Work in progress

# Unix-Prozesse aus Nutzersicht

Ute Bormann, TI2

2023-10-13

# Inhalt

1. Grundlegendes
2. Prozessverwaltung an der Nutzerschnittstelle
3. Effizientes Arbeiten mit der Shell (Bash)

# Teil 1:
# Grundlegendes

# Unix-Prozesse aus Nutzersicht

- Prozess ist „Programm in Ausführung“
- Muss vom Betriebssystem verwaltet werden
  (z.B. Zugriff auf Betriebsmittel)

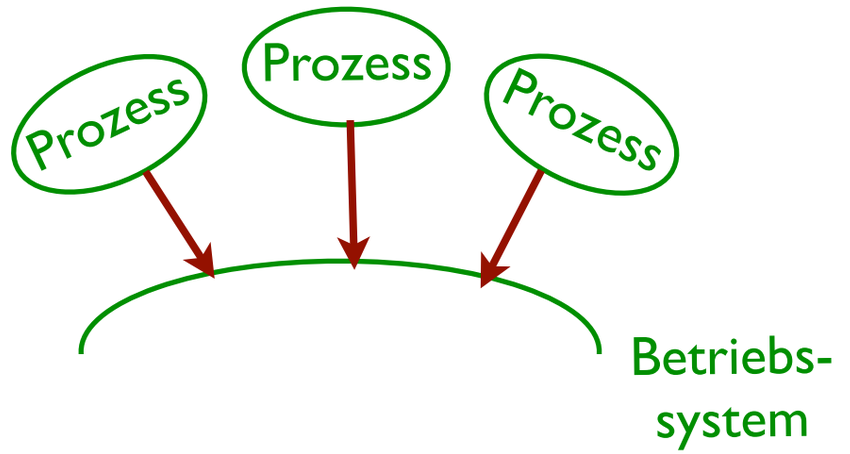
# Unix-Prozesse aus Nutzersicht

- Prozess ist „Programm in Ausführung“
- Muss vom Betriebssystem verwaltet werden
  (z.B. Zugriff auf Betriebsmittel)
- Shell ist ein Prozess
  - wartet auf Benutzereingabe (Kommando)
  - „ruft Kommando auf“

  ⇒ neues Programm in Ausführung

  ⇒ Shell erzeugt dazu neuen Prozess

  (Achtung: cd ändert lediglich Status der Shell)



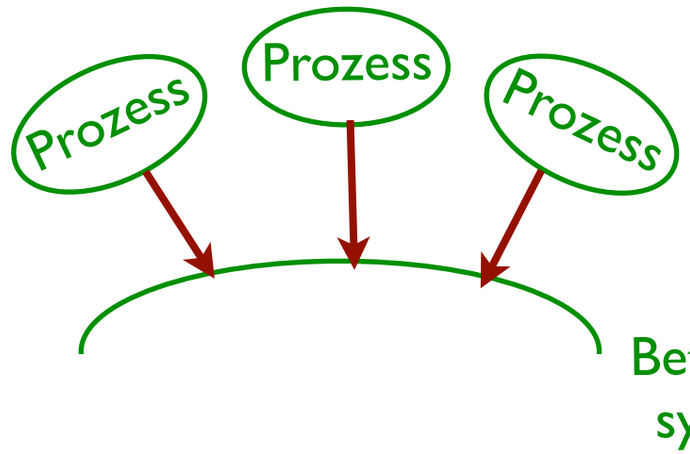
# Unix-Prozesse aus Nutzersicht

- Prozess ist „Programm in Ausführung“
- Muss vom Betriebssystem verwaltet werden
  (z.B. Zugriff auf Betriebsmittel)
- Shell ist ein Prozess
  - wartet auf Benutzereingabe (Kommando)
  - „ruft Kommando auf“

    ⇒ neues Programm in Ausführung

    ⇒ Shell erzeugt dazu neuen Prozess

    (Achtung: cd ändert lediglich Status der Shell)




- Erzeugung von Kindprozessen

  ⇒ Hierarchie

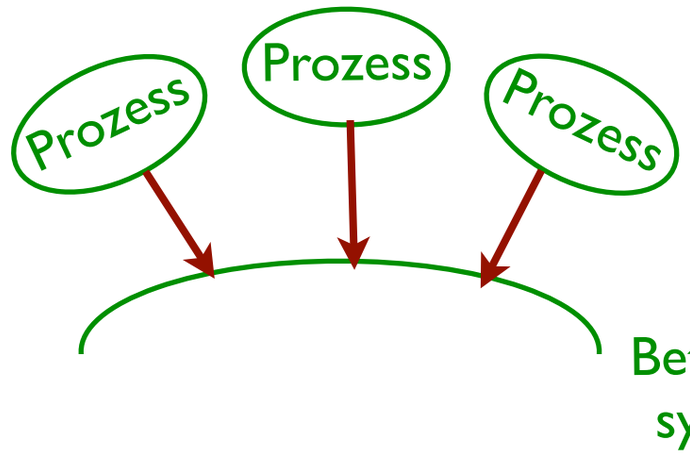
# Unix-Prozesse aus Nutzersicht

- Prozess ist „Programm in Ausführung“
- Muss vom Betriebssystem verwaltet werden
  (z.B. Zugriff auf Betriebsmittel)
- Shell ist ein Prozess
  - wartet auf Benutzereingabe (Kommando)
  - „ruft Kommando auf“

    ⇒ neues Programm in Ausführung

    ⇒ Shell erzeugt dazu neuen Prozess

    (Achtung: cd ändert lediglich Status der Shell)




- Erzeugung von Kindprozessen

  ⇒ Hierarchie

- In diesem Beispiel: Shell hat Kontrolle über „Terminal“ (insbes. Tastatur) an date abgegeben und wartet auf Terminierung von date.

  ⇒ keine nebenläufige Kommandoausführung möglich.

  ⇒ sehr unschön bei langwierigen Kommandos (klassisches Beispiel: cc)

- In diesem Beispiel: Shell hat Kontrolle über „Terminal“ (insbes. Tastatur) an date abgegeben und wartet auf Terminierung von date.

  ⇒ keine nebenläufige Kommandoausführung möglich.

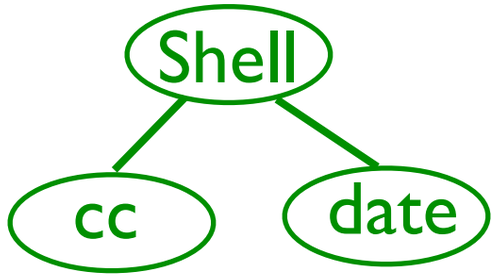
  ⇒ sehr unschön bei langwierigen Kommandos (klassisches Beispiel: cc)

- Nicht alle Kommandos benötigen Kontrolle über das „Terminal“ (insbes. Tastatur)

  Beispiel: cc bla.c (⇒ a.out)

  ⇒ kann man im „Hintergrund“ starten

```
$ cc bla.c &
$ date
Mon Nov 1 ...
```



- In diesem Beispiel: Shell hat Kontrolle über „Terminal“ (insbes. Tastatur) an date abgegeben und wartet auf Terminierung von date.

  ⇒ keine nebenläufige Kommandoausführung möglich.

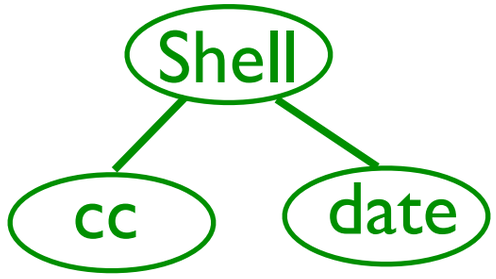
  ⇒ sehr unschön bei langwierigen Kommandos (klassisches Beispiel: cc)

- Nicht alle Kommandos benötigen Kontrolle über das „Terminal“ (insbes. Tastatur)

  Beispiel: cc bla.c (⇒ a.out)

  ⇒ kann man im „Hintergrund“ starten

```
$ cc bla.c &
$ date
Mon Nov 1 ...
```




- Allerdings:

```
$ date &   ⇒ schreibt durcheinander
$ ...
```

```
$ sort &   ⇒ wird „gestoppt“
$ ...
```

- In diesem Beispiel: Shell hat Kontrolle über „Terminal“ (insbes. Tastatur) an date abgegeben und wartet auf Terminierung von date.

