Prozessverwaltung (2)

Ute Bormann, TI2

2023-10-13

Inhalt

1. Prozesserzeugung
2. Prozesstermination
3. Überblick Nebenläufigkeit
4. Nebenläufigkeit in Unix

Teil 1:
Prozesserzeugung

Prozesserzeugung (in Unix)

- Statische Anzahl von Prozessen i.d.R. zu inflexibel

  ⇒ Prozesse werden bei Bedarf erzeugt

  ⇒ Terminieren, wenn Aufgabe erledigt

Prozesserzeugung (in Unix)

- Statische Anzahl von Prozessen i.d.R. zu inflexibel

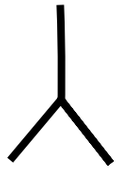
  ⇒ Prozesse werden bei Bedarf erzeugt

  ⇒ Terminieren, wenn Aufgabe erledigt

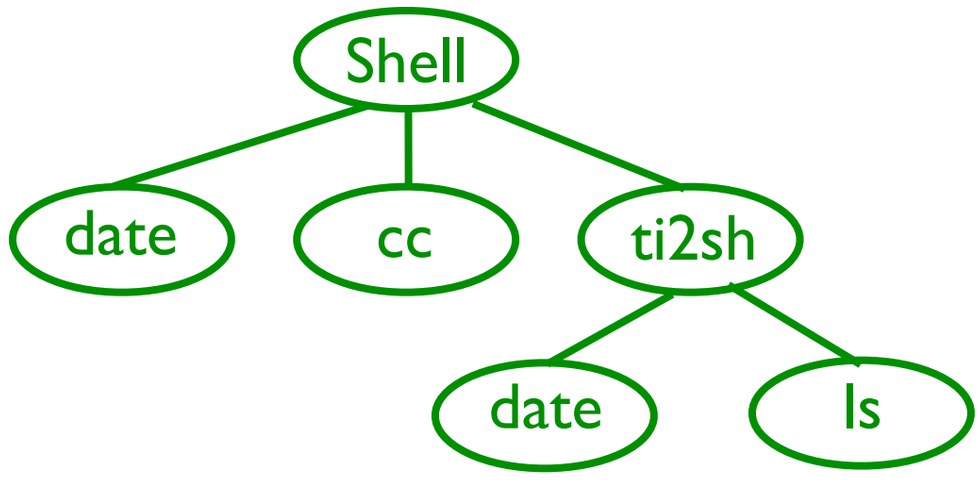
- Erzeugung aus Prozesskontext heraus

  ⇒ neuer Prozess ist Kindprozess dieses Prozesses

fork()



⇒ Prozess-Hierarchie



⇒ „Erster Prozess“ (Proc 0) ist „handgearbeitet“

- Kindprozess ist logische Kopie des Vaterprozesses

  ⇒gleicher Adressraum (Text, Daten, Stack), offene Dateien, Umgebungsvariablen, Pipes, … ⇒ Vererbung

  ⇒beide Prozesse durchlaufen dasselbe Programm

- Kindprozess ist logische Kopie des Vaterprozesses

  ⇒gleicher Adressraum (Text, Daten, Stack), offene Dateien, Umgebungsvariablen, Pipes, … ⇒ Vererbung

  ⇒beide Prozesse durchlaufen dasselbe Programm

- Ausnahme: fork() liefert:
  - beim Vater: Prozess-ID des Kindprozesses
  - beim Kind: 0

  ⇒ Programm kann dies unterscheiden:

```
main() {
   int pid;
   pid = fork();
   if (pid == 0) {
      ... Kind ...
   } else {
      ... Vater ...
   }
}
```

(genaugenommen noch Fehlerfall bei Rückgabewert von –1)

- Kind soll meist anderes Programm ausführen (z.B. Shell: date)

