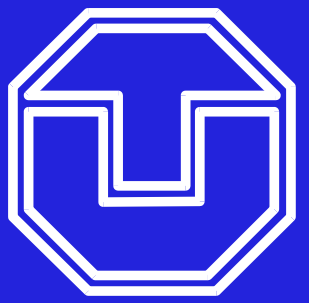
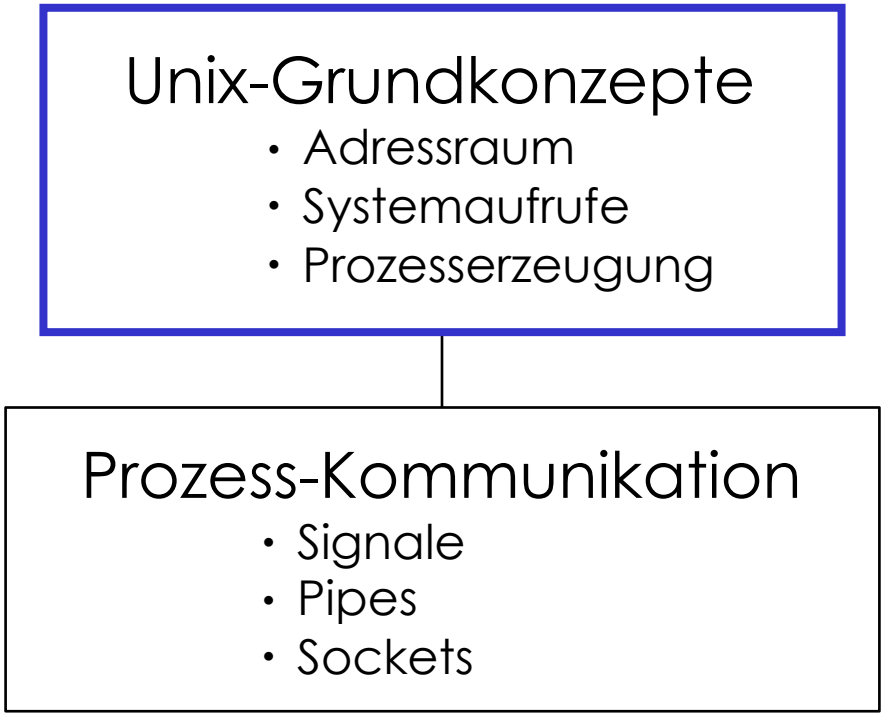
Fallbeispiel Unix

Betriebssysteme

Hermann Härtig
TU Dresden



Wegweiser


Unix-Grundkonzepte
• Adressraum
• Systemaufrufe
• Prozesserzeugung
Prozess-Kommunikation
• Signale
• Pipes
• Sockets

Prozesse

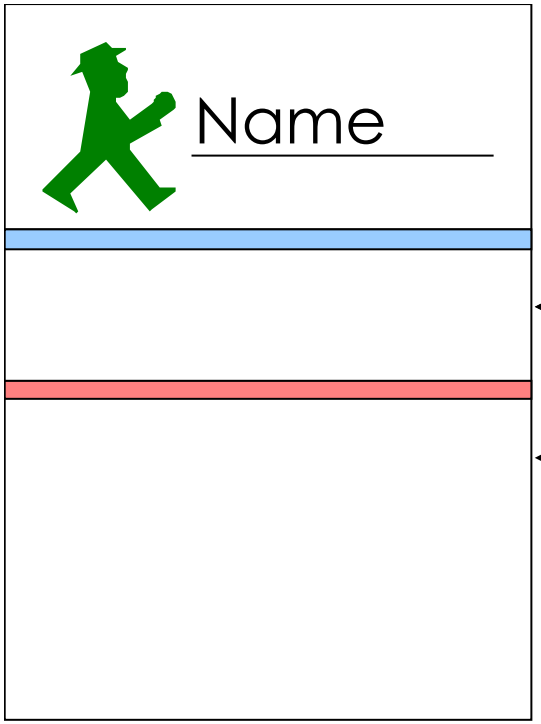
Unix-Prozess

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

Viele Prozesse pro Rechner

- Benutzerprozesse
- Hintergrund-Systemprozesse („daemons“)

