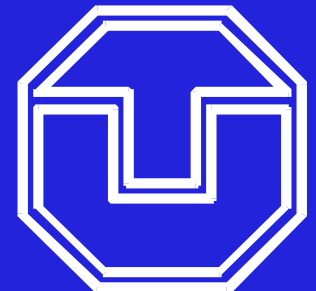


Fallbeispiel Unix

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

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

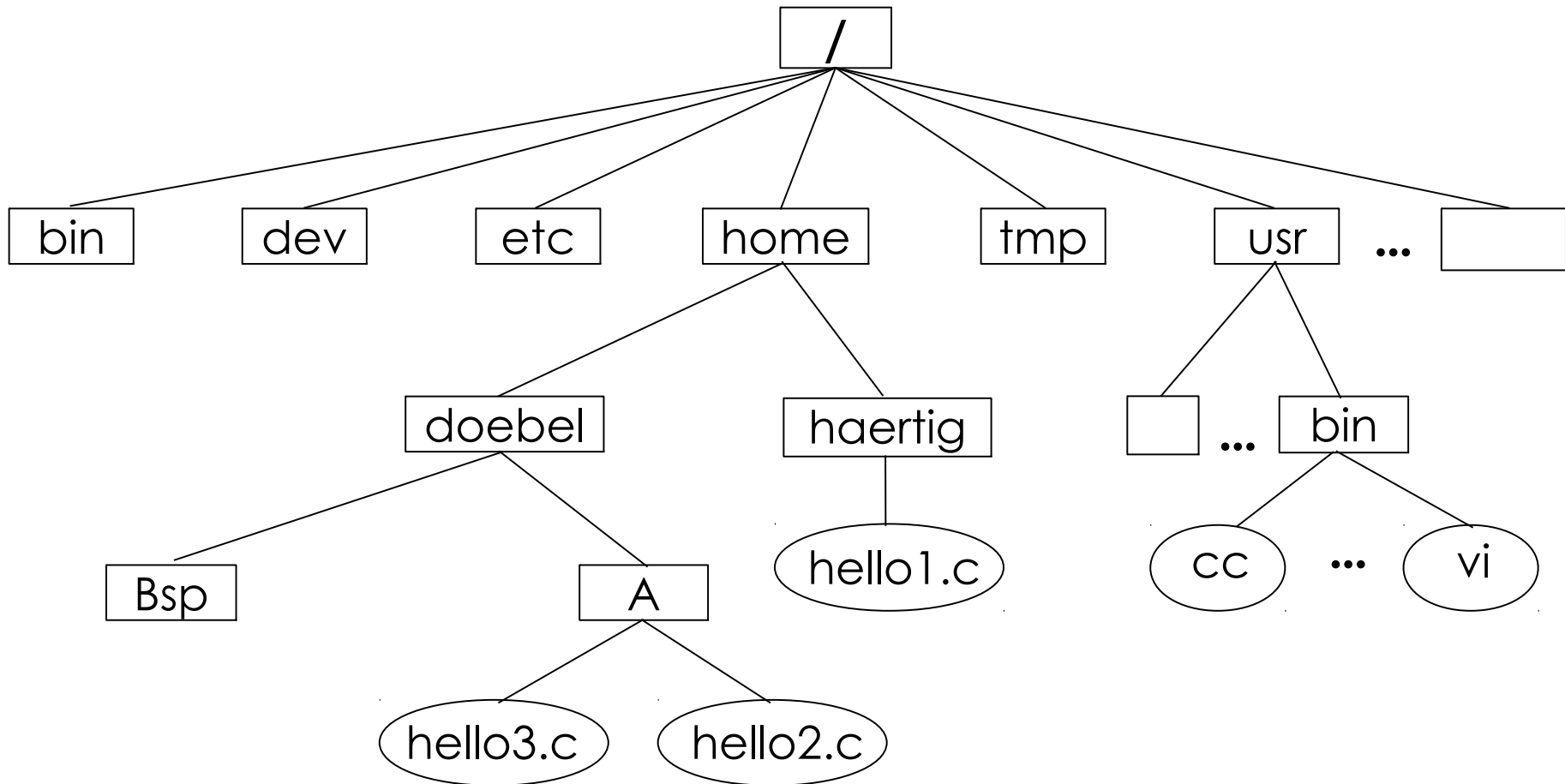
rm a

Datei nicht mehr unter dem Namen `a` zugreifbar
aber nach wie vor unter `/etc/xy`

rm /etc/xy

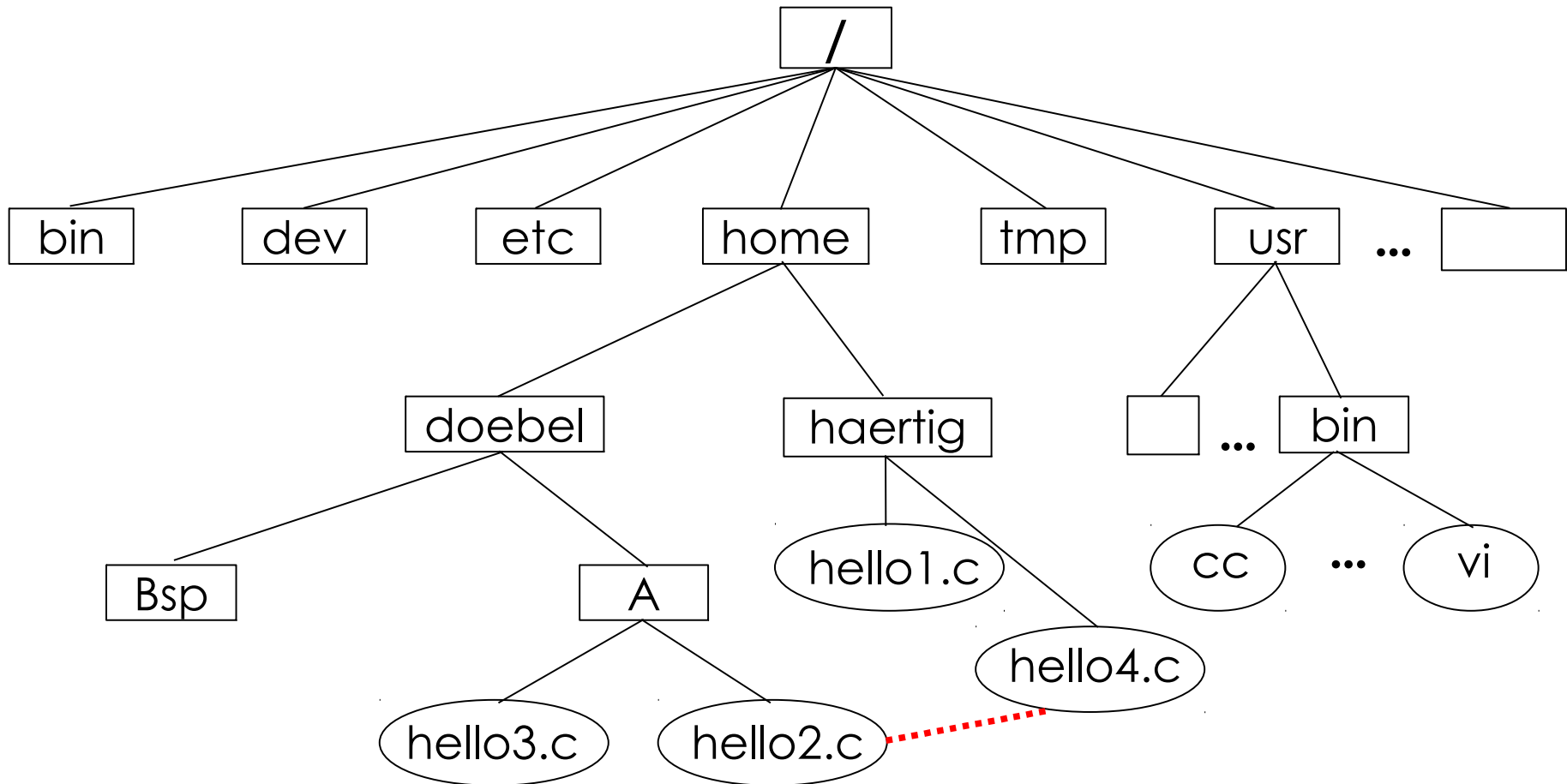
keine Name mehr für Datei vorhanden => gelöscht