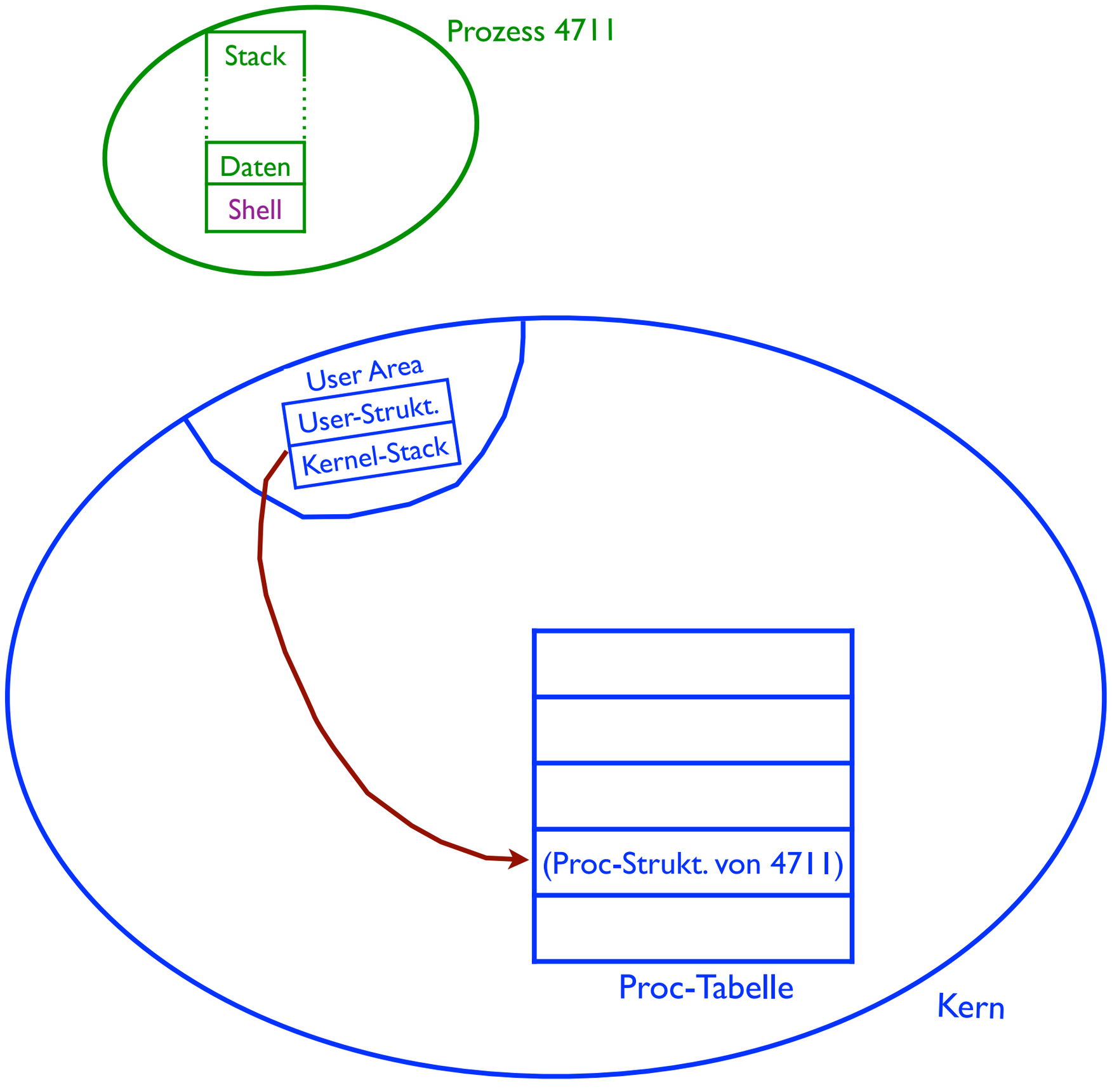
  exec() (eigentlich: execve())