  ⇒ keine nebenläufige Kommandoausführung möglich.

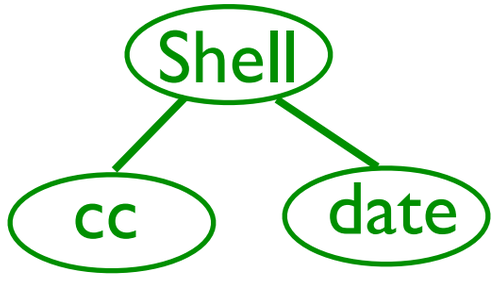
  ⇒ sehr unschön bei langwierigen Kommandos (klassisches Beispiel: cc)

- Nicht alle Kommandos benötigen Kontrolle über das „Terminal“ (insbes. Tastatur)

  Beispiel: cc bla.c (⇒ a.out)

  ⇒ kann man im „Hintergrund“ starten

```
$ cc bla.c &
$ date
Mon Nov 1 ...
```



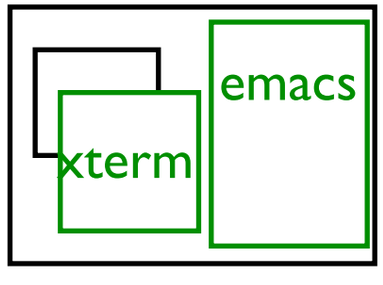

- Allerdings:

```
$ date &  ⇒ schreibt durcheinander
$ ...
```

```
$ sort &  ⇒ wird „gestoppt“
$ ...
```

- Etliche Kommandos erzeugen im „Windowing“-Umfeld neues Fenster

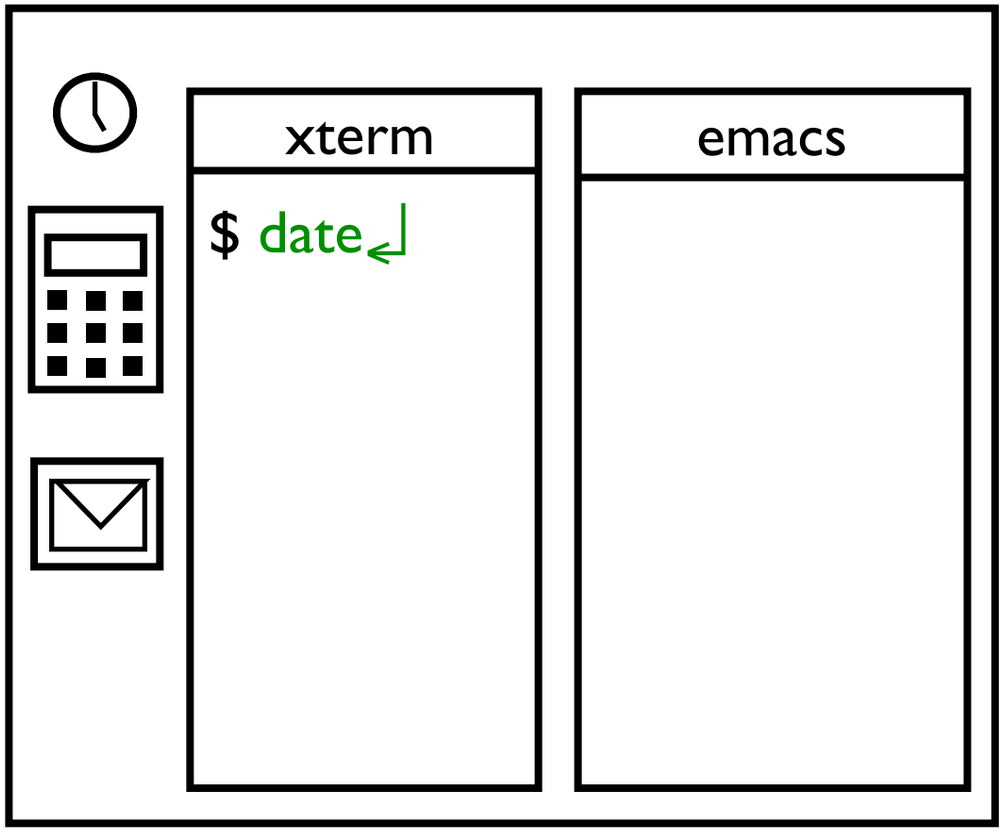
```
$ emacs &
$ xterm &
```

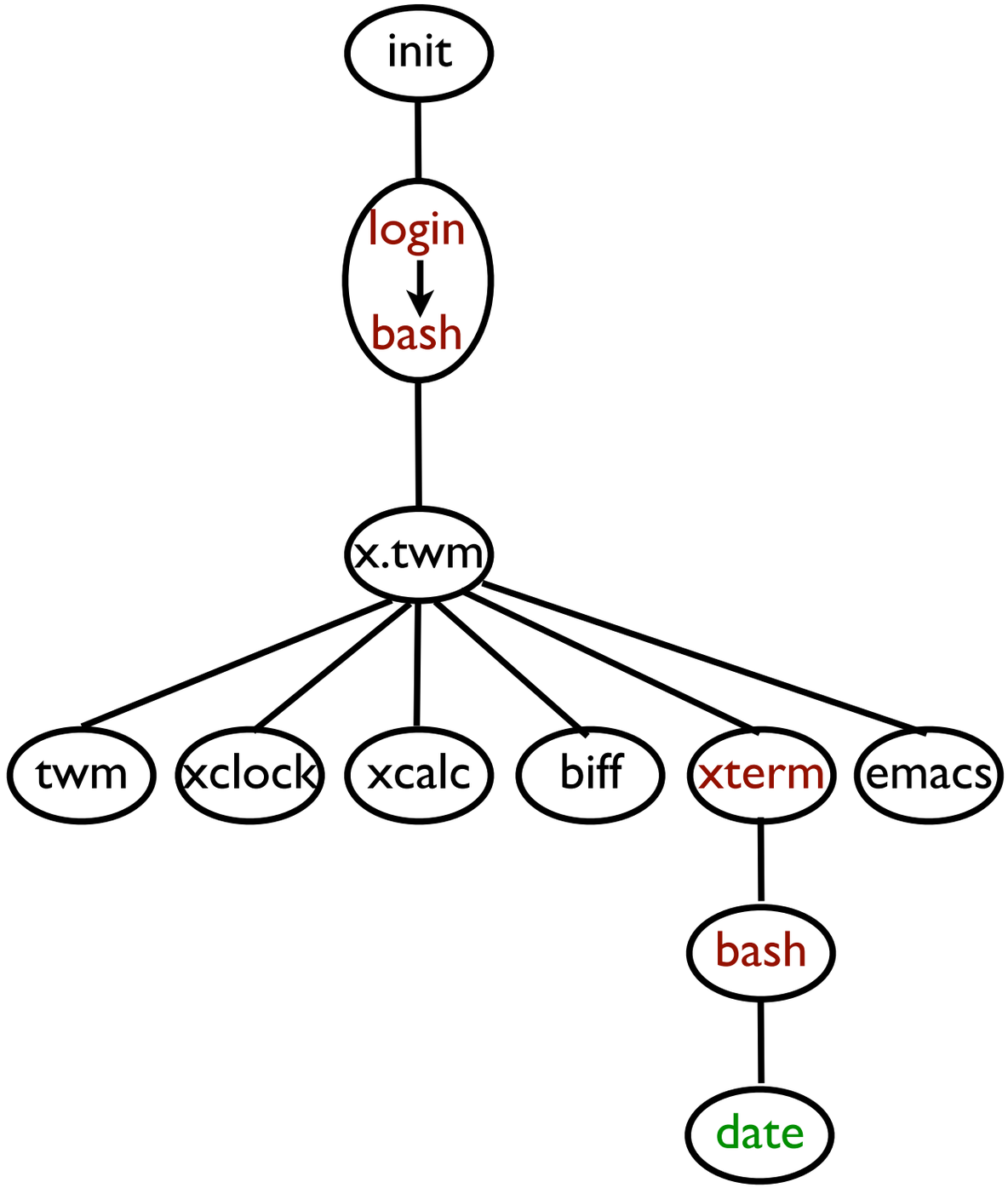



aktives Fenster entscheidet, wer Eingabe bekommt

# Beispiel einer einfachen Prozessstruktur

(vereinfacht)