Verzeichnisstruktur



In hello2.c /home/haertig/hello4.c

Verzeichnisstruktur



In 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

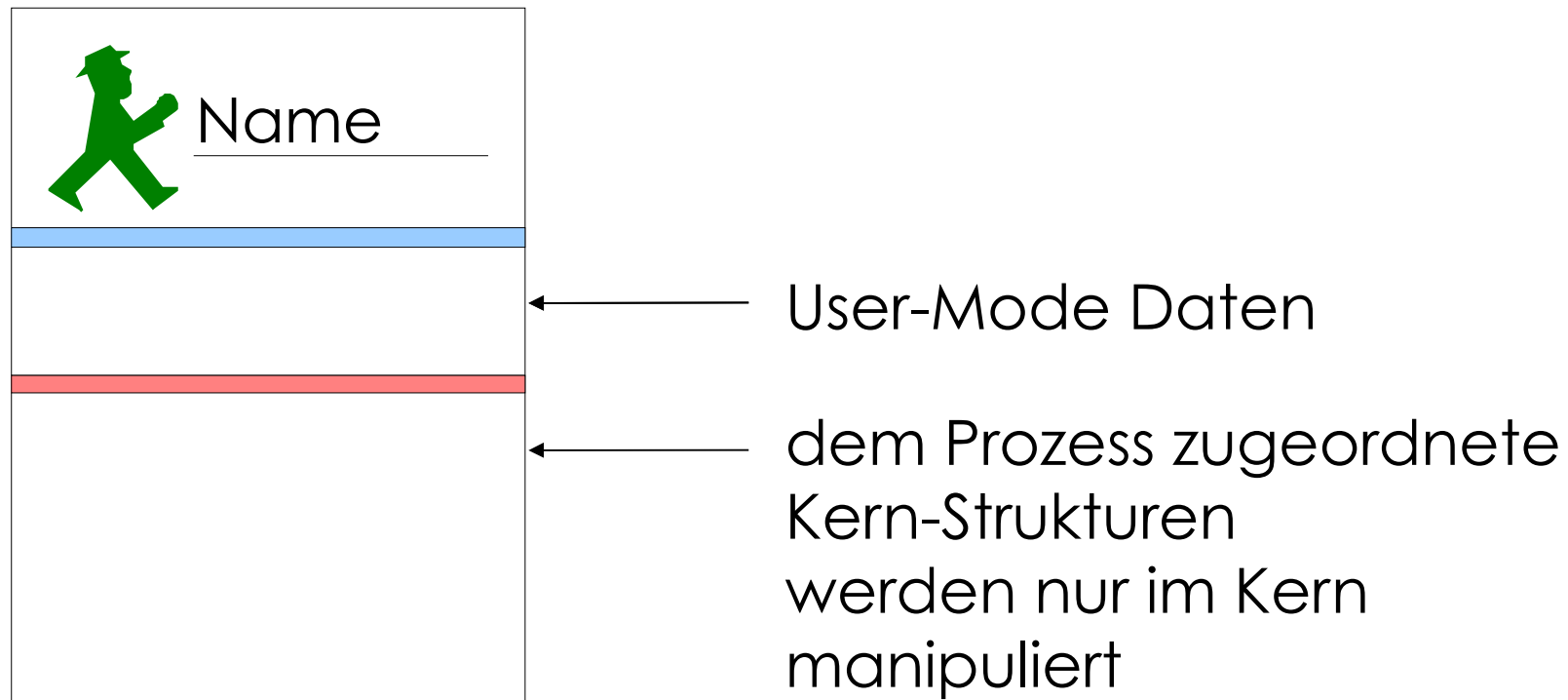
Unix-Prozess

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

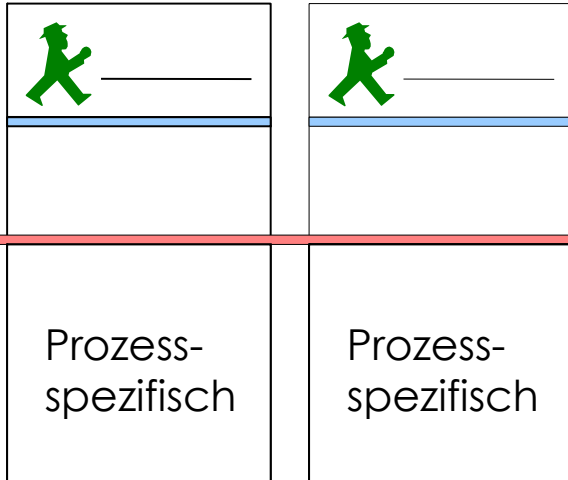
Viele Prozesse pro Rechner

- Benutzerprozesse
- Hintergrund-Systemprozesse („daemons“)

Darstellung Unix-Prozesse im Folgenden



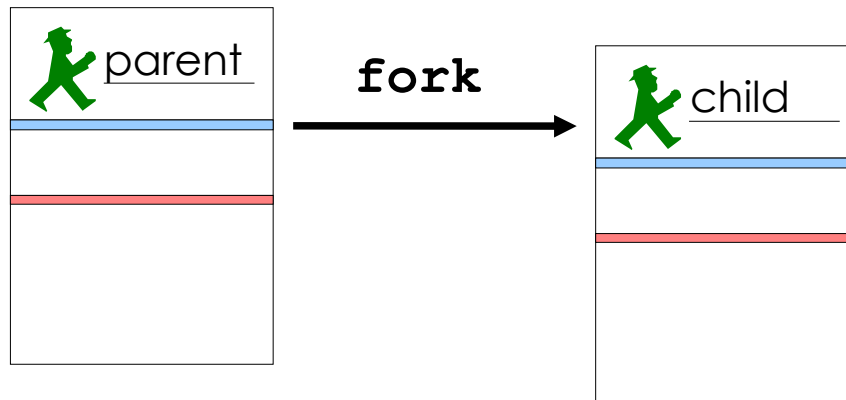
Kern-Adressraum



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

Kernaufwurf: Erzeugung von Prozessen

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



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

Weitere Kernaufufe

s = exec(file, argument, environment)

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

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);
```



```
X = fork();
```

```
// erzeugt Kopie des Aufrufers (d.h. der Shell)
```

```
// Kind erhält fd des Eltern-Prozesses
```

```
// beide Prozesse setzen Abarbeitung hinter fork fort
```

```
if (X < 0) {
```

```
    // Fehlerbehandlung
```

```
} else if (X != 0) {
```

```
    // Parent-Prozess
```

```
    waitpid(X, &status, 0); // warte auf Kind-Prozess
```

```
} else {
```

```
    // Child
```

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

```
}
```

Beispiel: Shell mittels fork/exec

```
read (command, params);  
  
X = fork();  
  
if (X < 0) {  
  
} else if (X != 0) {  
    // Parent  
    waitpid(X, &status, 0);  
} else {  
    // Child  
    exec (command, params, env);  
}
```

```
read (command, params);  
  
X = fork();  
  
if (X < 0) {  
  
} else if (X != 0) {  
    // Parent  
    waitpid(X, &status, 0);  
} else {  
    // Child  
    exec (command, params, env);  
}
```

Beispiel: Shell mittels fork/exec

```
read (command, params);  
X = fork();  
  
if (X < 0){  
} else if (X != 0) {  
    // Parent  
    waitpid(X, &status, 0);  
} else {  
    // Child  
    exec (command, params, env);  
}
```

```
// Program-Code for command
```

```
.....
```

```
exit
```

Threads (Unix Nachfolger)

- Bibliotheksfunktionen (z.B. „pthread“)
- Linux syscall; clone(....)

