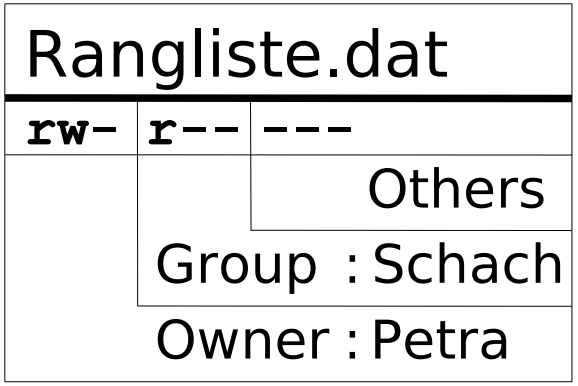
- Datei, die vertrauenswürdigen Programmcode (z. B. Schach) enthält, besitzt Kennzeichnung als „Set-UID“ (**s**).

- ➔ Bei **exec** auf Set-UId Programme erhält ausführender Prozess als Effektive UId die UId des Installateurs (Owners)
  des Programms (genauer: der Datei, die Programm enthält).

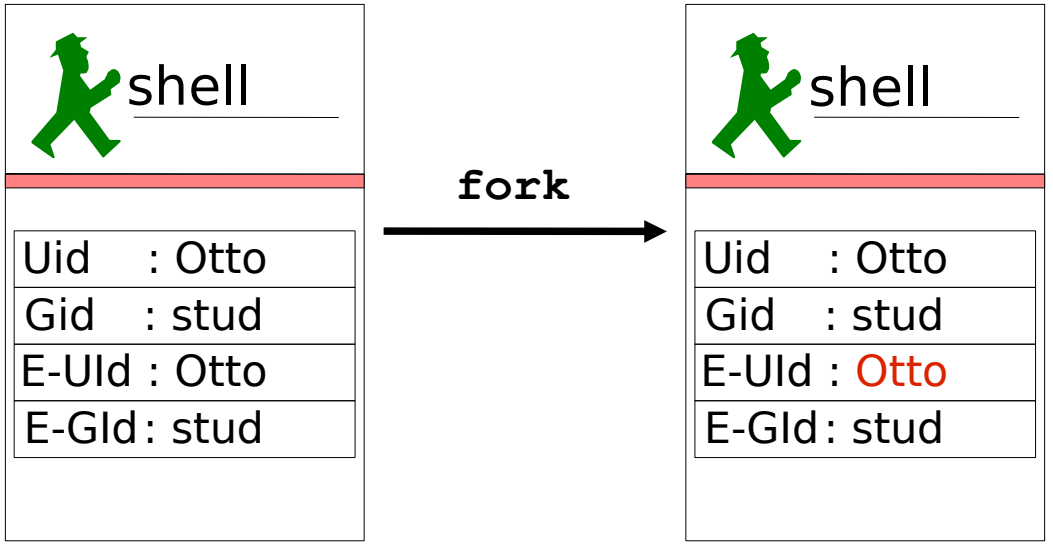
SetUId am Beispiel Rangliste

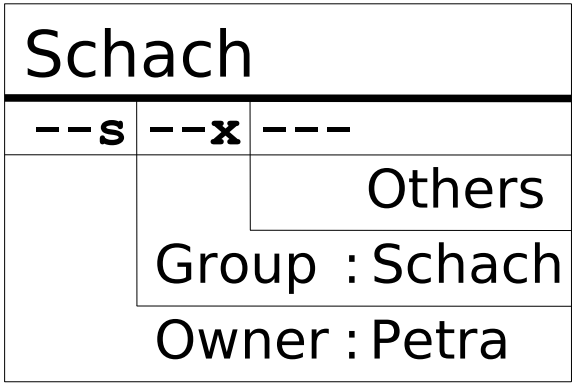


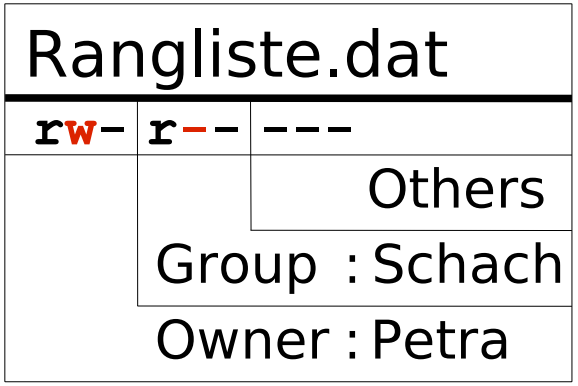




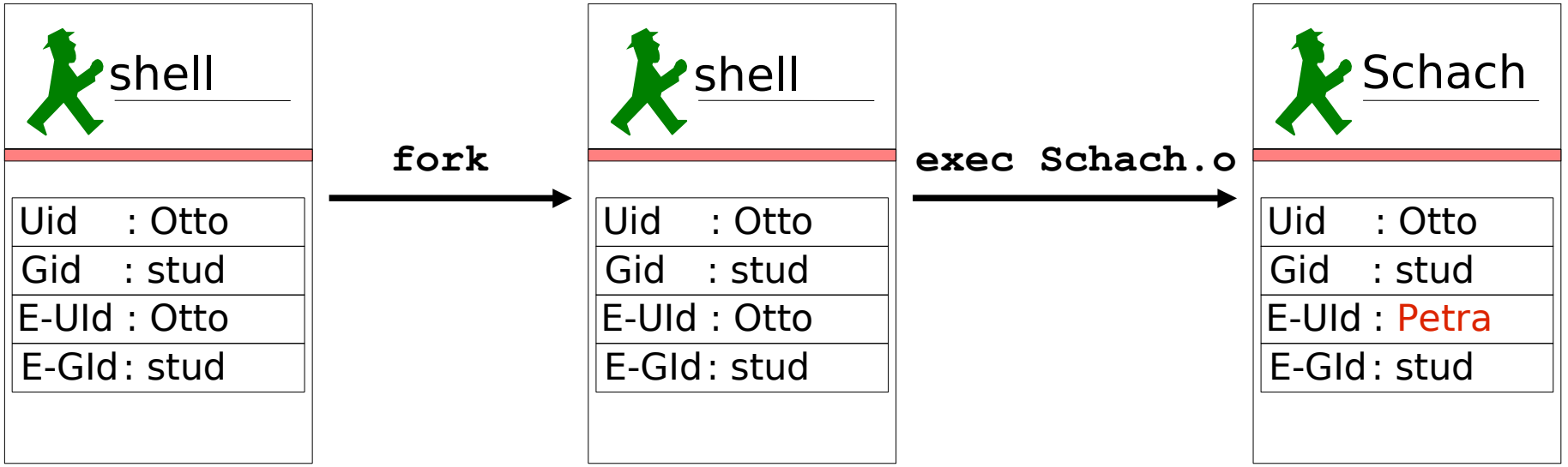
SetUId am Beispiel Rangliste

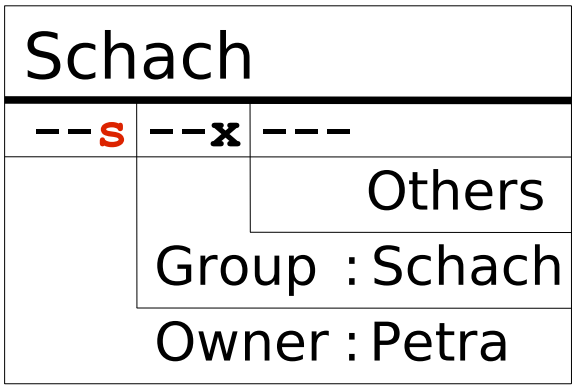