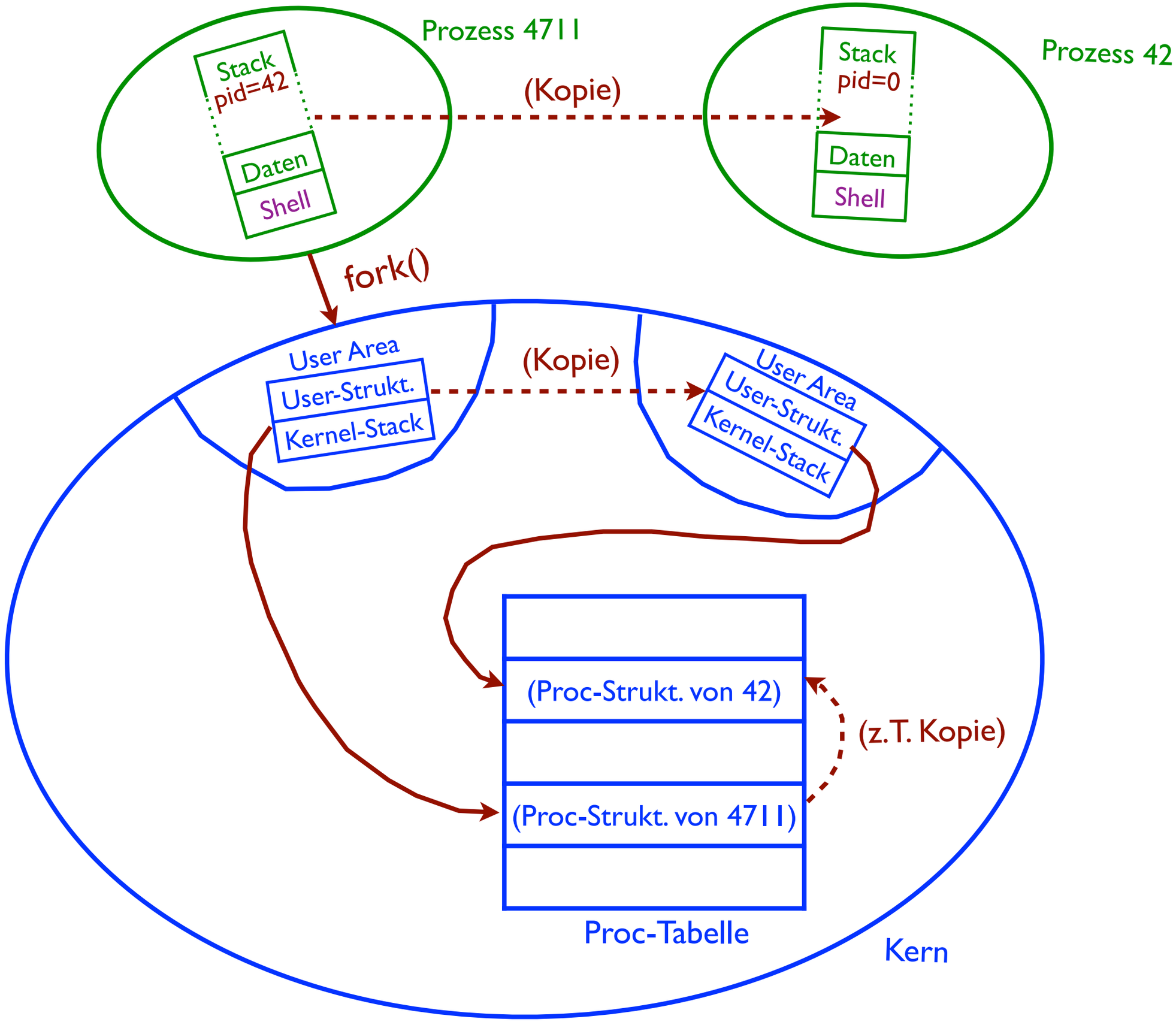
  ⇒ Austausch des Adressraums eines Prozesses

  ⇒ Starten des angegebenen Programms

- Vor exec() u.U. Ein-/Ausgabeumlenkung durchführen
  (z.B. date > bla)

  ⇒ Innerhalb des Kindprozesses im kopierten Adressraum des Vaters


Prozess 4711
Stack
Daten
Shell
User Area
User-Strukt.
Kernel-Stack
(Proc-Strukt. von 4711)
Proc-Tabelle
Kern


Prozess 4711
Stack
pid=42
Daten
Shell
(Kopie)
Prozess 42
Stack
pid=0
Daten
Shell
fork()
User Area
User-Strukt.
Kernel-Stack
(Kopie)
User Area
User-Strukt.
Kernel-Stack
(Proc-Strukt. von 42)
(Proc-Strukt. von 4711)
(z.T. Kopie)
Proc-Tabelle
Kern

Prozess 4711

Stack
pid=42

Daten

Shell

Prozess 42

Stack

Daten

date

exec (date...)

User Area

User-Strukt.

Kernel-Stack

(Kopie)

User Area

User-Strukt.