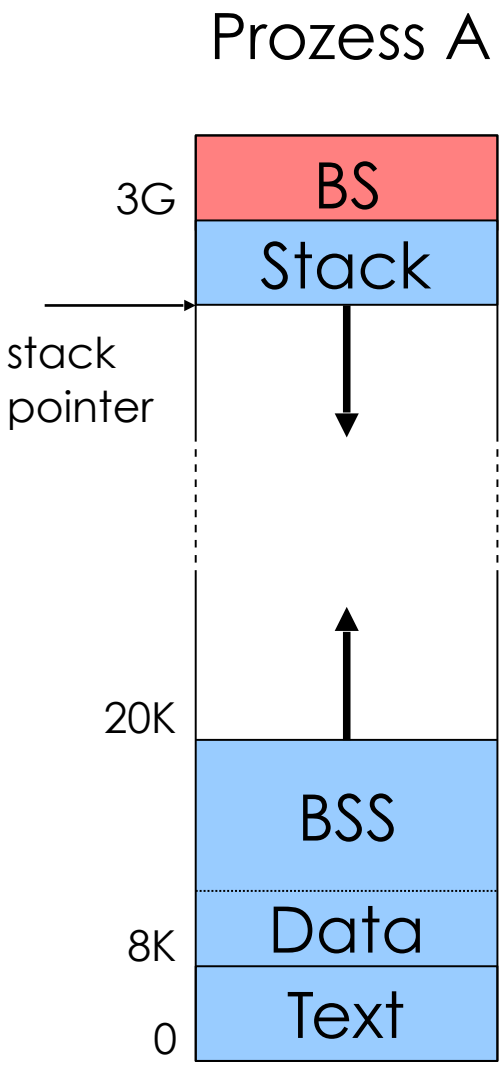
Prozesse



User-Mode Daten

dem Prozess zugeordnete Kern-Strukturen werden nur im Kern manipuliert

Adressraum



Daten-Segment: globale Daten

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

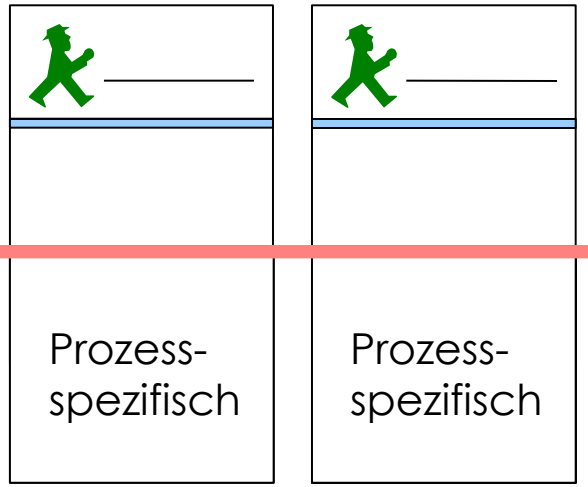
Textsegment: Maschinencode

- read only
- erste Seite frei zum Entdecken nicht-initialisierter Pointer

Stack-Segment: Kellerspeicher

- enthält Funktionsparameter, lokale Variablen und Rücksprung-Adresse

Kern-Adressraum



- weitere Datenstrukturen des Kerns
- z. B. Liste aller Prozesse,
  Tabelle der offenen Dateien
- Kern-Code
- Gerätespeicher
- Zwischenspeicher für Festplatte

Systemaufrufe

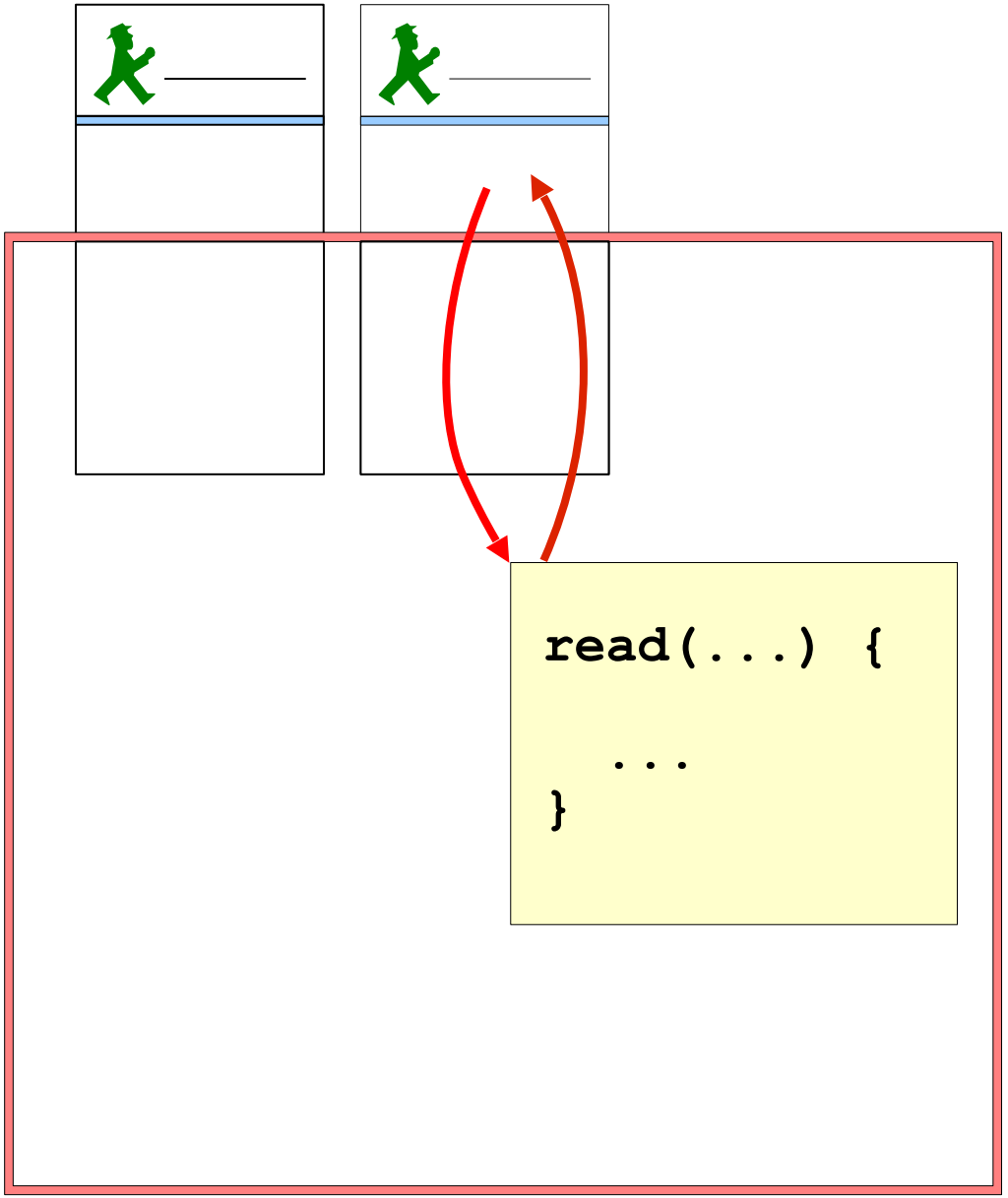
Funktionalitäten des Betriebssystem-Kerns werden über Systemaufrufe (System-Calls) angefordert.

Beispiel

```
status = read (file, buffer, count);
```

- Ergebnis: Anzahl der gelesenen Bytes
- Konvention: -1 → Fehler
- Grund für Fehler: **`errno`**

Schutz des Kerns?

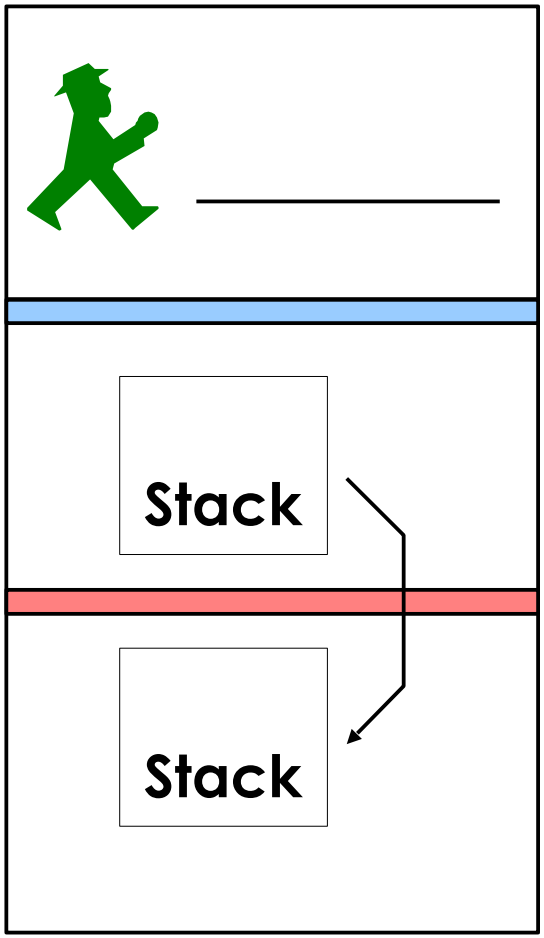


- Prozesse sollen durch getrennte Adressräume voreinander isoliert sein
- gemeinsamer Zugriff auf den Kern würde diesen Schutz untergraben
- Wie werden Kern-Strukturen geschützt?
- Wie wird der Kern sicher aufgerufen?