```
main()  
{  
  p1 = pthread_create(thread_function);  
  p2 = pthread_create(thread_function);  
  
  // do something else  
  
  pthread_join(p1);  
  pthread_join(p2);  
}
```

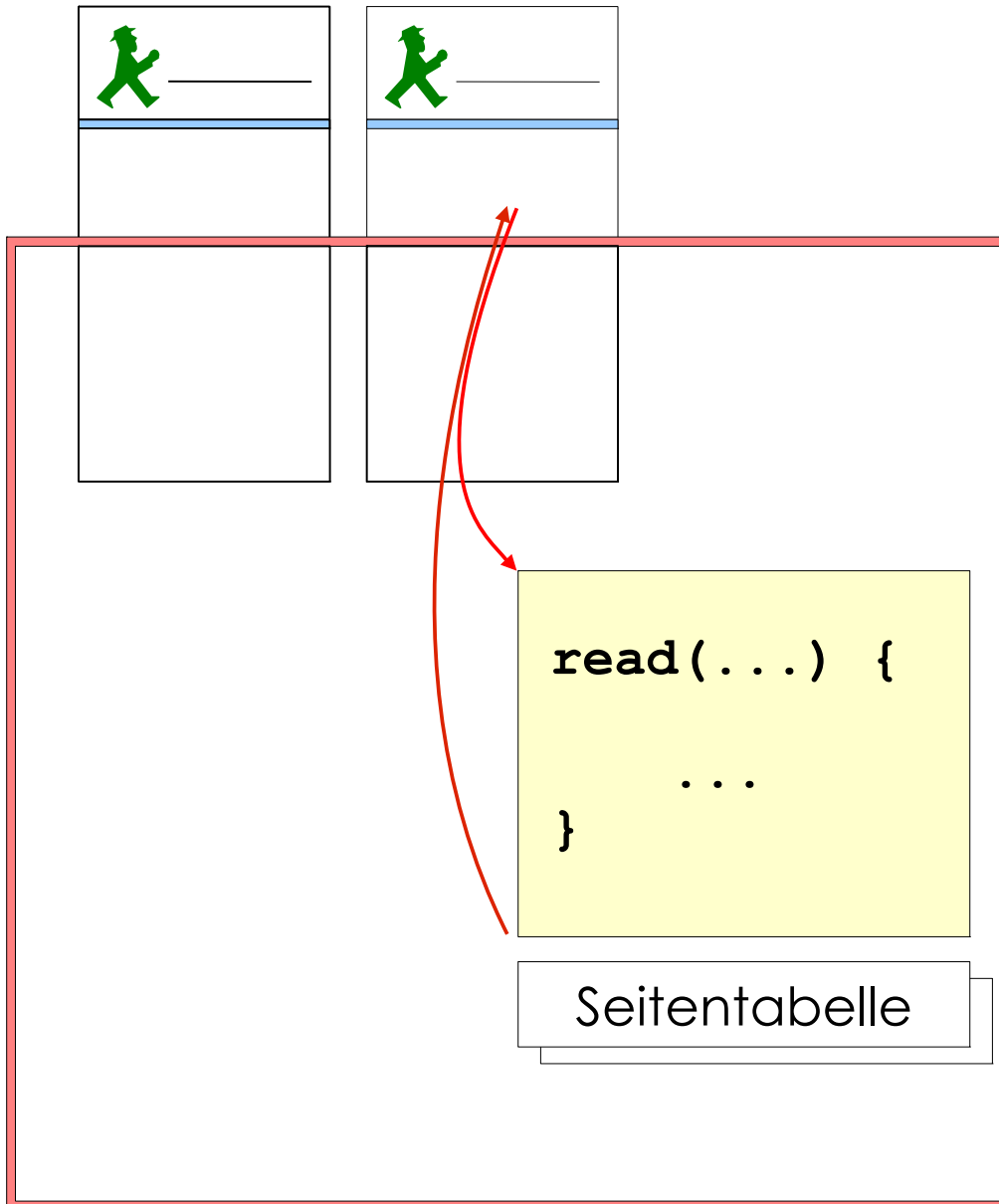
Kernaaufrufe (System-Calls)

Beispiel

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

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

Schutz des Kerns?



- Wie wird der Kern sicher aufgerufen?
- Wie werden Kern-Strukturen geschützt?
(Beispiele später)

Prozessor-Modi: usermode/kernelmode

Kernelmode

Usermode

Alle Instruktionen

Teilmenge

User- und Kernel-Mode Speicher
des Adressraums

User-Mode Teil des
Adressraums

Umschalten des Adressraums

Kernaufruf im Detail

Benutzerprozess