Kernel-Stack

(Proc-Strukt. von 42)

(Proc-Strukt. von 4711)

(z.T. Kopie)

Proc-Tabelle

Kern

- Über C-Bibliothek viele exec-Varianten verfügbar:

  z.B. execvp (file, argv)

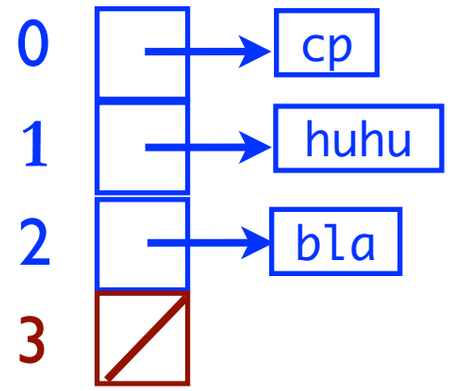
  „ausführbare Datei“ Argumentvektor

- Beispiel: Shell

  cp huhu bla

  - Erzeugen eines Kindprozesses (fork())
  - Starten des cp-Programms

  execvp ("cp", argv)



- Über C-Bibliothek viele exec-Varianten verfügbar:

  z.B. execvp (file, argv)

  „ausführbare Datei“ Argumentvektor

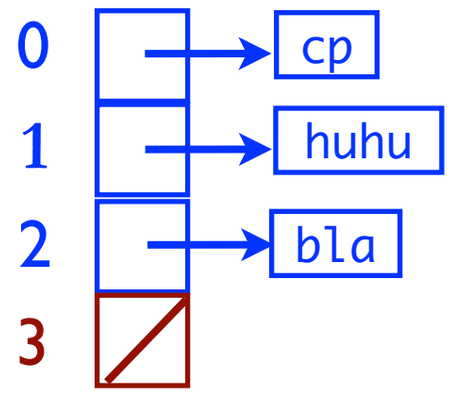
- Beispiel: Shell

  cp huhu bla

  - Erzeugen eines Kindprozesses (fork())
  - Starten des cp-Programms

  execvp ("cp", argv)

  argv:



- execvp() erhält implizit Umgebungsvariablen des Vaterprozesses

  z.B. PATH

  ⇒ Welche „Pfade“ sollen zum Auffinden der ausführbaren Datei durchsucht werden? (z.B. /home/ute/bin, /usr/bin)

- Ausführbare Datei besteht aus:
  - übersetztem C-Programm (evtl. mehrere Teile)

```
main () {
   ...
}
```

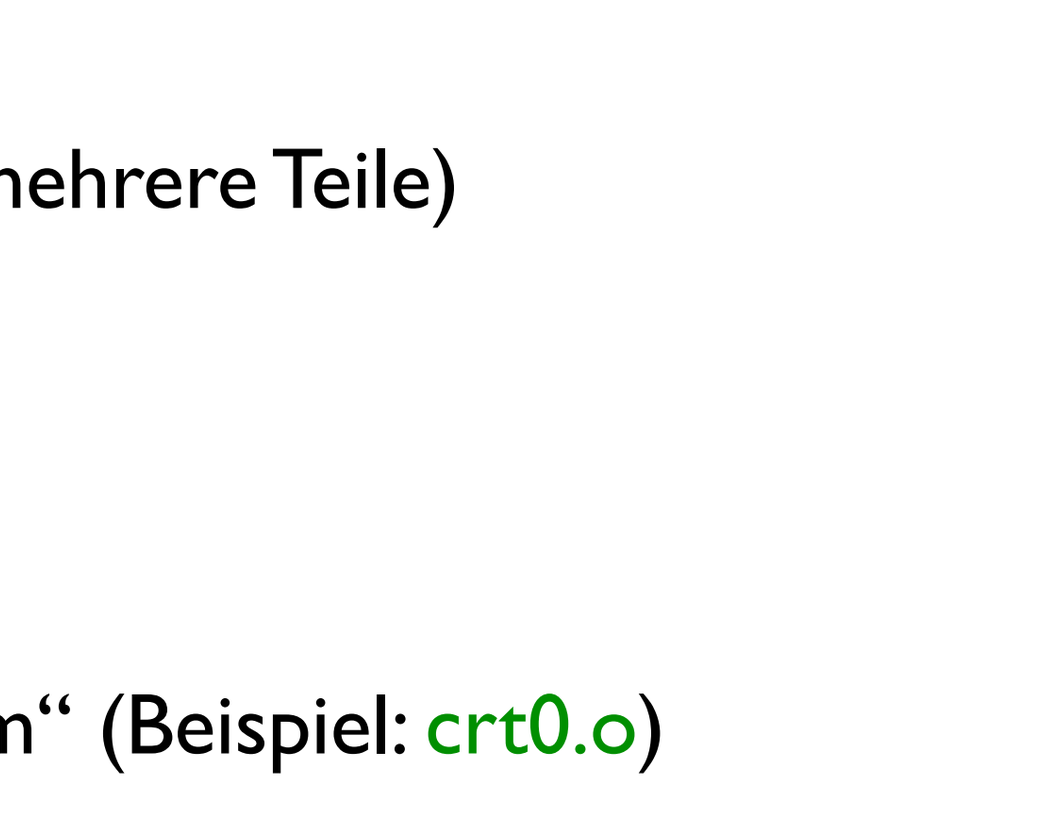
  - dazugebundenen Bibliotheken
  - dazugebundenem „Laufzeitsystem“ (Beispiel: crt0.o)