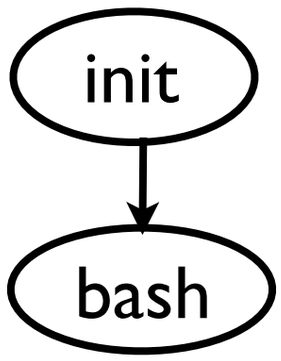



# Fragen – Teil 1

- `bla` sei ein im Pfad ausführbares Programm. Was ist der Unterschied zwischen dem Aufruf `bla` und dem Aufruf `bla &` in der Shell? Welche Auswirkungen hat dies, wenn `bla` von *Standard Input* liest bzw. auf *Standard Output* schreibt?

# Kleine Aufgabe

Gegeben sei der folgende Prozessbaum:




Wie ändert er sich nach Eingeben der folgenden Kommandofolge in der Shell?

```
sleep 1000 &
emacs &
bash
date
```

# Teil 2:
# Prozessverwaltung an der Nutzerschnittstelle

03 Unix-Prozesse aus Nutzersicht, Ute Bormann

# Prozessverwaltung an der Nutzerschnittstelle

- Mehrere Prozesse aktiv

  ⇒ Nutzer muss Kontrolle darüber behalten

- Zwei Situationen (innerhalb eines Shell-Fensters)

# Prozessverwaltung an der Nutzerschnittstelle

- Mehrere Prozesse aktiv

  ⇒ Nutzer muss Kontrolle darüber behalten

- Zwei Situationen (innerhalb eines Shell-Fensters)

  a) Shell im Vordergrund, andere Prozesse im Hintergrund

  ⇒ Shell-Kommandoschnittstelle direkt nutzbar (s. unten)

# Prozessverwaltung an der Nutzerschnittstelle

- Mehrere Prozesse aktiv

  ⇒ Nutzer muss Kontrolle darüber behalten

- Zwei Situationen (innerhalb eines Shell-Fensters)

  a) Shell im Vordergrund, andere Prozesse im Hintergrund

     ⇒ Shell-Kommandoschnittstelle direkt nutzbar (s. unten)

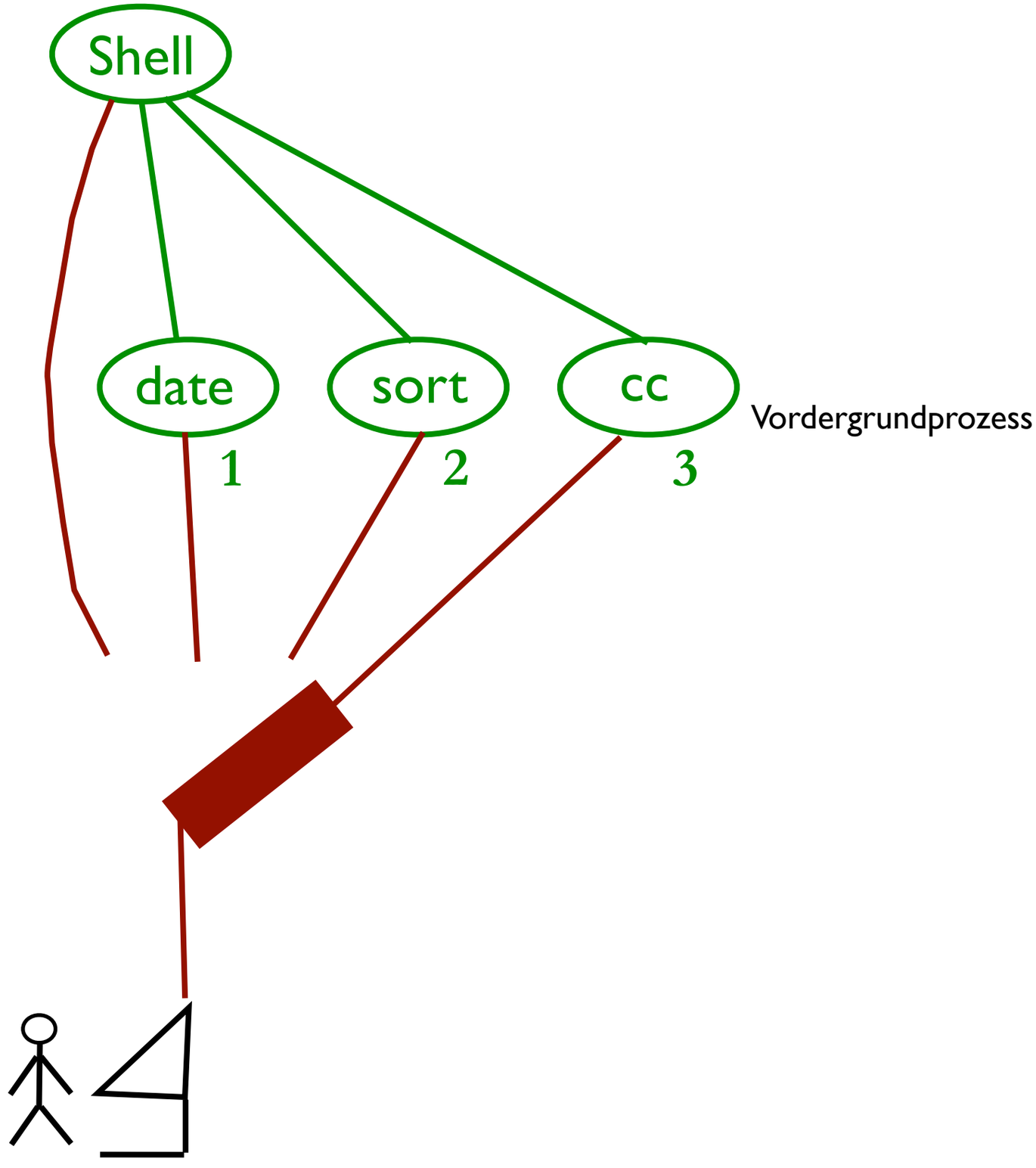
  b) Anderer Vordergrundprozess, Shell inaktiv,
     evtl. weitere Hintergrundprozesse