```
read(...) {  
  
    //Parametereaufbereitung  
    ...  
    call = read;  
    INT 0X80 //trap
```

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

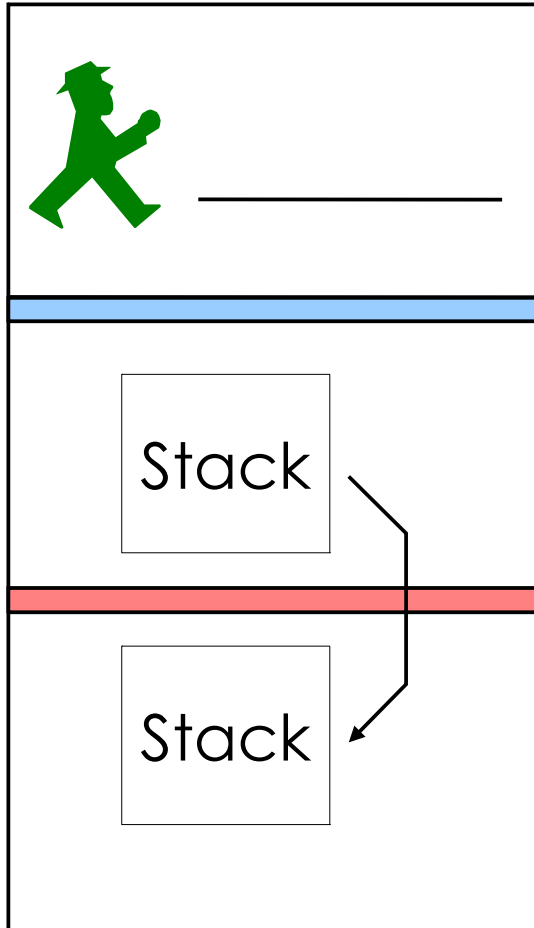
→ User-Mode: kein Zugriff auf Kern-Adressraum

Kern

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

→ Kernel-Mode: Zugriff auf Kern- und Benutzer-Adressraum

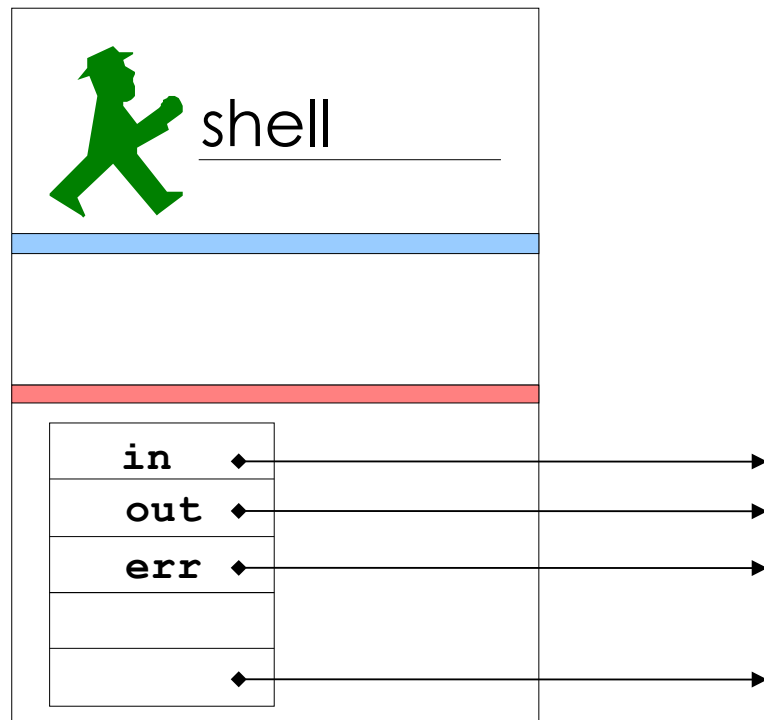
Zwei Keller pro Prozess: User, Kernel



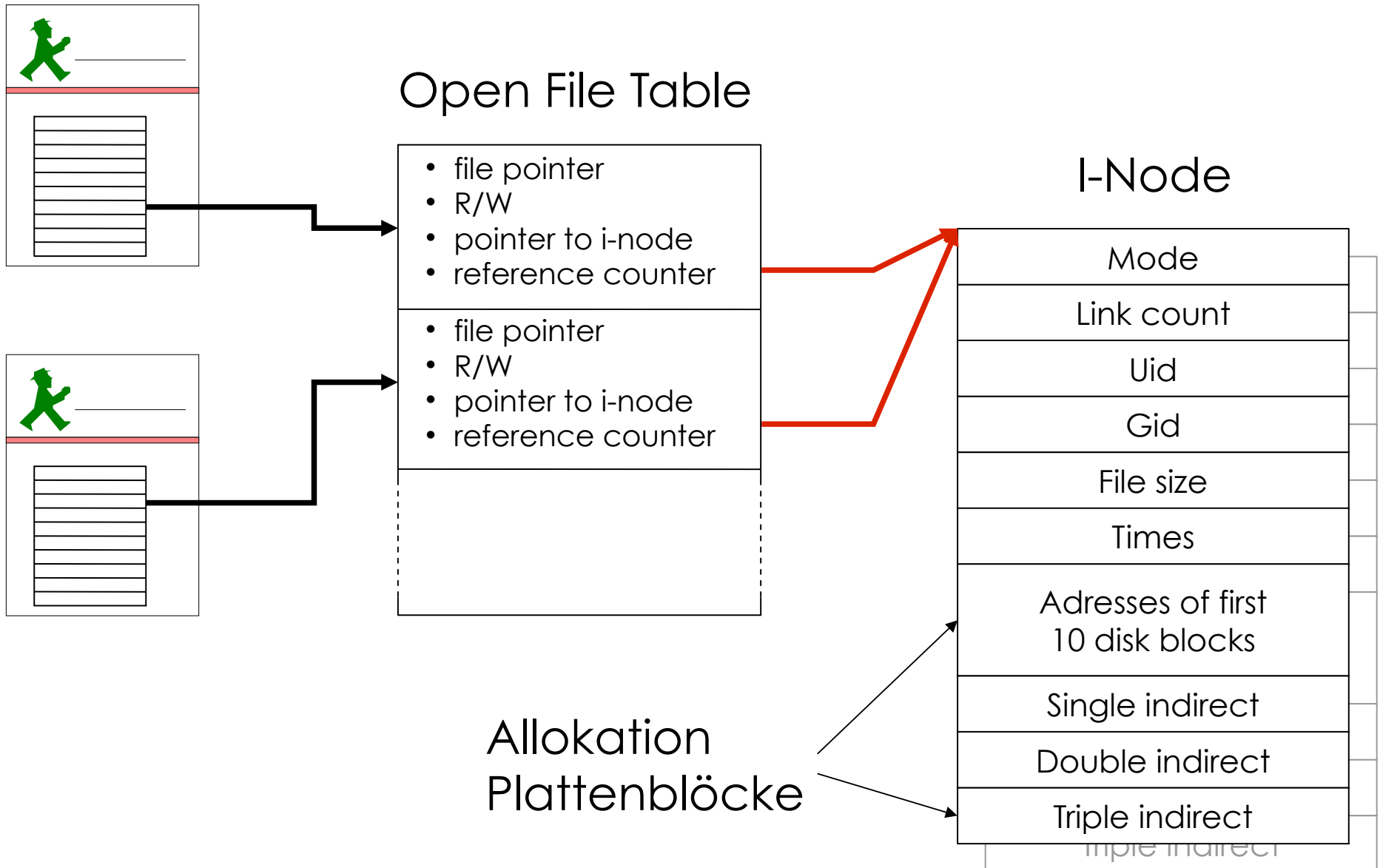
Bei Systemcalls wird

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

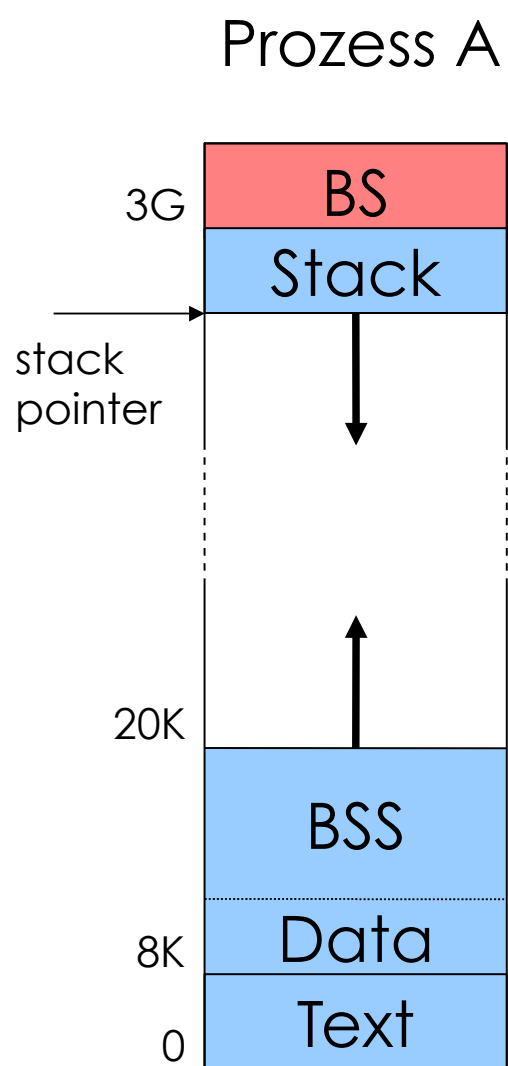
Kernschnittstelle: Datei-Deskriptoren



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

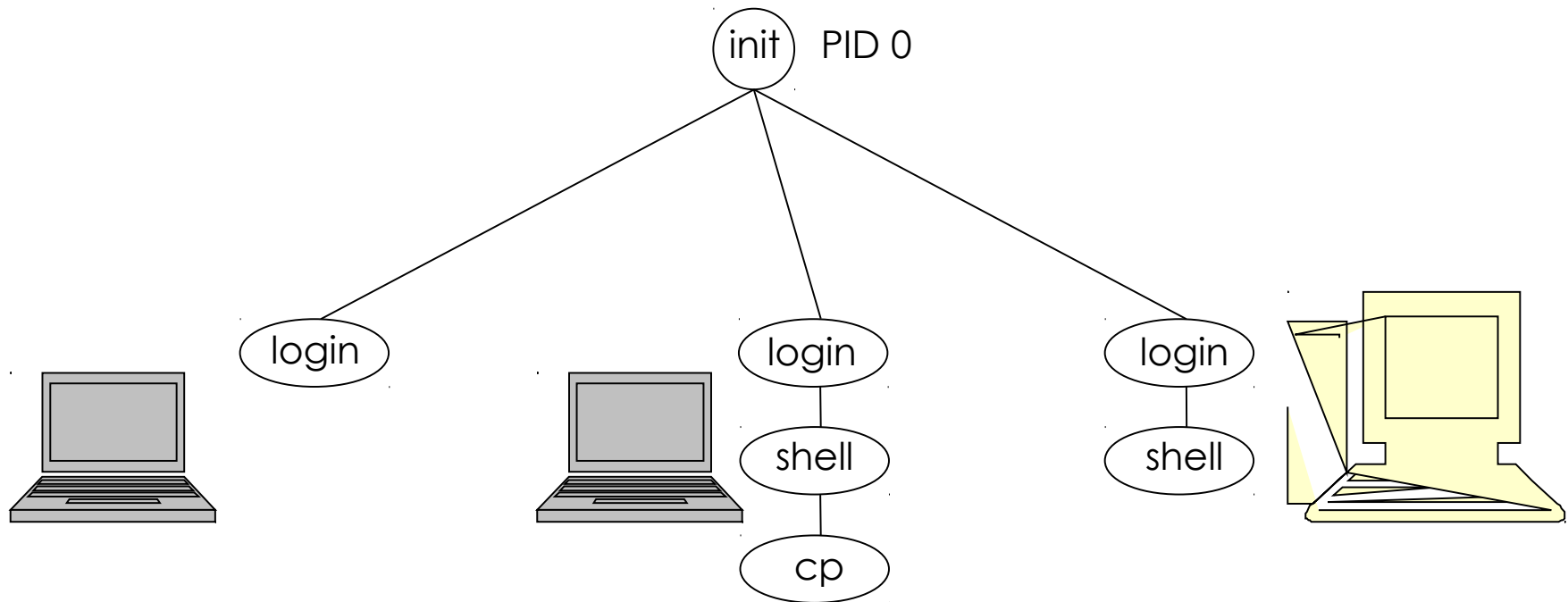
- enthält Maschinencode
- read only
- erste Seite frei zum Entdecken
nicht-initialisierter Pointer
- "shared text"

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-Kernaufufruf (**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
SIGQUIT	The user has hit the key 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.

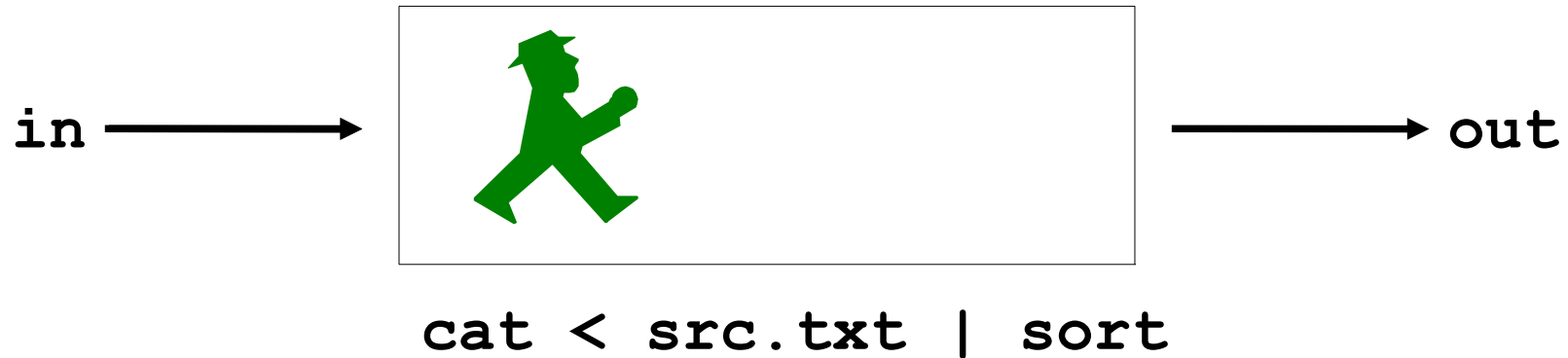
```
cat a > b
```