⇒ exec bewirkt Starten des Programms

⇒ Beginn in crt0.o:

```
/* Ermitteln von argc und argv */
status = main (argc, argv);
exit (status);
```

Erinnerung: a.out-Format



Fragen – Teil 1

- Skizziere kurz die Prozesserzeugung in Unix. Welche Rolle spielen die Systemaufrufe `fork()` und `exec()` dabei?

Teil 2:
Prozesstermination

Termination eines Prozesses

- Durch Aufruf von exit (status)
  (oder durch nicht abgefangenes Signal)
- Freigeben aller „Betriebsmittel“ (Dateien, Adressraum, ...)
- Außer Proc-Struktur (Status eintragen)
- Übergang in Zustand SZOMB („Zombie“)
- Aufruf von swtch()

Termination eines Prozesses

- Durch Aufruf von exit (status)
  (oder durch nicht abgefangenes Signal)
- Freigeben aller „Betriebsmittel“ (Dateien, Adressraum, ...)
- Außer Proc-Struktur (Status eintragen)
- Übergang in Zustand SZOMB („Zombie“)
- Aufruf von swtch()

- Terminierte Kindprozesse werden i.d.R. vom Vater „ausgewertet“

  ⇒ Grund der Termination (exit-Status, Signal, ...)

- Auswertung geschieht im Systemaufruf wait()

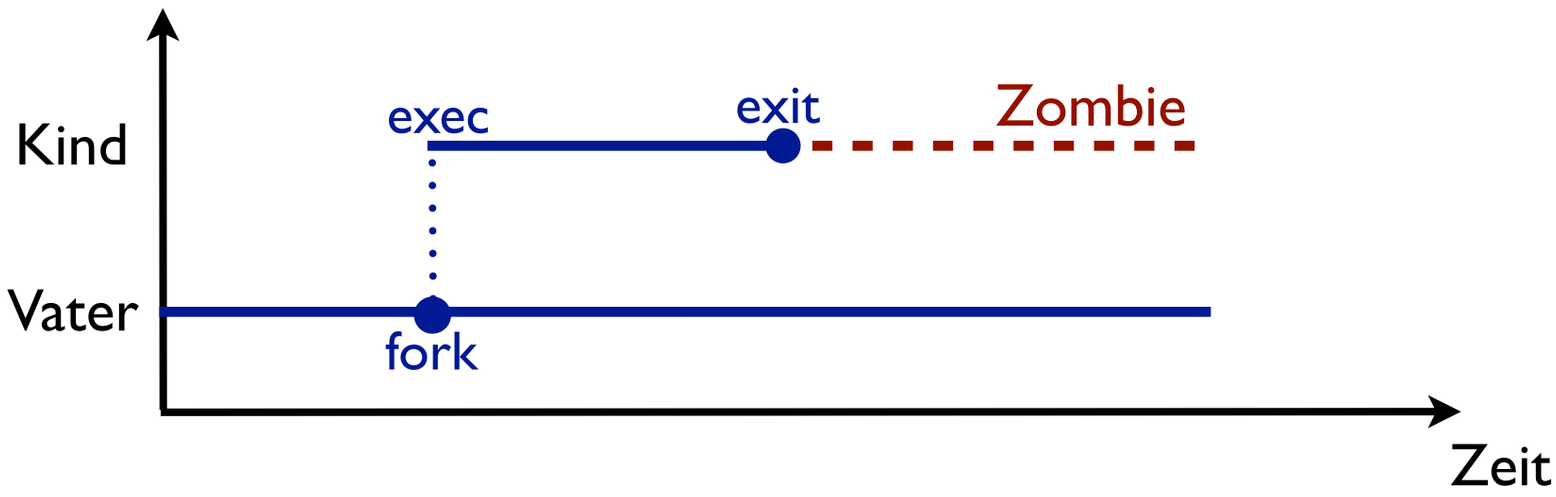
  ⇒ falls gerade kein terminiertes Kind: warten darauf (sleep)