Prozessor-Modi: usermode/kernelmode

Kernelmode	Usermode
Alle Instruktionen	Teilmenge
User- und Kernel-Mode Speicher	User-Mode Teil des Adressraums
Umschalten des Adressraums	

Ablauf eines Systemcalls



Bei Systemcalls wird

- der Kernmodus eingeschaltet, dadurch wird der Kern-Adressraum sichtbar
- auf den Kern-Stack geschaltet
- an eine feste Einsprungstelle gesprungen und von dort kontrolliert verzweigt
- dann führt der Kern die angeforderte Operation aus

Kernaufruf im Detail

Benutzerprozess

```
read(...) {

  //Parameteraufbereitung
  ...
  call = read;
  int 0x80  // trap
```


Kern

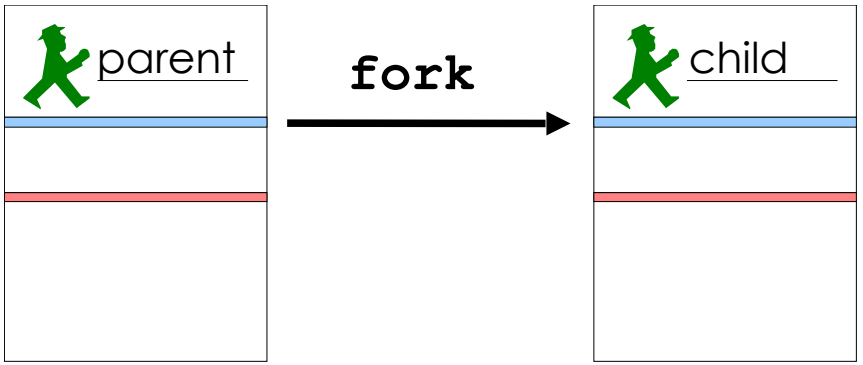
```
//TRAP-Entry
switch (call) {
case read:
  ...
  iret //return from trap
case write: ...
```

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

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

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 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, …)`

- 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);
}
```

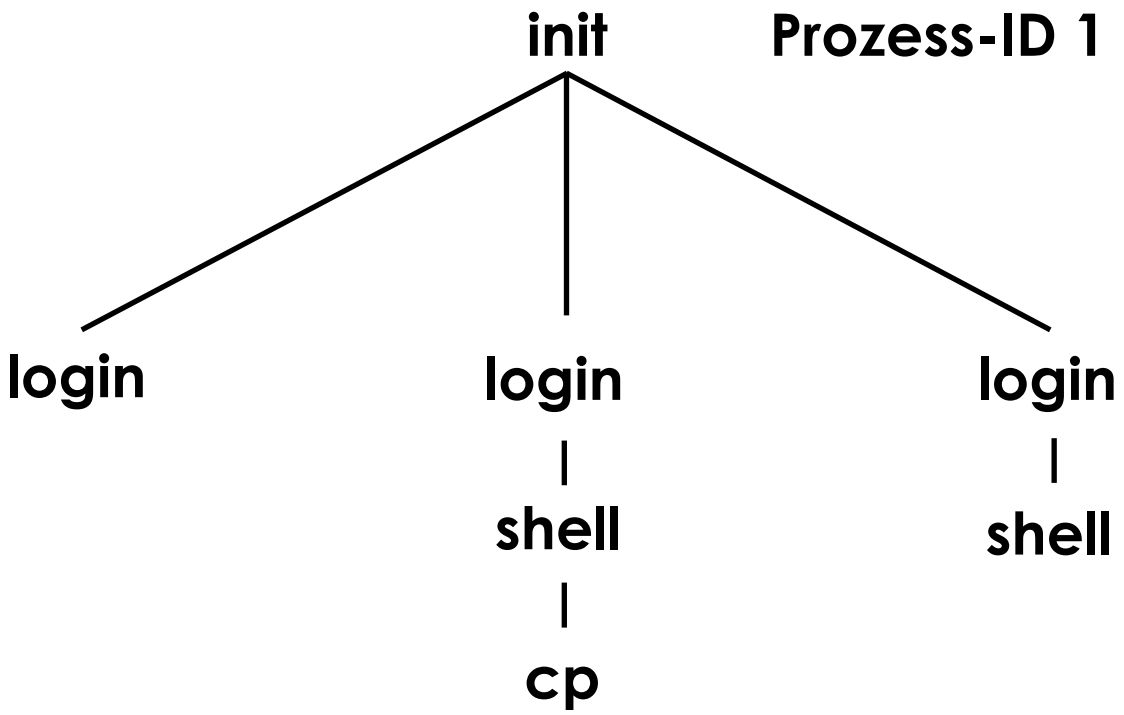
Beispiel: Shell mittels fork/exec

```
read (command, params);