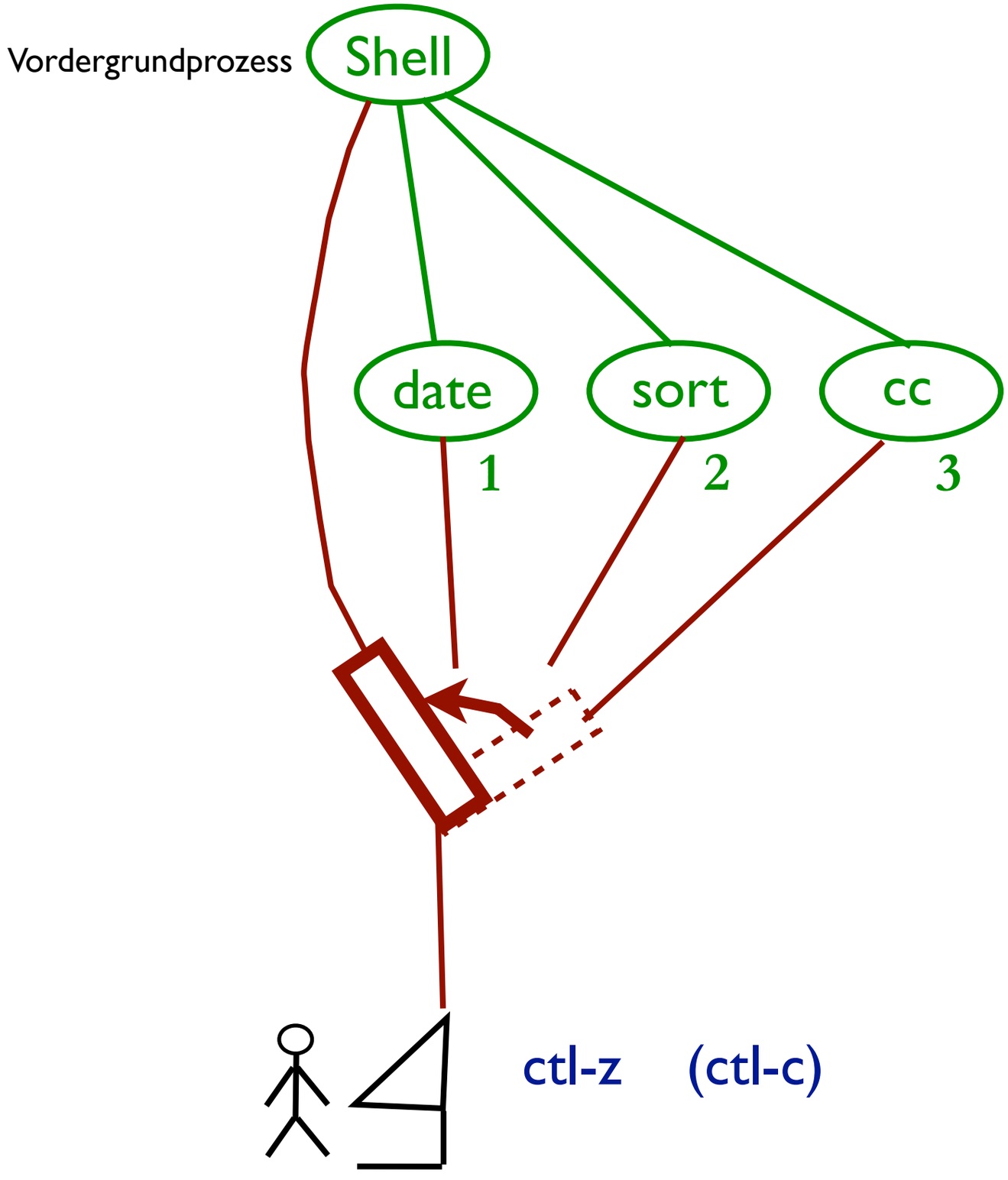
     ⇒ zuvor Vordergrundprozess inaktivieren:

     ctl-z „Stoppen“ (Unterbrechen/Pausieren)

     ctl-c „Abbrechen“ (genaue Semantik hängt von Prozess ab)

     ⇒ dann Shell-Kommandos nutzbar


Shell
date
sort
cc
1
2
3
Vordergrundprozess


Vordergrundprozess
Shell
date
sort
cc
1
2
3
ctl-z
(ctl-c)

- Einige Shell-Kommandos zur Prozesssteuerung:

  `wait` Warten auf Terminierung der Hintergrundprozesse

- Einige Shell-Kommandos zur Prozesssteuerung:

  `wait` Warten auf Terminierung der Hintergrundprozesse

  `fg...` Fortsetzen des angegebenen (DEFAULT: gerade gestoppten) Prozesses im Vordergrund

  `bg...` Fortsetzen des angegebenen (DEFAULT: gerade gestoppten) Prozesses im Hintergrund

  ⇒ setzt Identifikation der Prozesse voraus

  ⇒ Jobnummer (wird bei Start im Hintergrund ausgegeben)

- Einige Shell-Kommandos zur Prozesssteuerung:

  `wait` Warten auf Terminierung der Hintergrundprozesse

  `fg...` Fortsetzen des angegebenen (DEFAULT: gerade gestoppten) Prozesses im Vordergrund

  `bg...` Fortsetzen des angegebenen (DEFAULT: gerade gestoppten) Prozesses im Hintergrund

  ⇒ setzt Identifikation der Prozesse voraus

  ⇒ Jobnummer (wird bei Start im Hintergrund ausgegeben)

- `jobs` (Shell-intern)

```
[1]  Running  emacs
[2]  Running  cc  bla.c
[3]  Running  date
```

- `ps` (Unix-Kommando)

```
PID    TTY/Fenster  CPU-Zeit   Kommando
4460   console      0:45       emacs
4461   console      0:20       cc bla.c
4462   console      0:01       date
4463   console      0:01       ps
```

(Prozess-ID (PID) wird bei Start im Hintergrund ausgegeben)

⇒ Wichtigste Informationen über Prozesse des aktuellen Fensters

- `ps` (Unix-Kommando)

```
PID    TTY/Fenster  CPU-Zeit   Kommando
4460   console      0:45       emacs
4461   console      0:20       cc bla.c
4462   console      0:01       date
4463   console      0:01       ps
```

(Prozess-ID (PID) wird bei Start im Hintergrund ausgegeben)

⇒ Wichtigste Informationen über Prozesse des aktuellen Fensters

- Varianten:

a) Mehr Prozesse anzeigen:

`ps -a` Alle Prozesse, die einem „Terminal“ (allen Fenstern des aktuellen Arbeitsplatzes) zugeordnet sind

`ps -e` Alle Prozesse des Rechners

- `ps` (Unix-Kommando)

```
PID    TTY/Fenster  CPU-Zeit   Kommando
4460   console      0:45       emacs
4461   console      0:20       cc bla.c
4462   console      0:01       date
4463   console      0:01       ps
```

(Prozess-ID (PID) wird bei Start im Hintergrund ausgegeben)

⇒ Wichtigste Informationen über Prozesse des aktuellen Fensters

- Varianten:

a) Mehr Prozesse anzeigen:

`ps -a` Alle Prozesse, die einem „Terminal“ (allen Fenstern des aktuellen Arbeitsplatzes) zugeordnet sind

`ps -e` Alle Prozesse des Rechners

b) Mehr Informationen anzeigen

`ps -f` Auch: Besitzer, Startzeit, Vater-Prozess (PPID) ⇒ Prozesshierarchie ermittelbar

`ps -l` Auch noch: Zustand, Priorität, Größe, Adresse, ... (BS-interne Informationen)

- Auch kombinierbar: z.B. `ps -af`

- Auf Jobnummern bzw. PIDs kann bei Prozesssteuerung zugegriffen werden:

  `fg` `jobnummer` Prozess in Vordergrund holen

- Auf Jobnummern bzw. PIDs kann bei Prozesssteuerung zugegriffen werden:

| | |
|---|---|
| `fg jobnummer` | Prozess in Vordergrund holen |
| `kill -STOP %jobnummer` | Prozess Stop-Signal senden (s. unten) |
| `kill -KILL PID` | Prozess Kill-Signal senden |

- Auf Jobnummern bzw. PIDs kann bei Prozesssteuerung zugegriffen werden:

| | |
|---|---|
| `fg jobnummer` | Prozess in Vordergrund holen |
| `kill -STOP %jobnummer` | Prozess Stop-Signal senden (s. unten) |
| `kill -KILL PID` | Prozess Kill-Signal senden |

## Signale

Meldung (einer Ausnahmesituation) an einen Prozess

- Prozess ⇒ Prozess (später)
- Nutzer ⇒ Prozess
  - an Hintergrundprozess: Shell-Kommando kill