```
cat < a > b
```

```
cat a | lpr
```

```
cat a | sort | lpr
```

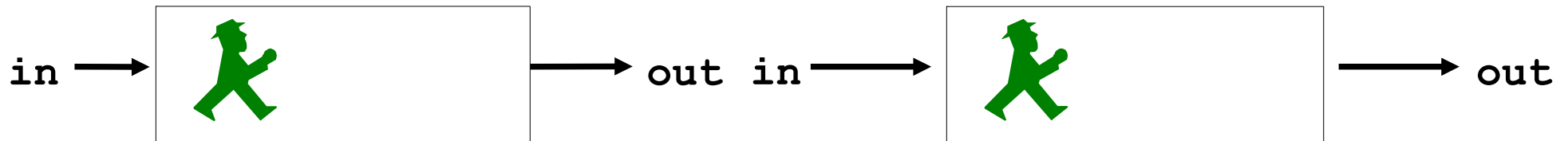
Shell-Ebene: Prozesse als Filter



`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
```

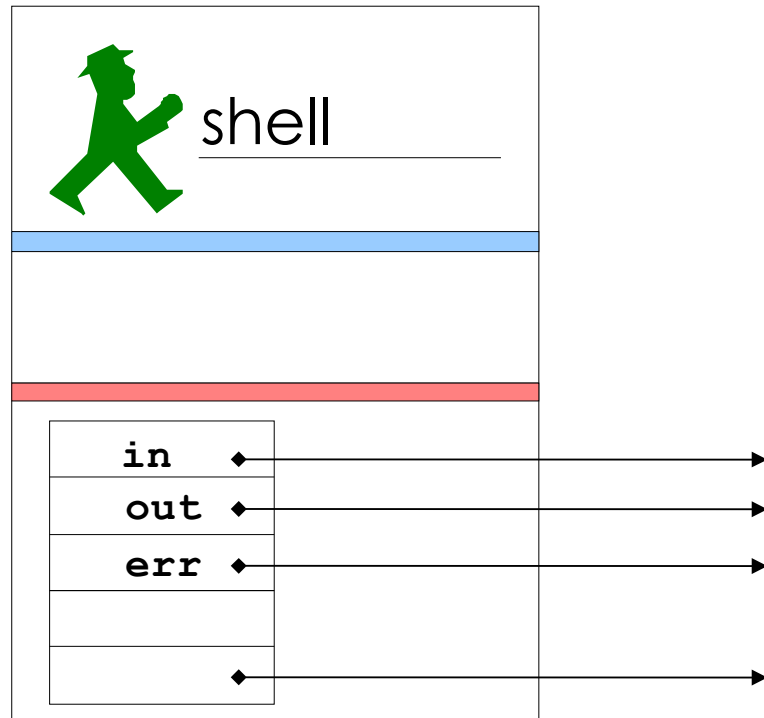
```
sort <mylist.txt | lpr
```

```
cat | sort | lpr
```

“Pipes” als spezielle Datenstrom-Dateien

→ Datenstrom als durchgängiges Konzept !