X = fork();




if (X < 0){

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

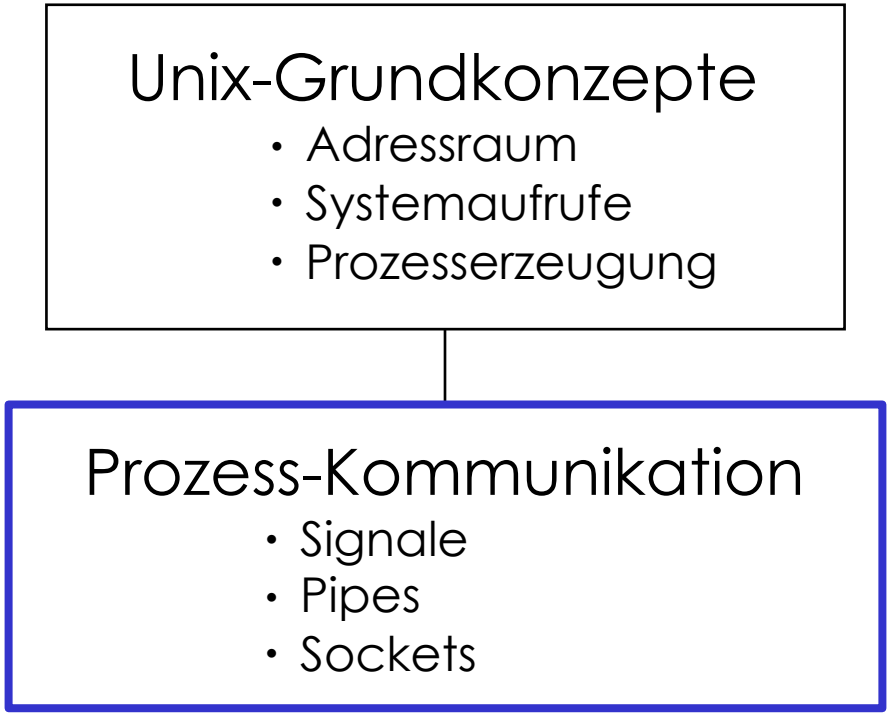
```
// Programm-Code von command

.......




exit
```

Systemstart und Login

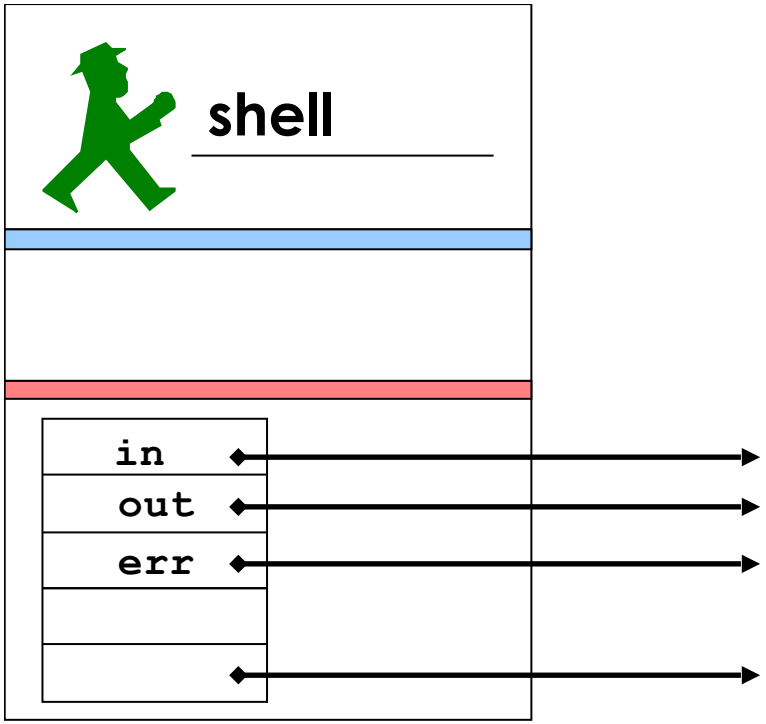
`init` – der erste Prozess



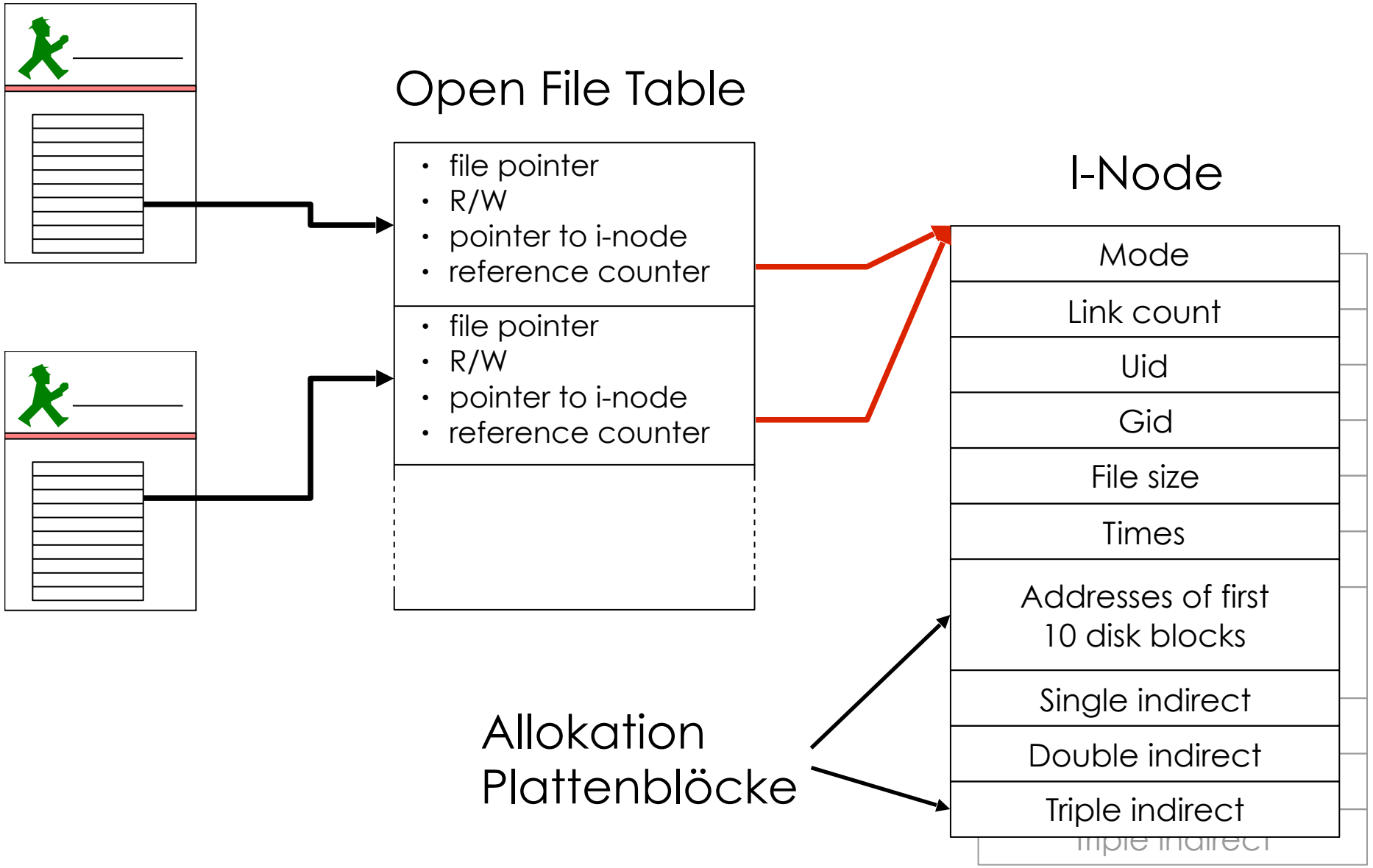
Wegweiser


Unix-Grundkonzepte
• Adressraum
• Systemaufrufe
• Prozesserzeugung
Prozess-Kommunikation
• Signale
• Pipes
• Sockets