- Auf Jobnummern bzw. PIDs kann bei Prozesssteuerung zugegriffen werden:

| | |
|---|---|
| `fg jobnummer` | Prozess in Vordergrund holen |
| `kill -STOP %jobnummer` | Prozess Stop-Signal senden (s. unten) |
| `kill -KILL PID` | Prozess Kill-Signal senden |

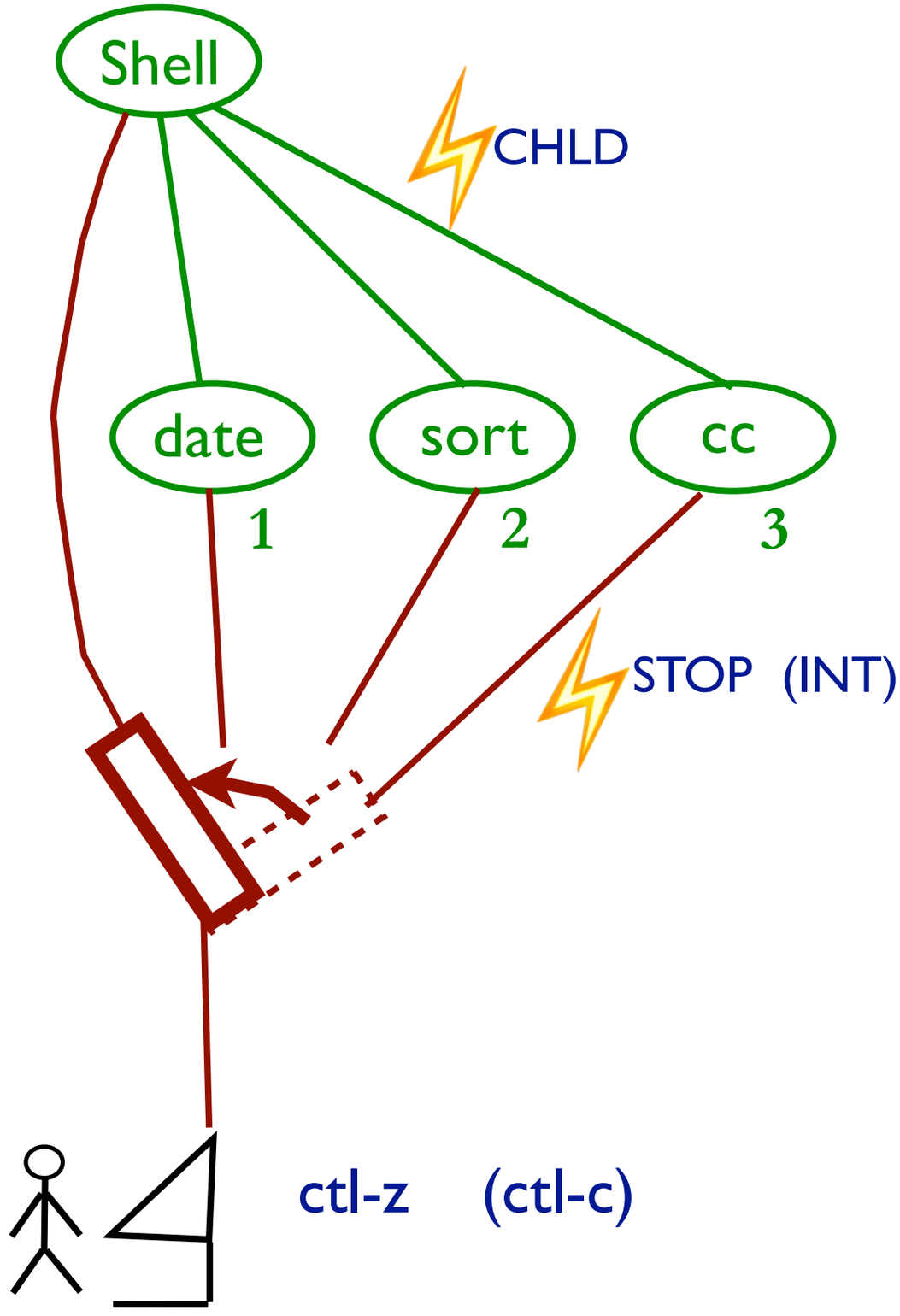
## Signale

Meldung (einer Ausnahmesituation) an einen Prozess

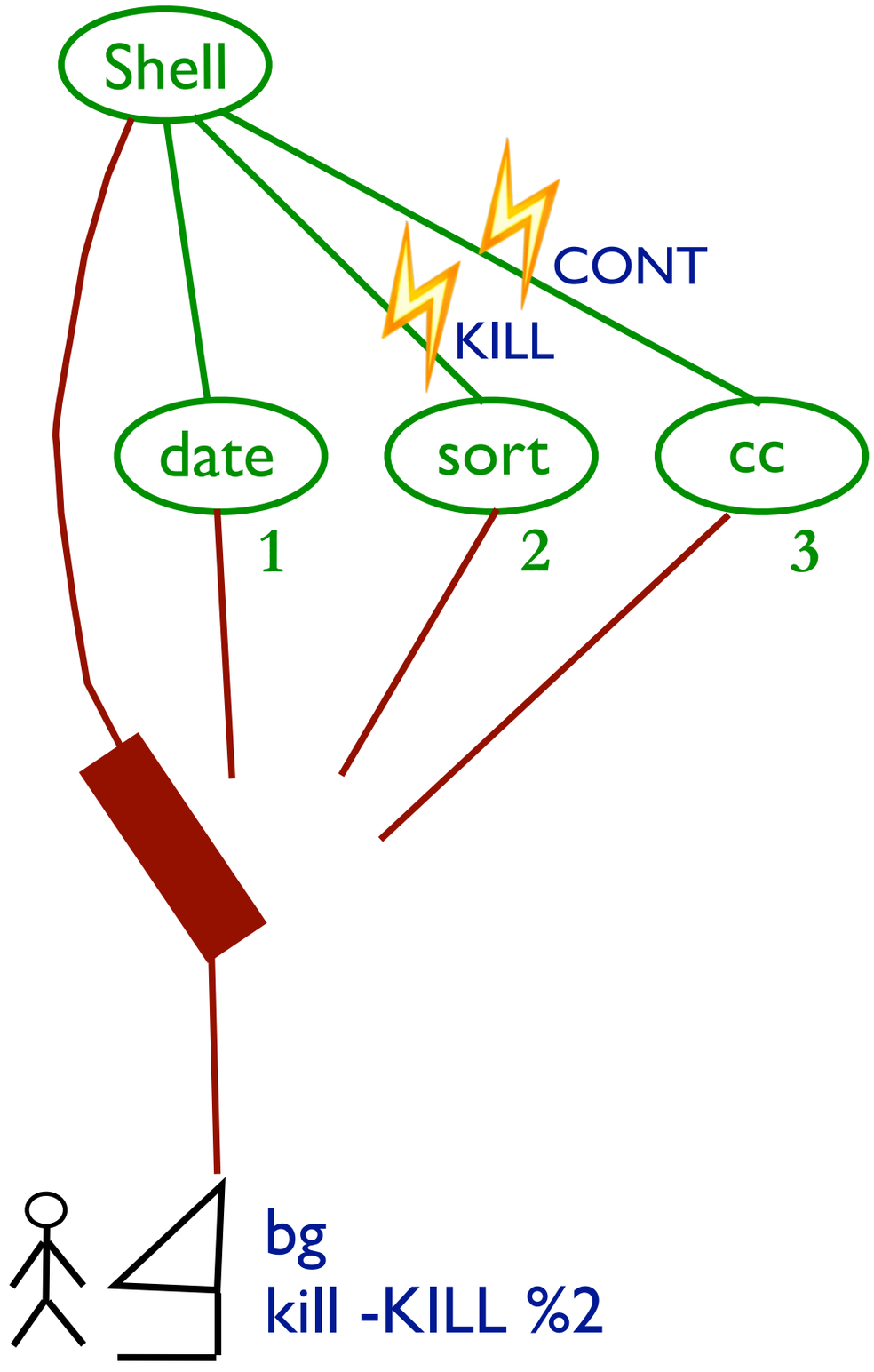
- Prozess ⇒ Prozess (später)
- Nutzer ⇒ Prozess
  - an Hintergrundprozess: Shell-Kommando kill
  - an Vordergrundprozess: ctl-...