Kernschnittstelle: Datei-Deskriptoren



Beispiel: cat <in >out

```
read (command, params);

X = fork();

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

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

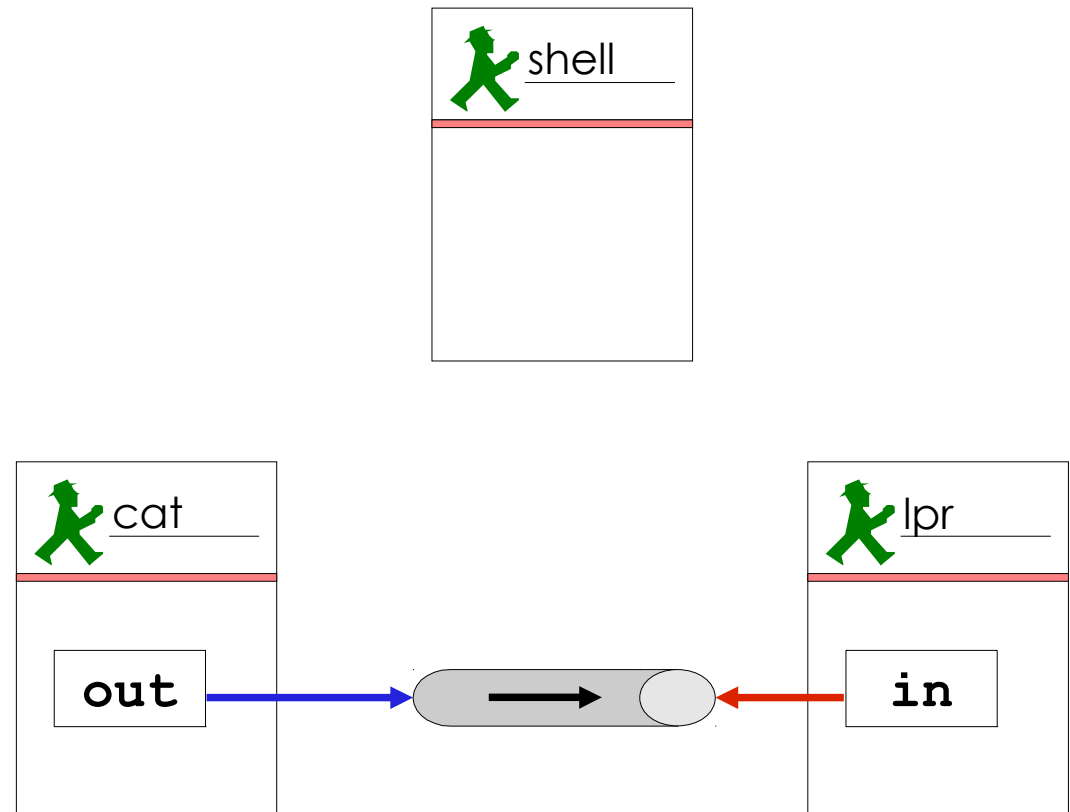
    close(0); open(in, ...); // ersetze vorh. fd
    close(1); open(out, ...); // durch in/out

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

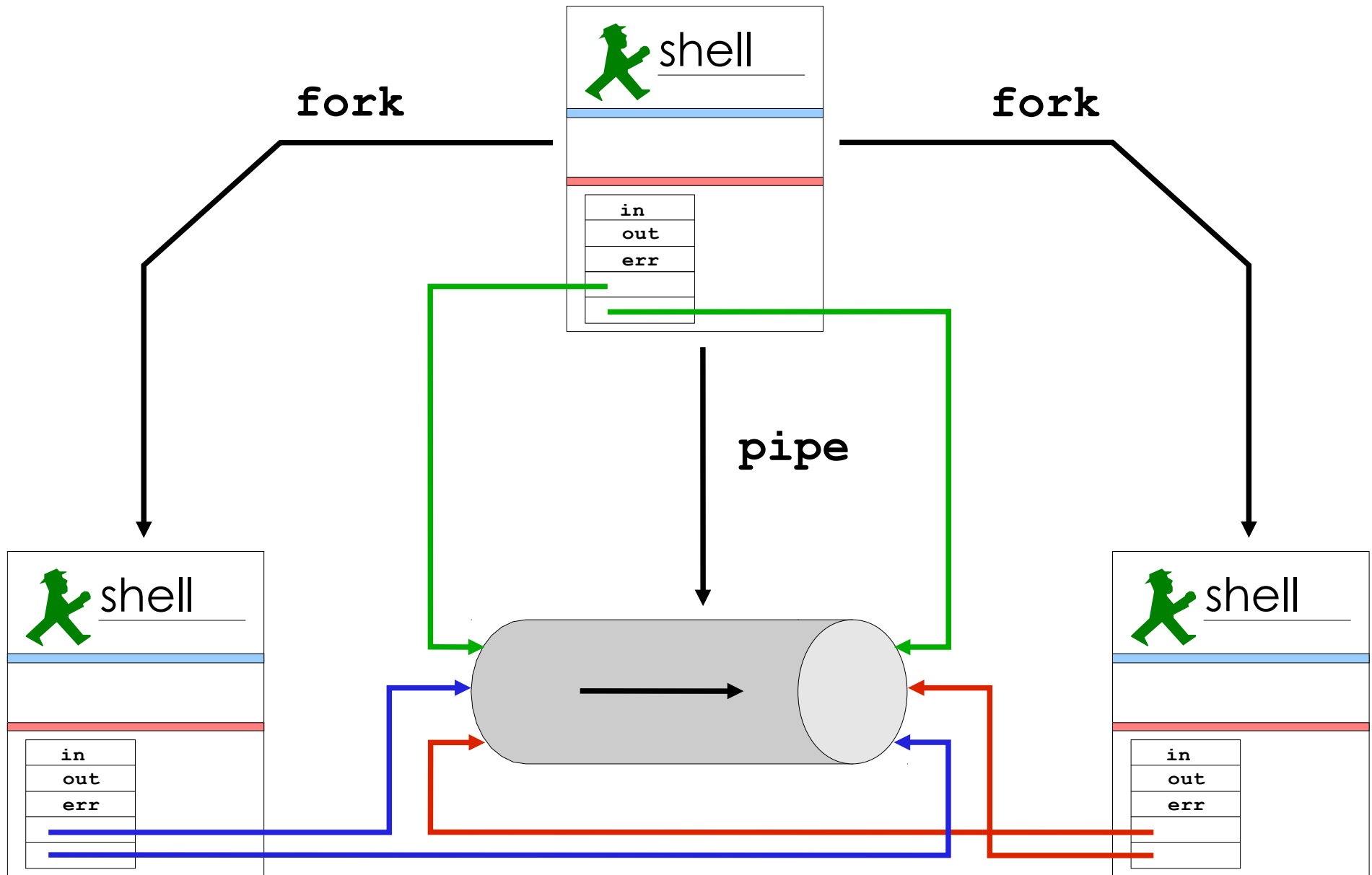
Pipes und Filterketten

z. B. **cat | lpr**

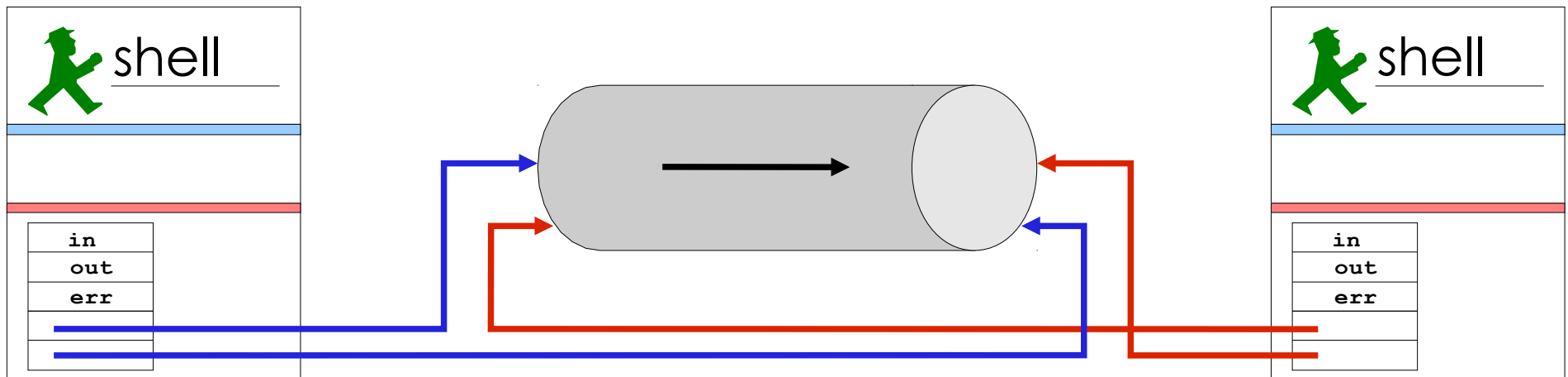
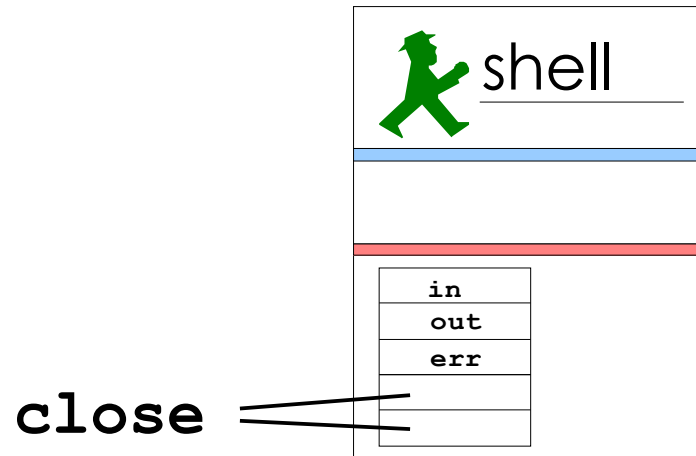
- **pipe**
erzeugt pipe mit 2 fd (fd1, fd2)