Datei-Deskriptoren



Kerninterne Datenstrukturen



Signale

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

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

Signale

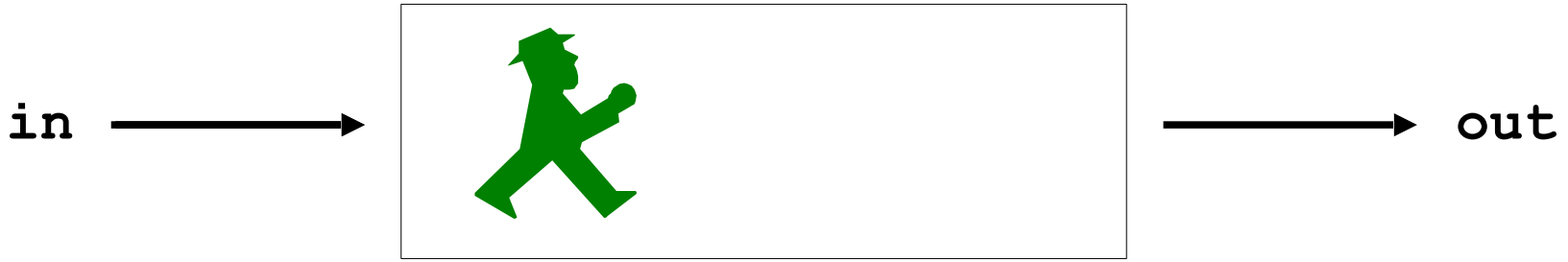
Signal	Cause
SIGABRT	Sent to abort process and force a core dump
SIGALARM	The alarm clock has gone off
SIGFPE	A floating point error has occurred (e.g., division by 0)
SIGHUP	The phone line the process was using has been hung up
SIGILL	The process has executed an illegal machine instruction
SIGINT	The 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 Standard-Eingabe und schreiben nach Standard-Ausgabe
- diese Kanäle belegen stets die ersten drei Datei-Deskriptoren (0, 1, 2)
- **Pipe:** Datenkanal, auf Datei-Deskriptoren abgebildet
- Kein Unterschied, ob Lesen/Schreiben von/in Datei oder über Pipe zu einem anderen Prozess.

```
cat a > b

cat < a > b

cat a | uniq

cat a | sort | uniq
```

Shell-Ebene: Prozesse als Filter



`cat < src.txt | sort`

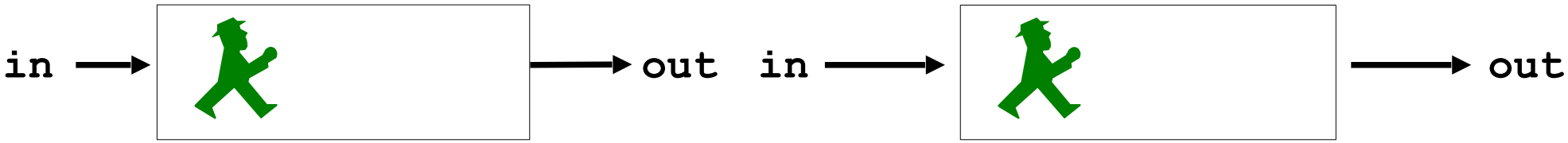
- Datenstrom als durchgängiges Konzept, spezielle Dateien per Konvention (**`stdin`**, **`stdout`**)
- Normalfall:
  - Tastatur als Standard-Eingabe
  - Terminal als Standard-Ausgabe

→ Ein/Ausgabe als Spezialfall von Dateien

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 vorhandene fd's
   close(1); open(out, ...); // durch in/out