  ⇒ liefert PID und Status des/eines terminierten Kindes zurück

  ⇒ Proc-Struktur des terminierten Kindes wird freigegeben

Prozesstermination: Mögliche Situationen

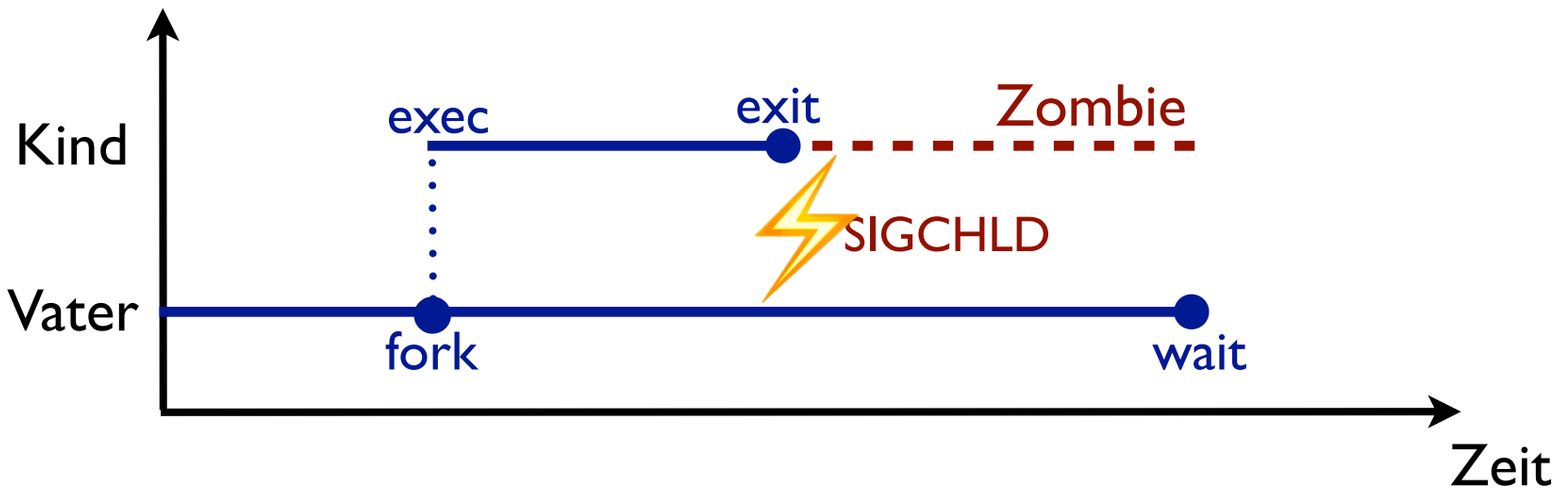
Variante a)