- **fork**: Kind1 für **cat**
- **fork**: Kind2 für **lpr**
- Elternteil (shell):
schliesst fd1, fd2
- Kind1:
schließt fd2
schließt stdout
fd1 → stdout
schließt fd1
exec cat
- Kind2: spiegelbildlich



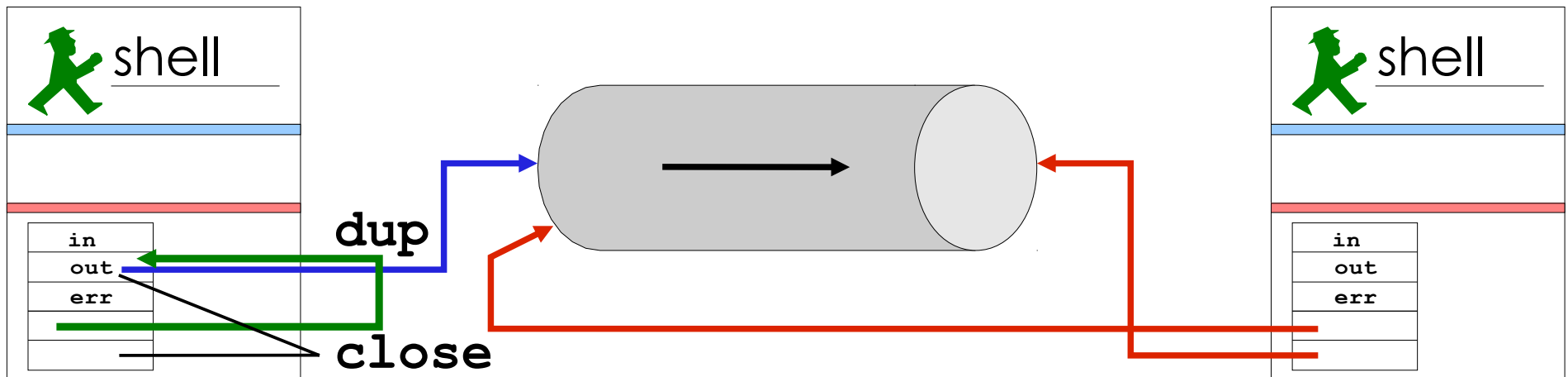
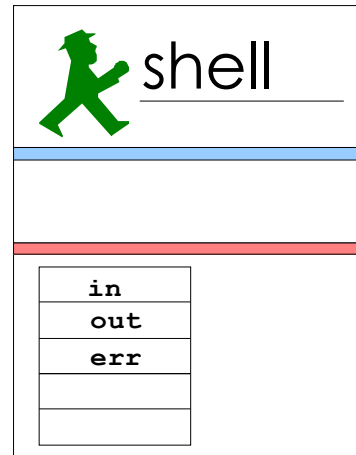
Aufbau einer Filterkette mit Pipes



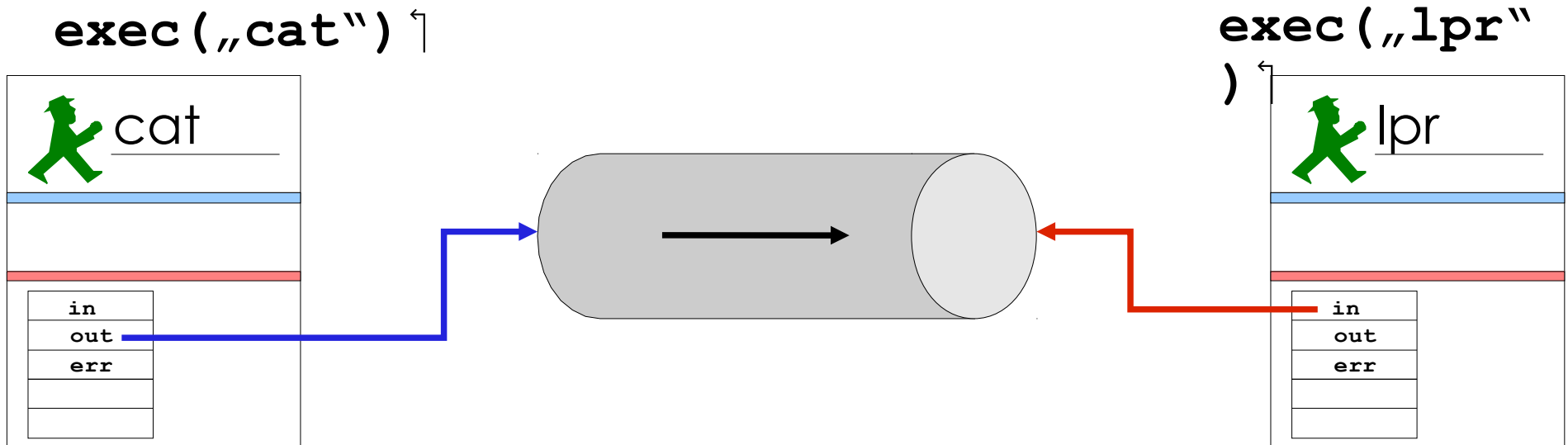
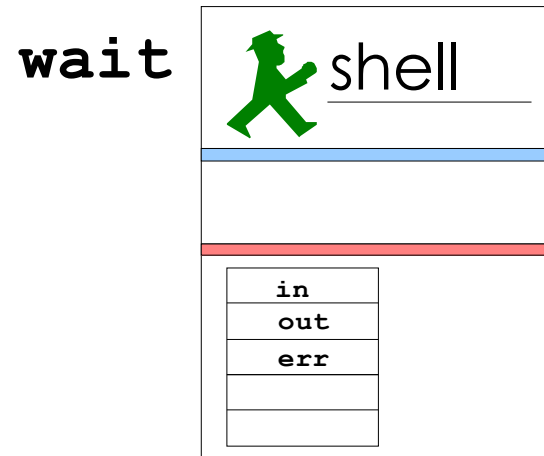
Aufbau einer Filterkette mit Pipes



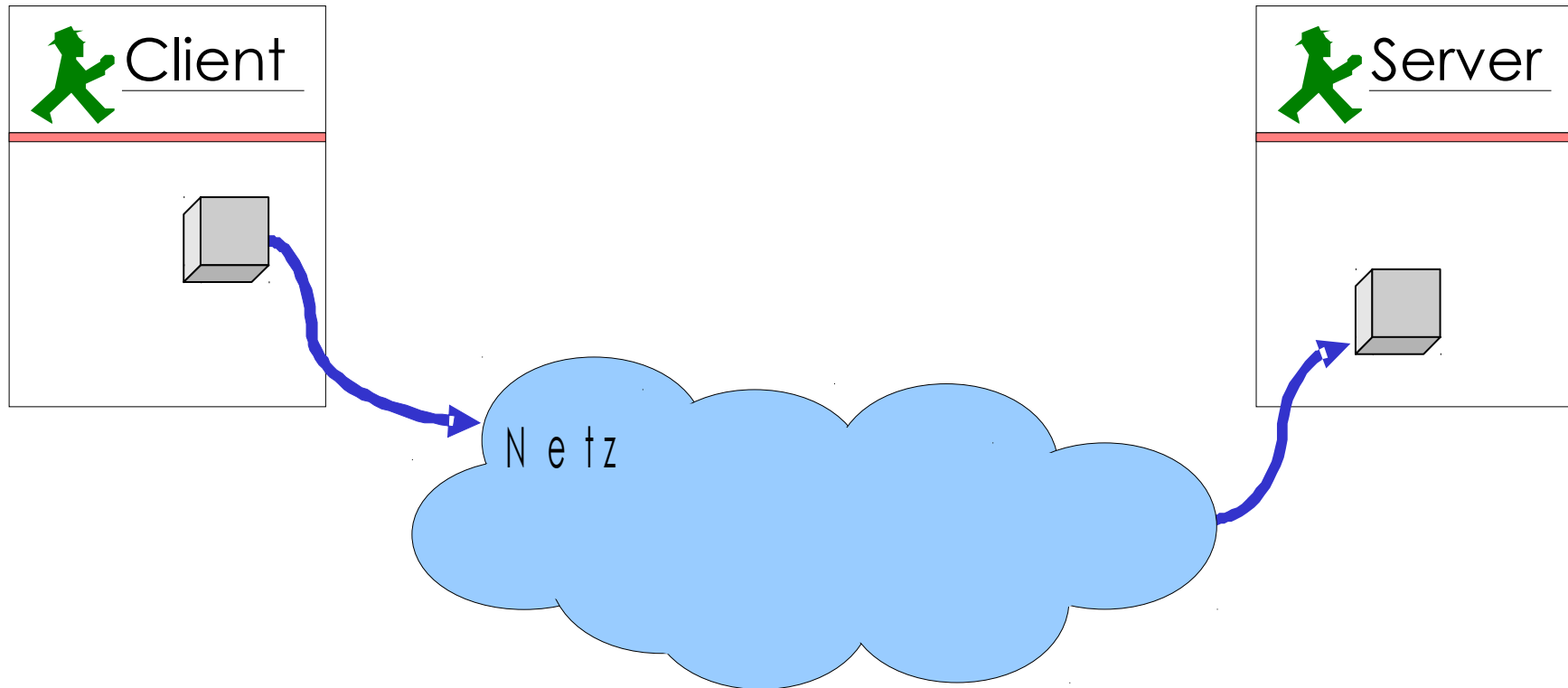
Aufbau einer Filterkette mit Pipes



Aufbau einer Filterkette mit Pipes



Sockets



Client

Server

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