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

Shell-Ebene: Prozesse als Filter



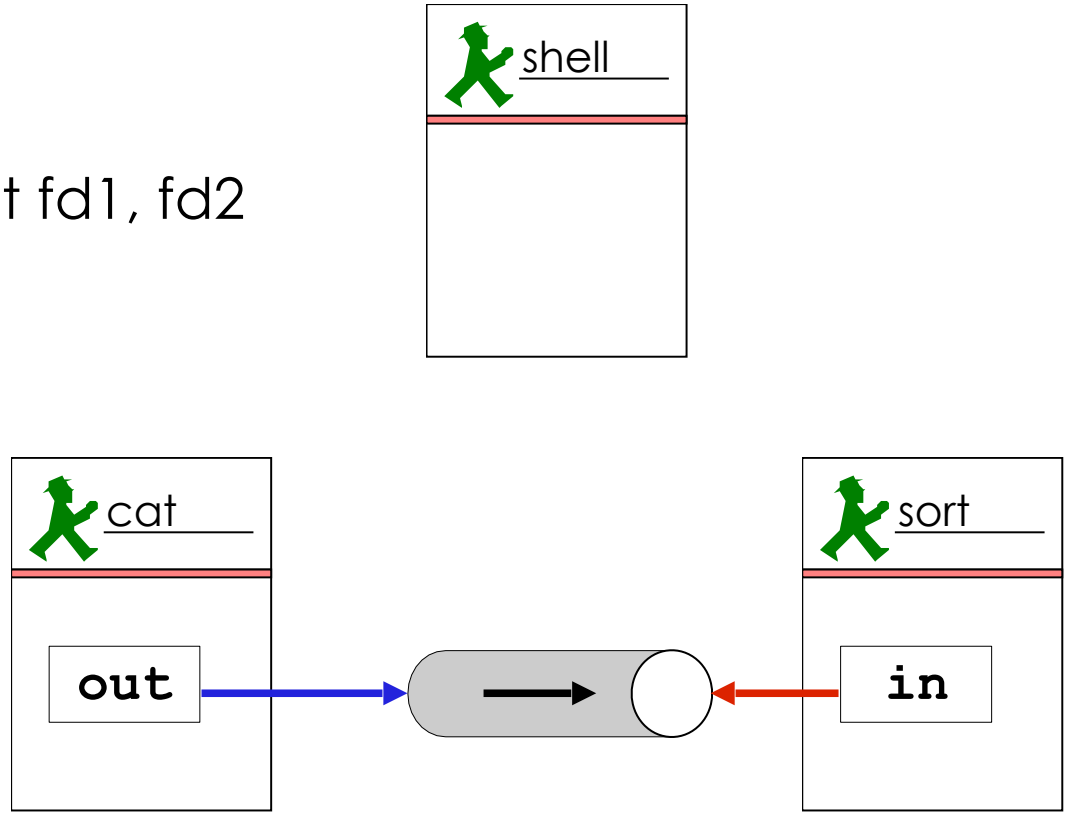

```
cat < src.txt | sort
```

Pipes als Datenstrom-Dateien

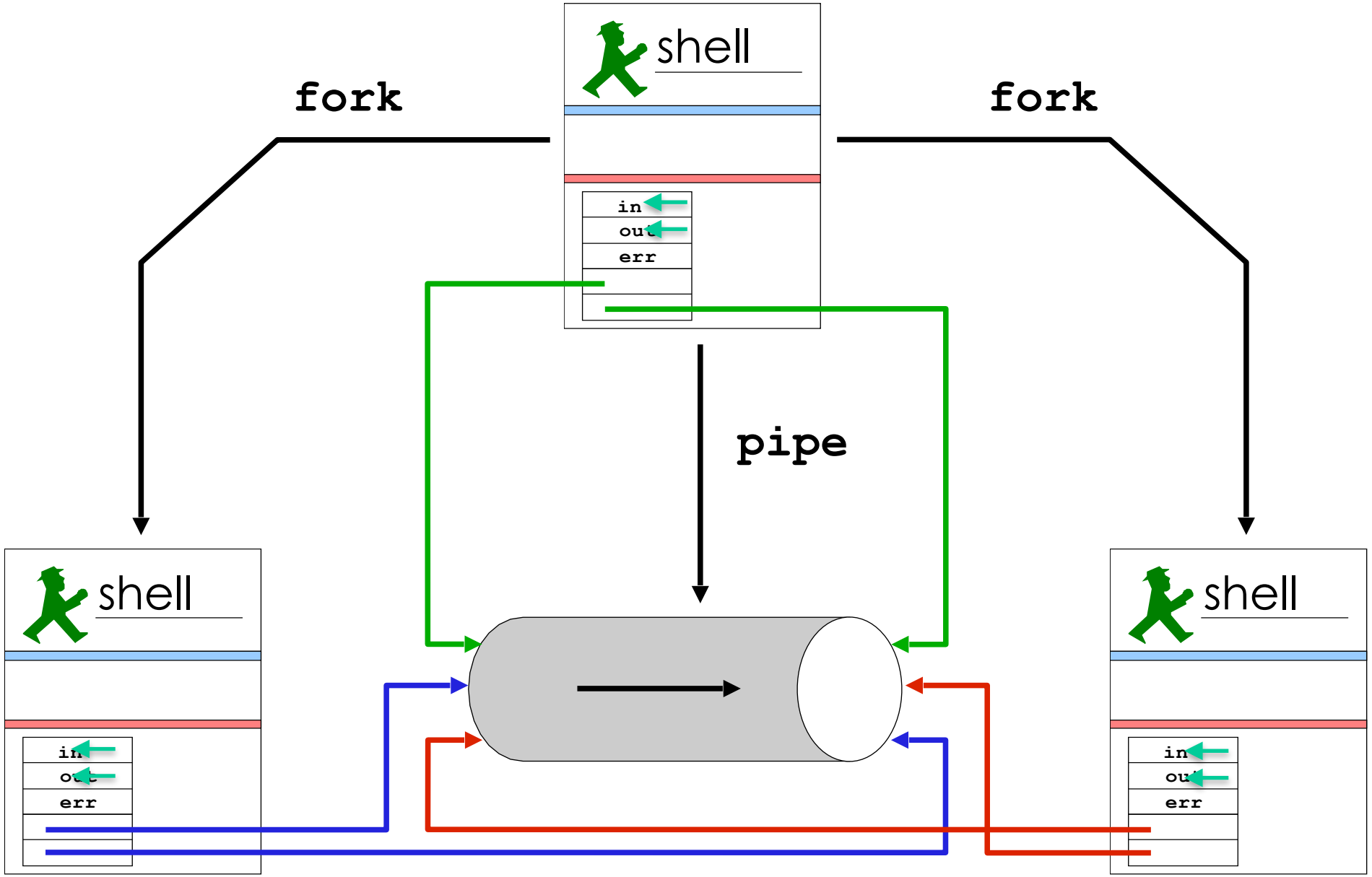
Pipes und Filterketten

z. B. **`cat | sort`**

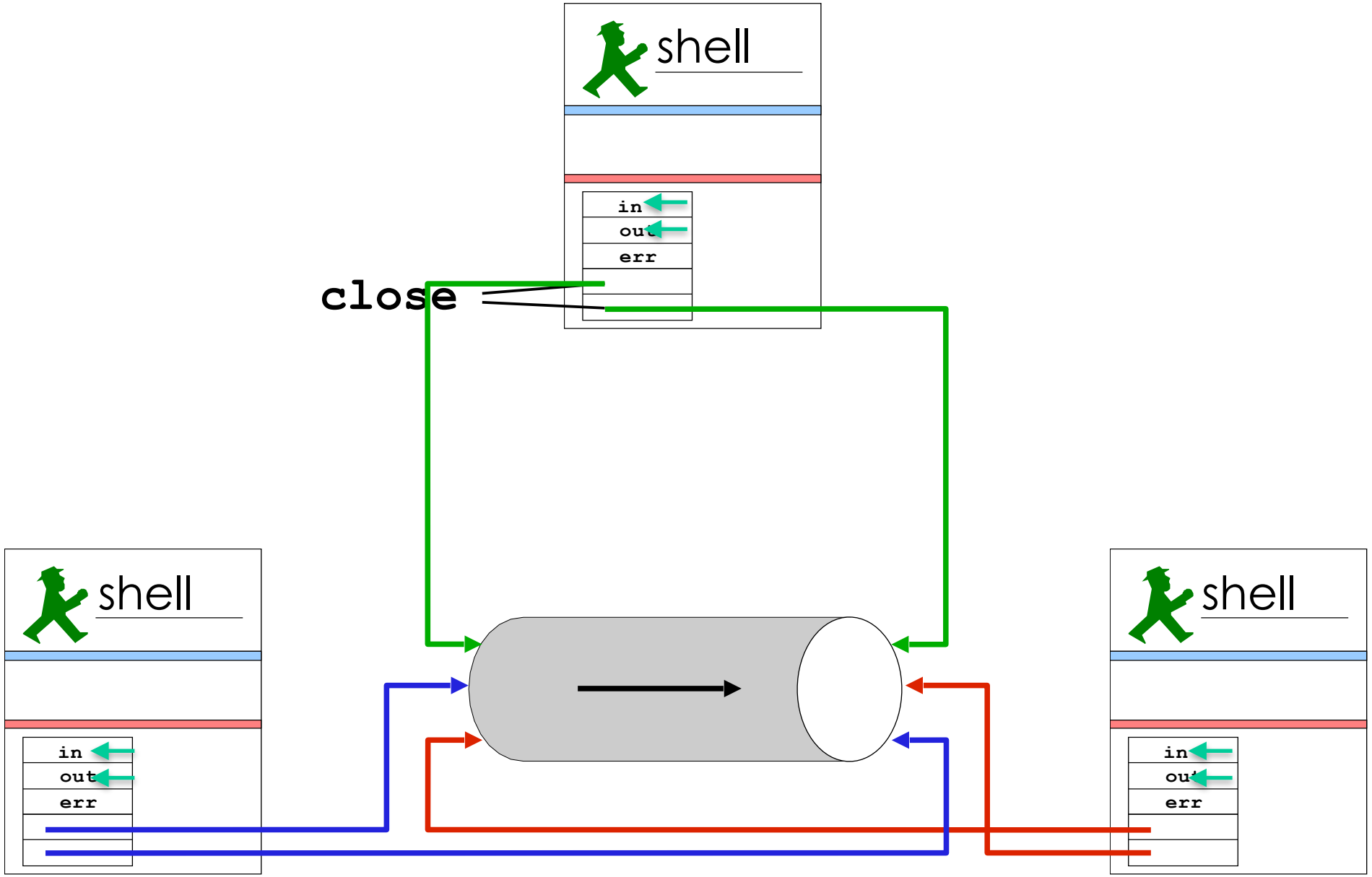
- **`pipe`**: erzeugt pipe mit 2 fd (fd1, fd2)
- **`fork`**: Kind1 für **`cat`**
- **`fork`**: Kind2 für **`sort`**
- Eltern-Prozess (shell): schliesst fd1, fd2
- Kind1:
  - schließt fd2
  - schließt stdout
  - fd1 → stdout
  - schließt fd1
  - **`exec cat`**
- Kind2: spiegelbildlich