- Beispiel: Shell startet Kindprozess im Hintergrund

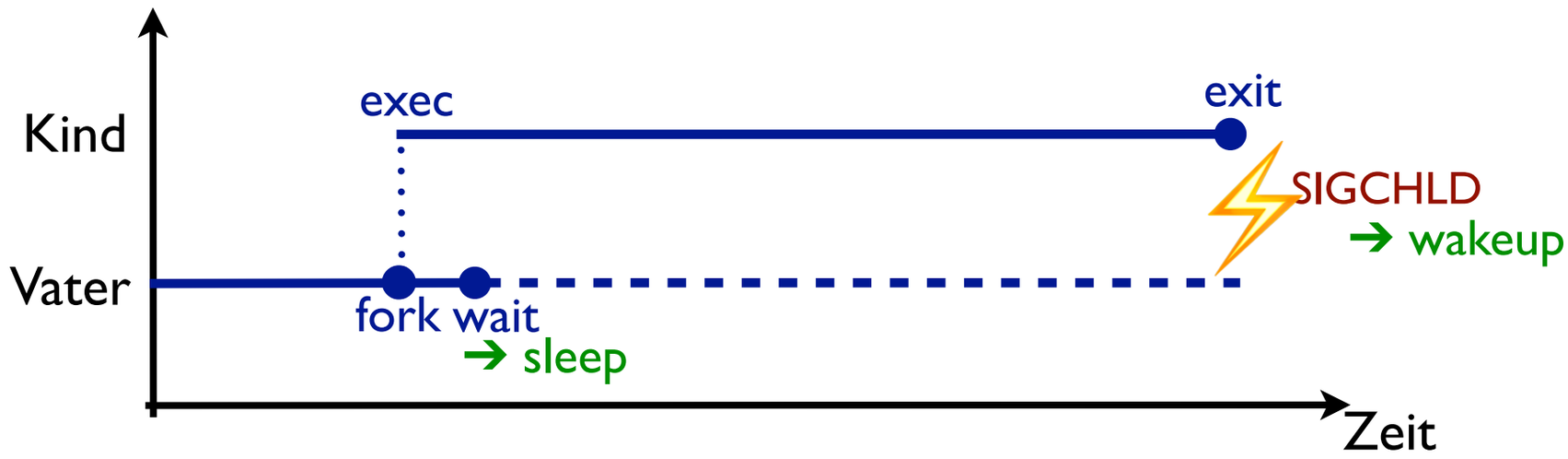
Prozesstermination: Mögliche Situationen

Variante a)



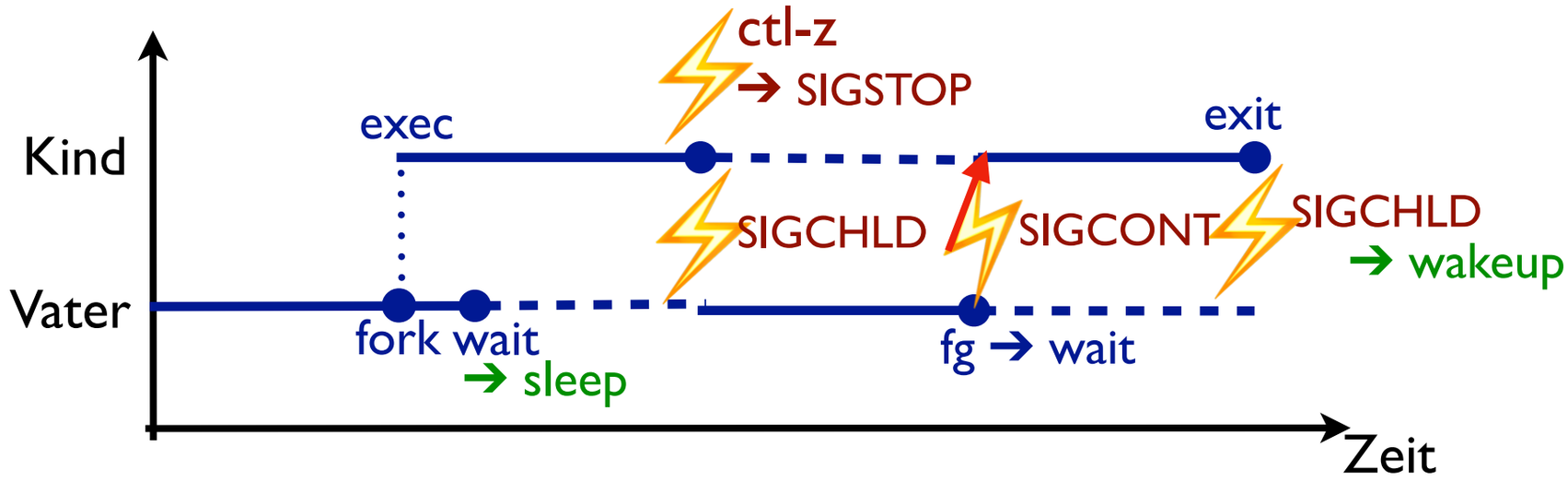
- Beispiel: Shell startet Kindprozess im Hintergrund
- Melden der Termination über Signal SIGCHLD
  - entweder ignorieren ⇒ Kind bleibt Zombie
  - oder behandeln ⇒ wait() aufrufen

Variante b)



- Beispiel: Shell startet Kindprozess im Vordergrund
- Vater wartet sofort auf Termination des Kindes

Variante b)