    ⇒ i.d.R. Unterscheidung:
    - „normale“ Zeichenfolge eingeben: direkt an Prozess weiterleiten
    - ctl-... eingeben: vom Betriebssystem abgefangen ⇒ Signal an Prozess

# Signale



# Signale



# Interprozesskommunikation

- Sonderfall: Senden von Signalen

  ⇒ Meldung von Ausnahmesituation; veranlasst empfangenden Prozess i.d.R. zum Handeln (z.B. Abbruch, Weitermachen, ...)

# Interprozesskommunikation

- Sonderfall: Senden von Signalen

  ⇒ Meldung von Ausnahmesituation; veranlasst empfangenden Prozess i.d.R. zum Handeln (z.B. Abbruch, Weitermachen, ...)

- Mit Ausgabe eines Kommandos soll weitergearbeitet werden:

  a) 
```
$ date
Mon Nov 1...
$ lprx
  Mon Nov 1...
```

⇒ Wiedereintippen oder Copy&Paste :-(

# Interprozesskommunikation

- Sonderfall: Senden von Signalen

  ⇒ Meldung von Ausnahmesituation; veranlasst empfangenden Prozess i.d.R. zum Handeln (z.B. Abbruch, Weitermachen, ...)

- Mit Ausgabe eines Kommandos soll weitergearbeitet werden:

  a)
  ```
  $ date
  Mon Nov 1...
  $ lprx
  Mon Nov 1...
  ```
  ⇒ Wiedereintippen oder Copy&Paste :-(

  b)
  ```
  $ date > blub
  $ lprx < blub
  ```
  ⇒ Zwischendatei nutzen :-|

# Interprozesskommunikation

- Sonderfall: Senden von Signalen

  ⇒ Meldung von Ausnahmesituation; veranlasst empfangenden Prozess i.d.R. zum Handeln (z.B. Abbruch, Weitermachen, ...)

- Mit Ausgabe eines Kommandos soll weitergearbeitet werden:

  a) `$ date`
  `Mon Nov 1...`
  `$ lprx`
  `Mon Nov 1...`

  ⇒ Wiedereintippen oder Copy&Paste :-(

  b) `$ date > blub`
  `$ lprx < blub`

  ⇒ Zwischendatei nutzen :-|

  c) `$ date | lprx`

  ⇒ Ausgabe direkt in zweites Kommando schieben :-)

  ⇒ Pipes

# Pipes

- Sequentielles Schreiben/Lesen
- Lesen kann schon begonnen werden, obwohl Schreiben noch nicht fertig


- Auch komplexere Szenarien möglich:

```
cat fn1 fn2 | sort > fn3
```

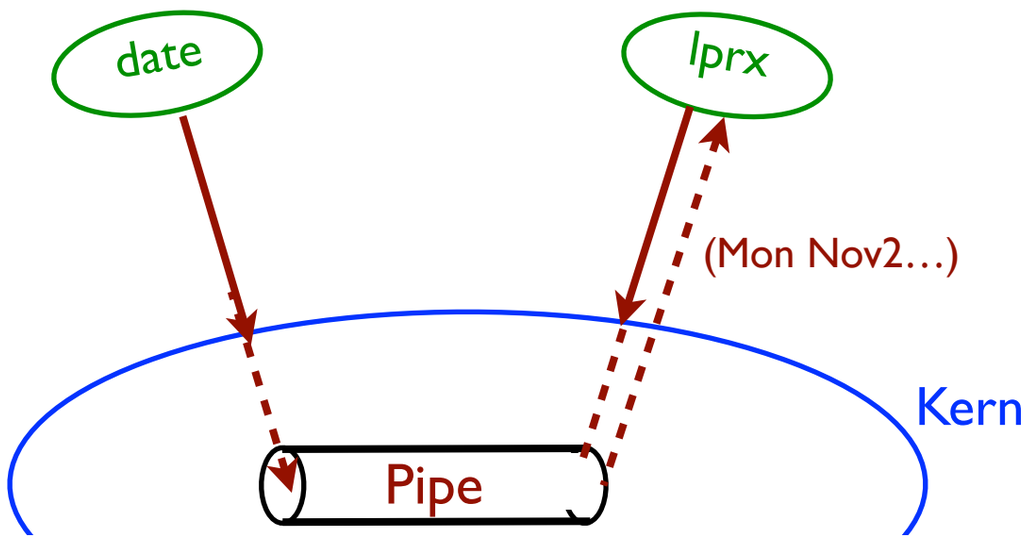
# Pipes

- Sequentielles Schreiben/Lesen
- Lesen kann schon begonnen werden, obwohl Schreiben noch nicht fertig


- Auch komplexere Szenarien möglich:

```
cat fn1 fn2 | sort > fn3
```

- Verwaltung erfolgt im Kern:



# Kleine Aufgabe

Was bewirkt die folgende Kommandozeileneingabe in einem Shell-Fenster?

```
a < b | c | d > e
```

# Fragen – Teil 2

- Wie kann ein Nutzer herausfinden, welche Prozesse in einem Unix-System gerade existieren?
- Was ist eine *Pipe*?

# Teil 3:
# Effizientes Arbeiten
# mit der Shell (Bash)

03 Unix-Prozesse aus Nutzersicht, Ute Bormann

# Effizientes Arbeiten mit der Shell (Bash)

## History

- Bash merkt sich letzte Kommandos

```
$ history
1 cd ~ute
2 ls
3 cat bla | lpr
4 date
```

Datei .bash_history in home directory bei exit
(Ähnliches auch bei anderen Shells…)

- können (verändert) wieder aktiviert werden
- auch per Copy&Paste
- ↑↓ Zeilenweises Durchsuchen der History

  ⇒ Jeweiliges Kommando wird angezeigt

- Emacs Line Editing zum Verändern (→, ←, ctl-a, ctl-e, Zeichen einfügen,...)
- Mit ↵ erneut abschicken