```
create sock(protocol type)
bind (Adresse)
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 /etc/group

```
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
...
```

```
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: Uld

group: Gid

Rangliste.dat		
rw-	r--	---
		others
		group: Schach
		owner: Heini

- 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

owner	group	others
rwx	-wx	--x

↑ ↑ ↑
execute
write
read

Rangliste.dat		
rw-	r--	---
		others
		group: Schach
		owner: Heini

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


	_____
<hr/>	
Uld: Otto	
Gld: stud	

Benutzer und Prozesse

- Jeder Prozess repräsentiert einen Benutzer.

Prozess-Attribute:

- Uld, Gld
- Effective-Uld, Effective-Gld




Uid	: Otto
Gid	: stud
E-Uld	: Otto
E-Gld	: stud

- Nur wenige hochprivilegierte Prozesse dürfen Uld und Gld manipulieren, z.B. Login-Prozess.
- Nach Überprüfung des Passwortes setzt Login-Prozess Uld, Gld, Eff-Uld, Eff-Gld.
- Alle anderen Prozesse: Kinder des Login-Prozesses.
- Kinder erben Attribute von Eltern.

Prozesse und Dateien

Die Attribute E-Uld und E-Gld bestimmen beim Zugriff auf Dateien die Rechte eines Prozesses.


	_____
<hr/>	
Uid	: Otto
Gid	: stud
E-Uld	: Otto
E-Gld	: stud

Aufgabe12.tex		
rw-	r--	---
		Others
	Group : stud	
	Owner : Heini	

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.


	_____
<hr/>	
Uid	: Otto
Gid	: Schach
E-Uid	: Otto
E-Gid	: Schach

Rangliste.dat		
rw-	r?	---
		Others
		Group : Schach
		Owner : Petra

Unix-Lösung: SetUld-Mechanismus

- Datei, die vertrauenswürdigen Programmcode (z. B. Schach) enthält, besitzt Kennzeichnung als „Set-UID“ (**s**).
- Bei **exec** auf Set-Uld Programme erhält ausführender Prozess als Effektive Uld die Uld des Installateurs (Owners) des Programms (genauer: der Datei, die Programm enthält).

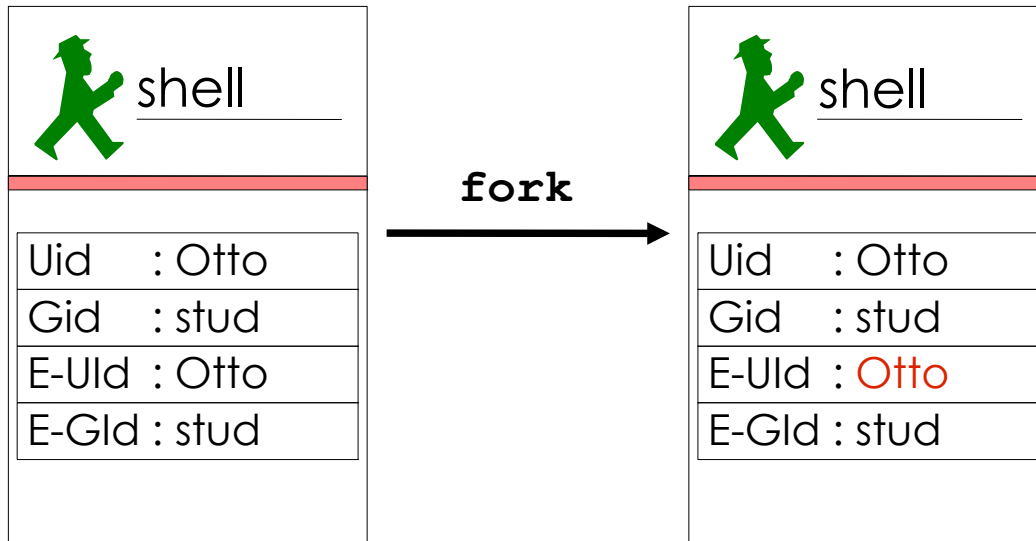
SetUid am Beispiel Rangliste

 shell
Uid : Otto
Gid : stud
E-Uid : Otto
E-Gid : stud

Schach		
--s	--x	---
		Others
		Group : Schach
		Owner : Petra

Rangliste.dat		
rw-	r--	---
		Others
		Group : Schach
		Owner : Petra

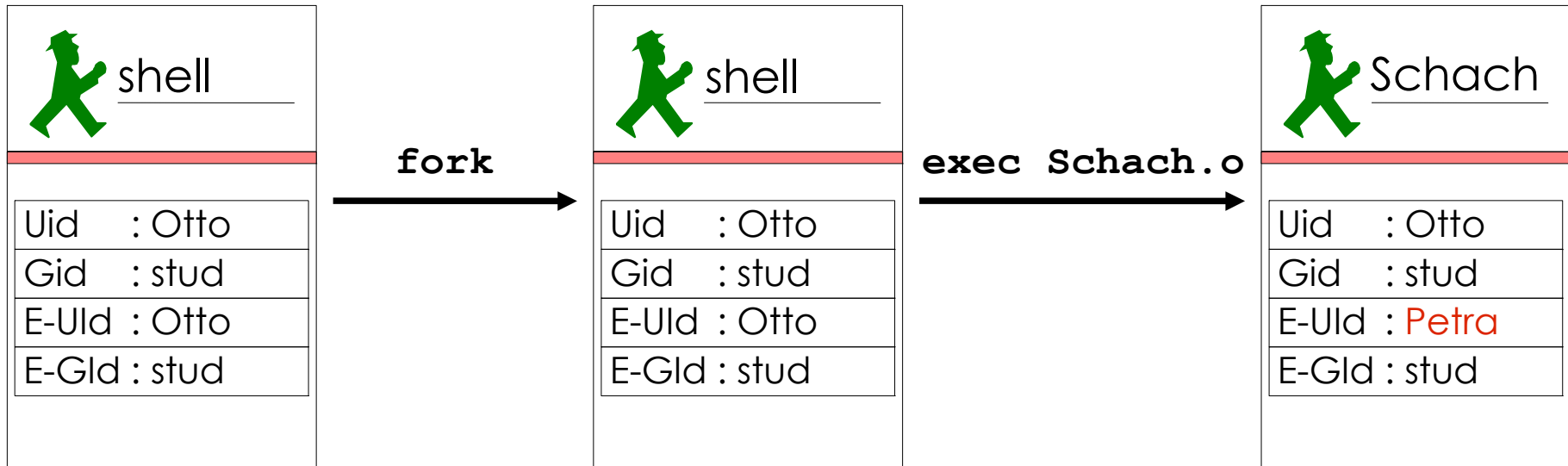
SetUid am Beispiel Rangliste



Schach		
--s	--x	---
		Others
		Group : Schach
		Owner : Petra

Rangliste.dat		
rw-	r--	---
		Others
		Group : Schach
		Owner : Petra

SetUid am Beispiel Rangliste



Schach		
--s	--x	---
		Others
		Group : Schach
		Owner : Petra

Rangliste.dat		
rw-	r--	---
		Others
		Group : Schach
		Owner : Petra

SetUld

- 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

Zusammenfassung/Weiterführung

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

Reimplementierungen/Ableger:

- Linux (Server, ..., eingebettete Systeme)
- Solaris (Server)
- MacOS