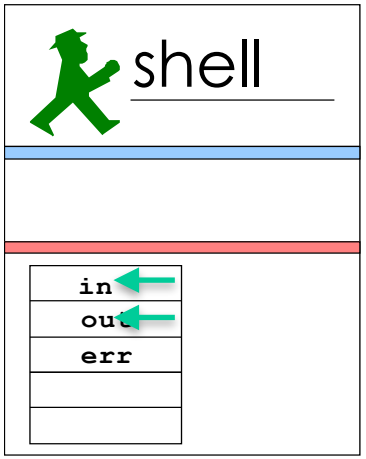
Aufbau einer Filterkette mit Pipes

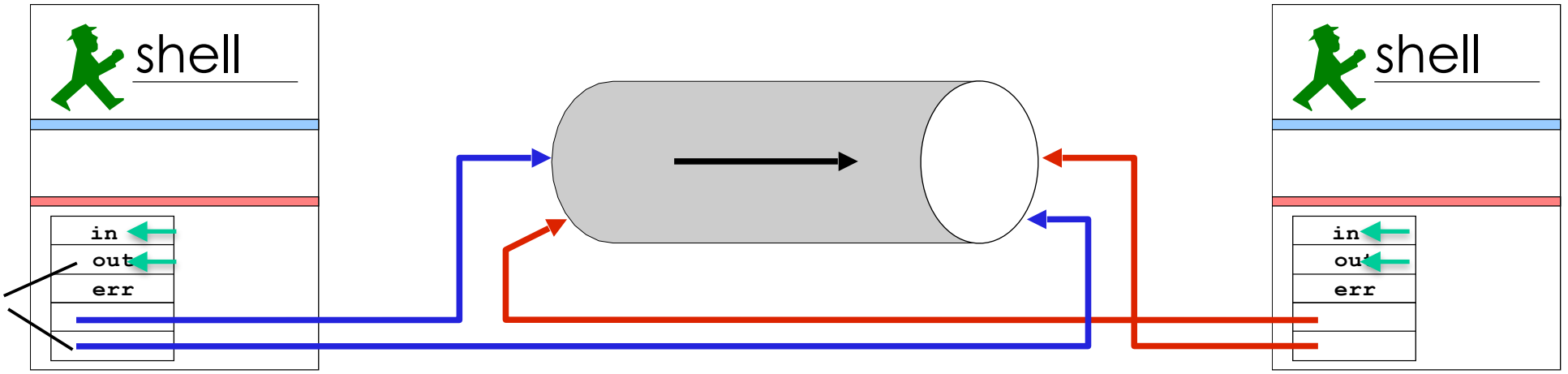


Aufbau einer Filterkette mit Pipes



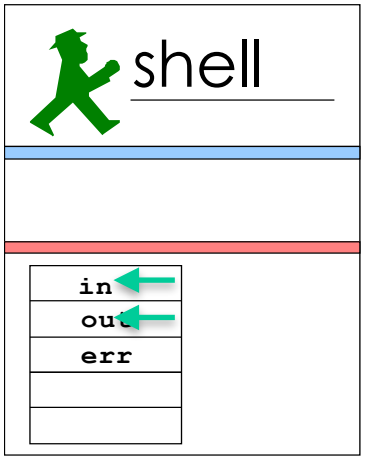
Aufbau einer Filterkette mit Pipes

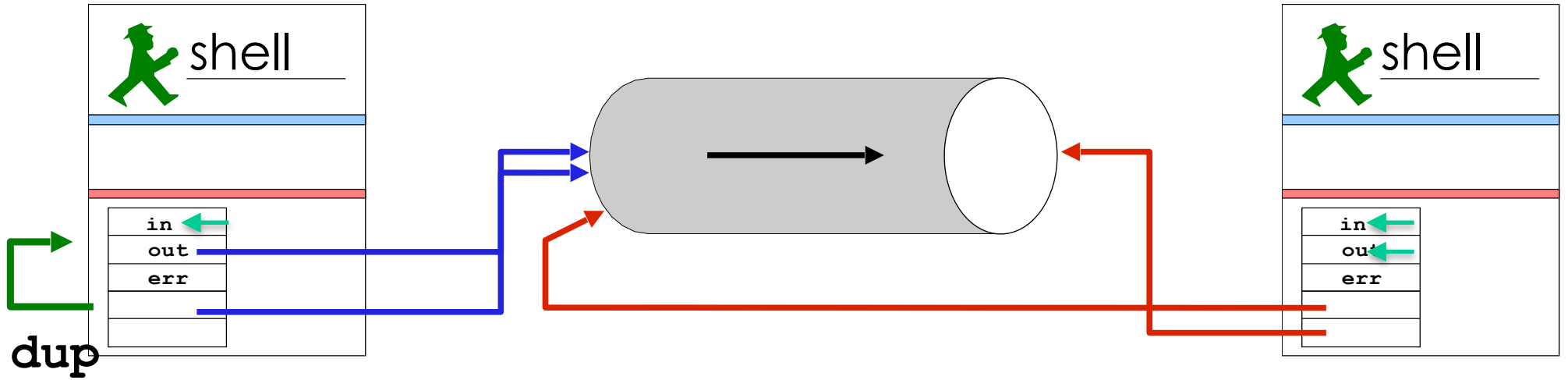




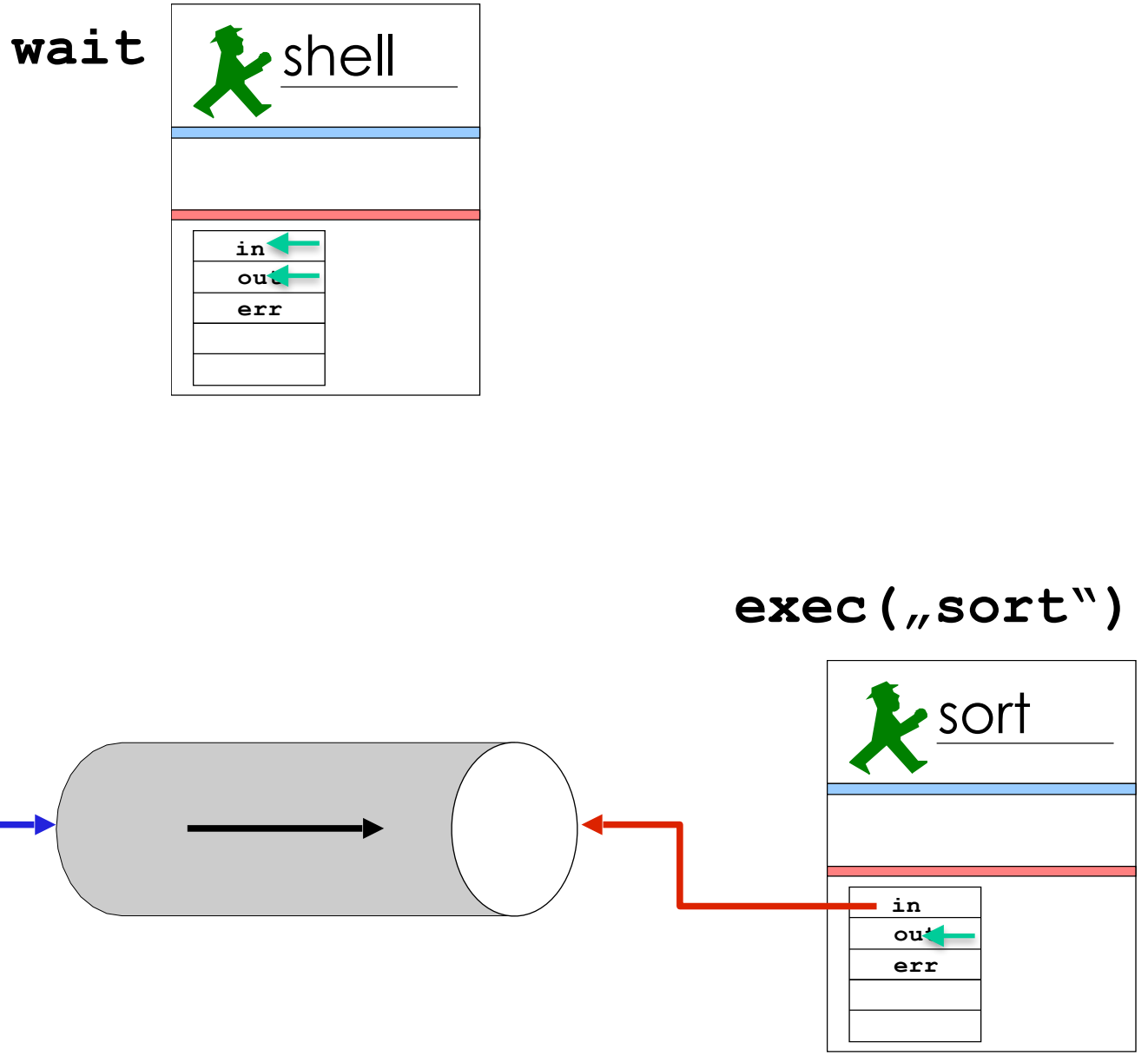
close

Aufbau einer Filterkette mit Pipes





Aufbau einer Filterkette mit Pipes



Sockets

- Zwischen beliebigen Prozessen
  (nicht notwendig “verwandt”)
- Beliebige Protokolle, auch über Rechnergrenzen

Sockets

Client

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

Server

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

Zusammenfassung: Unix

- Erfolgreiches Betriebssystem
  (akademisch, Workstations, Server)
- Wenige, einfache Designprinzipien
- Viele Versionen

Reimplementierungen/Ableger:

- Linux (Server, Desktop, eingebettete Systeme), Android
- Solaris (Server)
- macOS, iOS