- Alternative: Direktes Angeben des (veränderten) Kommandos

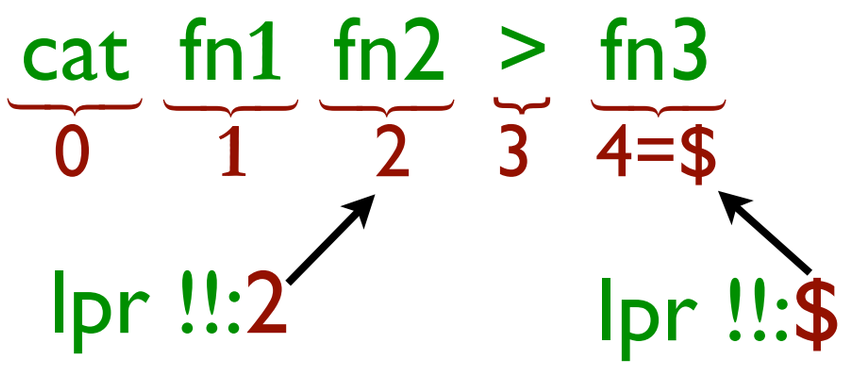
  `!!` ↵ voriges Kommando

  `!37` ↵ Kommando 37

  `!-2` ↵ vorletztes Kommando

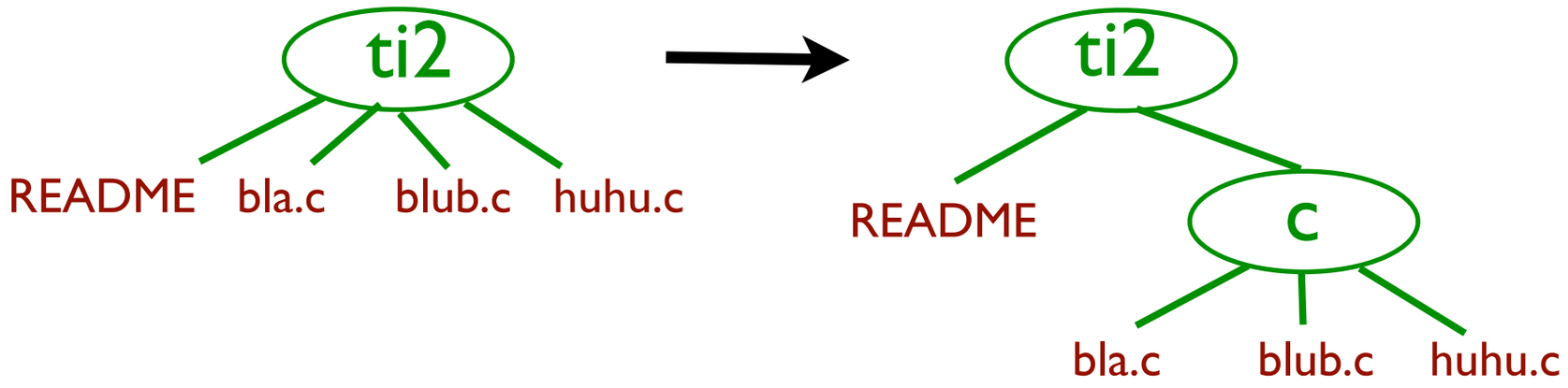
  `!r` ↵ letztes Kommando, das mit r begann

- Auch Zugriff auf einzelne Kommando-Teile (Token) möglich:



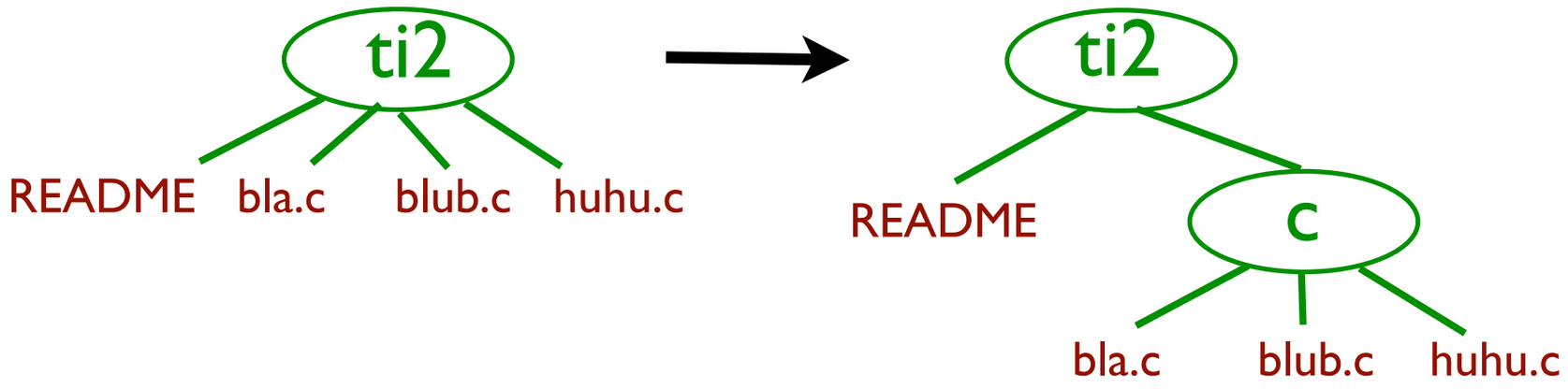
# Dateinamensubstitution

- Shell-Kommandos sollen häufig auf mehreren Dateien ausgeführt werden. Beispiel:



# Dateinamensubstitution

- Shell-Kommandos sollen häufig auf mehreren Dateien ausgeführt werden. Beispiel:

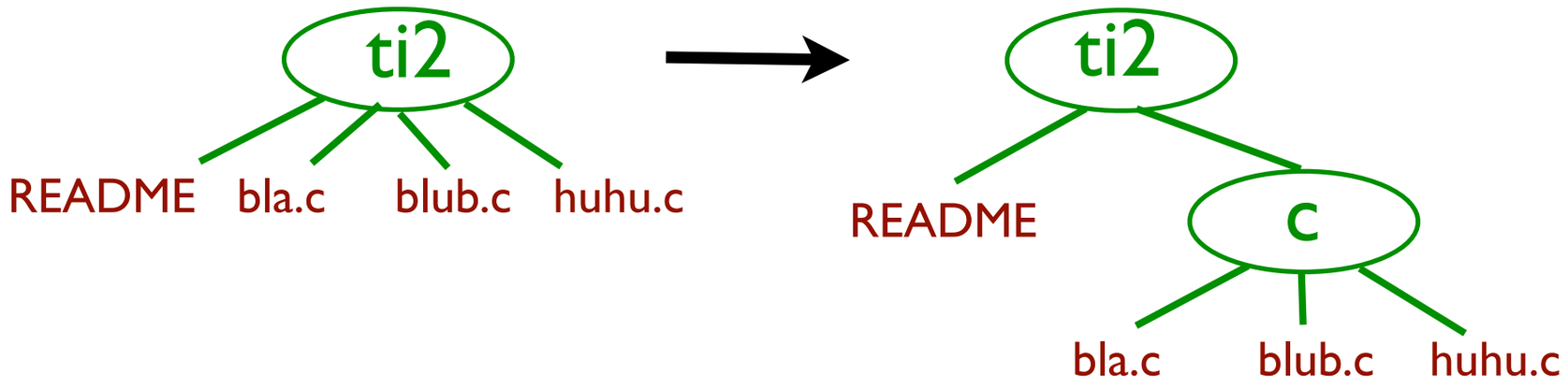



- Mögliche Alternativen (nach `mkdir c`):

a)
```
mv bla.c c/bla.c
mv blub.c c/blub.c
mv huhu.c c/huhu.c
```

# Dateinamensubstitution

- Shell-Kommandos sollen häufig auf mehreren Dateien ausgeführt werden. Beispiel:



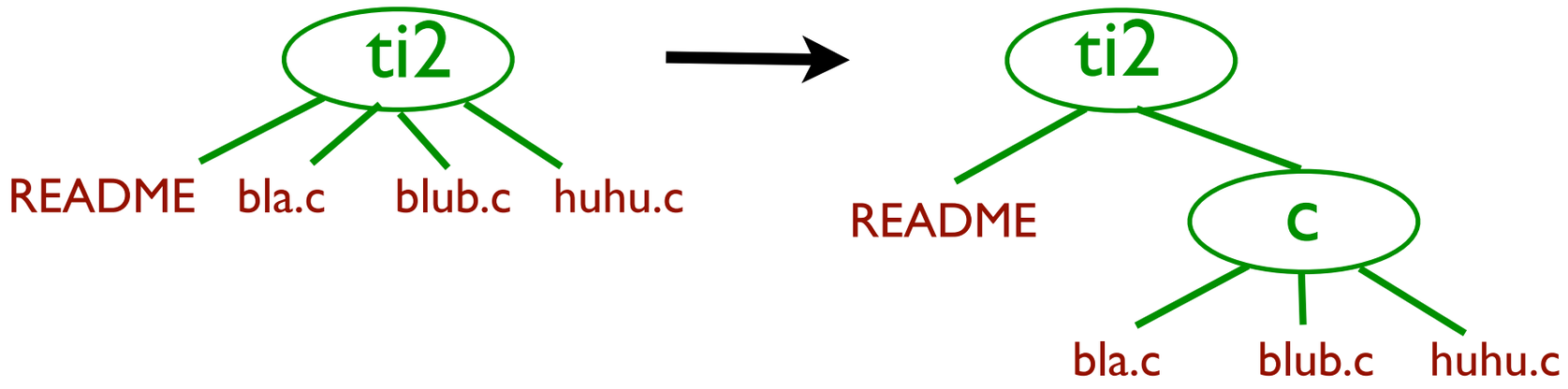

- Mögliche Alternativen (nach `mkdir c`):

a)
```
mv bla.c c/bla.c
mv blub.c c/blub.c
mv huhu.c c/huhu.c
```

b)
```
mv bla.c  c
mv blub.c  c
mv huhu.c  c
```

# Dateinamensubstitution

- Shell-Kommandos sollen häufig auf mehreren Dateien ausgeführt werden. Beispiel:




- Mögliche Alternativen (nach `mkdir c`):

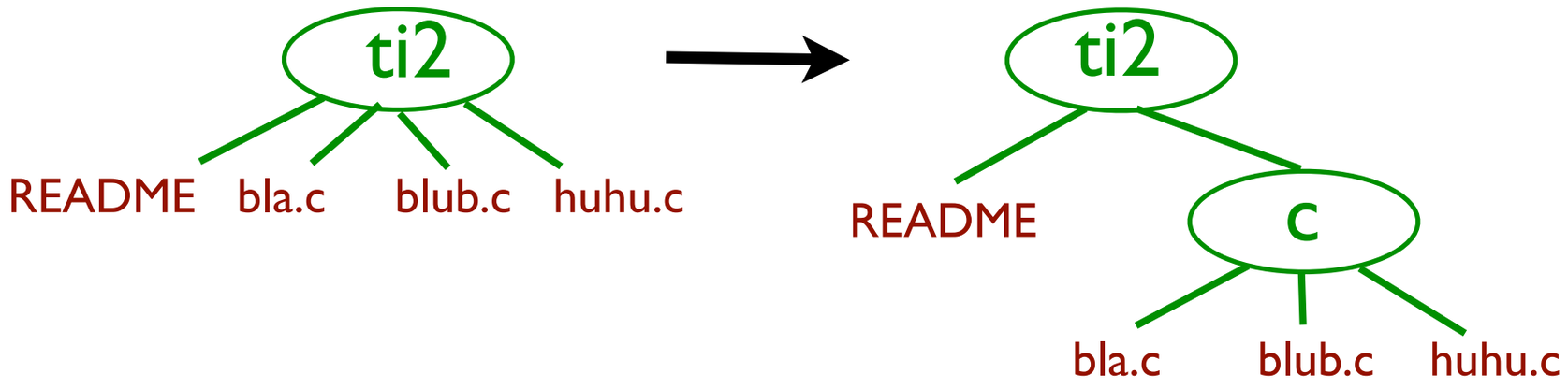
a)
```
mv bla.c c/bla.c
mv blub.c c/blub.c
mv huhu.c c/huhu.c
```

b)
```
mv bla.c  c
mv blub.c  c
mv huhu.c  c
```

c)
```
mv bla.c blub.c huhu.c c
```

# Dateinamensubstitution

- Shell-Kommandos sollen häufig auf mehreren Dateien ausgeführt werden. Beispiel:




- Mögliche Alternativen (nach `mkdir c`):

a)
```
mv bla.c c/bla.c
mv blub.c c/blub.c
mv huhu.c c/huhu.c
```

b)
```
mv bla.c  c
mv blub.c  c
mv huhu.c  c
```

c)
```
mv bla.c blub.c huhu.c c
```

d)
```
mv *.c c
```

- Dazu spezielle Form von Regular Expressions verwendbar

| | |
|---|---|
| `*` | beliebige Zeichenfolge (auch leer) |
| `?` | genau ein beliebiges Zeichen |
| `[abc]` | 1 Zeichen, a oder b oder c |
| `*.[hc]` | alles, was mit .h oder .c endet |
| `[a-z]` | 1 kleiner Buchstabe |
| `[a-zA-Z0-9]` | Buchstabe oder Ziffer |
| `~` | Eigene Home Directory |
| `~egon` | Home Directory von Egon |

# Quoting

- Dateinamen können auch Sonderzeichen enthalten

  `* ? [ ] ~ ! SP` ...

- Müssen entsprechend markiert werden:

```
cat "Dies ist ein Dateiname"
cat Dies\ ist\ ein\ Dateiname
```

# Umgebungsvariablen

- Definieren von Variablen (z.B. als Abkürzungsmechanismus)

```
var=wert
```

```
H=huhu.informatik.uni-bremen.de
ssh $H
```

# Umgebungsvariablen

- Definieren von Variablen (z.B. als Abkürzungsmechanismus)

```
var=wert

H=huhu.informatik.uni-bremen.de
ssh $H
```

- Datei .bash_profile in Home Directory enthält initiale Kommandos beim Starten der Bash nach dem Einloggen

⇒ dort können z.B. Umgebungsvariablen gesetzt werden

```
PRINTER=...
export PRINTER
```

⇒ implizites Argument von Kindprozessen

# Shell-Programmierung (Bash)

- Dateien können Kommandofolgen enthalten, die bei Aufruf ausgeführt werden

  ⇒ ausführbare Datei (x)

  ⇒ wird damit zu neuem „Kommando“

- Beispiel:

Ls

```
ls > bla
date >> bla
lprx bla
```

⇒ `chmod +x Ls`

`Ls`

# Shell-Programmierung (Bash)

- Dateien können Kommandofolgen enthalten, die bei Aufruf ausgeführt werden

  ⇒ ausführbare Datei (x)

  ⇒ wird damit zu neuem „Kommando“

- Beispiel:

**Ls**

```
ls > bla
date >> bla
lprx bla
```

⇒ `chmod +x Ls`

`Ls`

- Angabe von Parametern möglich:

**Cat**

```
cat $1 $2 > bla
date >> bla
lprx bla
```

⇒ `chmod +x Cat`

`Cat huhu blub`

# Shell-Programmierung (Bash)

- Dateien können Kommandofolgen enthalten, die bei Aufruf ausgeführt werden

  ⇒ ausführbare Datei (x)

  ⇒ wird damit zu neuem „Kommando“

- Beispiel:

**Ls**

```
ls > bla
date >> bla
lprx bla
```

⇒ `chmod +x Ls`

`Ls`

- Angabe von Parametern möglich:

**Cat**

```
cat $1 $2 > bla
date >> bla
lprx bla
```

⇒ `chmod +x Cat`

`Cat huhu blub`

- Beispiel: „C-Compiler“ `cc` ist Shell-Kommando

```
/lib/ccom $1 | as
ld /lib/crt0.o a.out -lc
```

⇒ `cc bla.c`

Bietet auch einfache „Programmiersprache“:

- Schleifen:

```
while ...
```

- Fallunterscheidungen:

```
if ... then ... else ... fi
case ...
```

- Tests:

```
if [$1=bla]
then ...
else ...
fi
```

- …

# Fragen – Teil 3

- Wofür könnte man das folgende Shell-Kommando verwenden?

  mv *.pdf bla

- Wie macht man ein soeben editiertes Shell-File ausführbar?

# Zusammenfassung

- Prozesshierarchie
- Vordergrund-/Hintergrundprozesse
- Prozesssteuerung in der Shell
- Signale
- Interprozesskommunikation
- Effizientes Arbeiten mit der Shell

# Unix-Prozesse aus Nutzersicht – Fragen

1. `bla` sei ein im Pfad ausführbares Programm. Was ist der Unterschied zwischen dem Aufruf `bla` und dem Aufruf `bla &` in der Shell? Welche Auswirkungen hat dies, wenn `bla` von *Standard Input* liest bzw. auf *Standard Output* schreibt?
2. Wie kann ein Nutzer herausfinden, welche Prozesse in einem Unix-System gerade existieren?
3. Was ist eine *Pipe*?
4. Wofür könnte man das folgende Shell-Kommando verwenden?

   mv *.pdf bla

5. Wie macht man ein soeben editiertes Shell-File ausführbar?