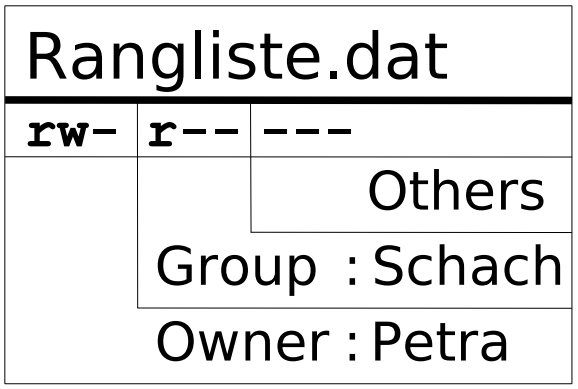




SetUId am Beispiel Rangliste







SetUId

- Erweiterung der Rechte eines Benutzers genau für den Fall der Benutzung dieses Programms.

➔ Installateur vertraut dem Benutzer, wenn er dieses Programm nutzt.

Probleme :

- Programmfehler führen zu sehr großen Rechteerweiterungen
- Bsp.: shell-Aufruf aus einem solchen Programm heraus

Paralleler Zugriff auf Dateien

- Locking (Synchronisation von Prozessen beim Dateizugriff)
  - ganze Dateien
  - Dateibereiche

- **`lock (Anfang, Länge)`**

➔ shared vs. exclusive

➔ blocking vs. non blocking