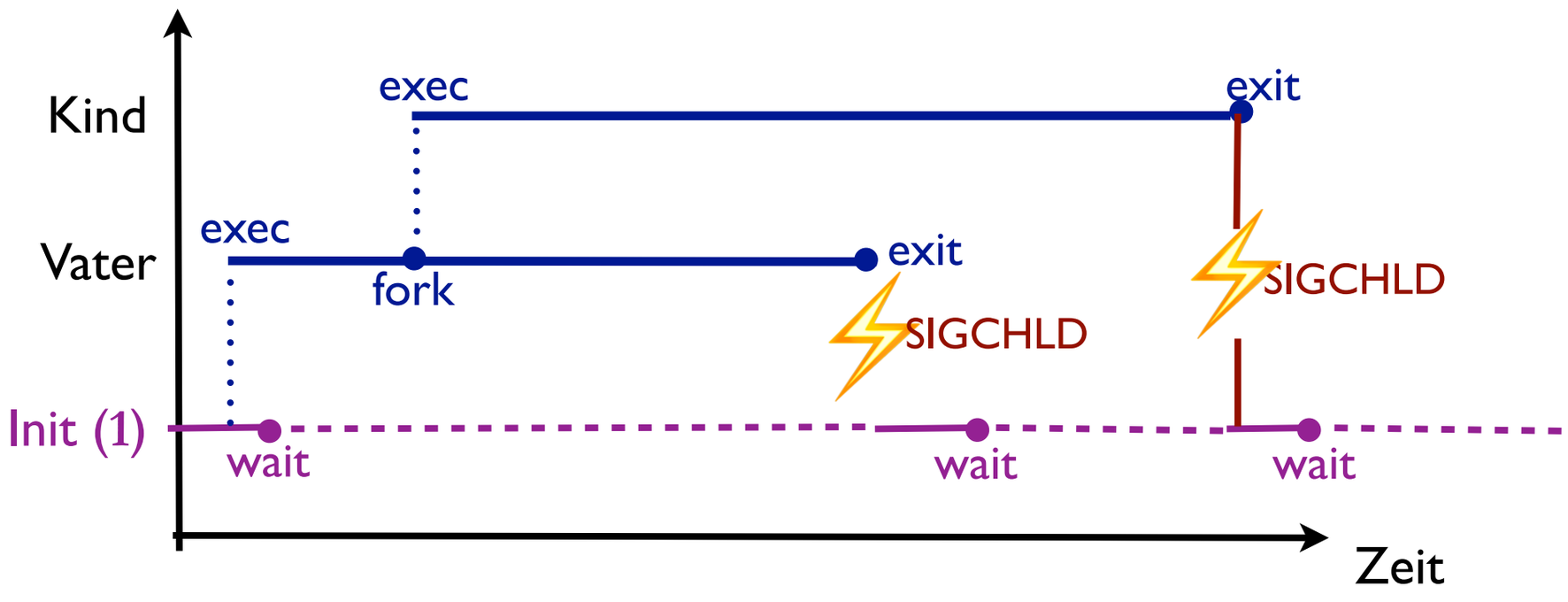
- Beispiel: Shell startet Kindprozess im Vordergrund
- Vater wartet sofort auf Termination des Kindes
- Dazwischen natürlich Wechsel zwischen Vordergrund und Hintergrund möglich

Variante c)

- Vaterprozess terminiert vor Termination/Auswertung des Kindprozesses

  ⇒ genaues Verhalten hängt von Unix-Version ab

- Im Prinzip:
  - Bei Termination werden (auch terminierte) Kindprozesse an Prozess 1 (= Init) vererbt
  - Init wartet ununterbrochen auf „Zombies“



Einige interessante Fälle:

- Vater hatte terminierte Kindprozesse ignoriert

  ⇒ Freigeben der Proc-Struktur des terminierten Kindes durch Init

Einige interessante Fälle:

- Vater hatte terminierte Kindprozesse ignoriert

  ⇒ Freigeben der Proc-Struktur des terminierten Kindes durch Init

- Noch aktive Kindprozesse, und Vater war „normaler“ Prozess

  ⇒ Kinder laufen weiter

  ⇒ werden später von Init aufgesammelt

Einige interessante Fälle:

- Vater hatte terminierte Kindprozesse ignoriert

  ⇒ Freigeben der Proc-Struktur des terminierten Kindes durch Init

- Noch aktive Kindprozesse, und Vater war „normaler“ Prozess

  ⇒ Kinder laufen weiter

  ⇒ werden später von Init aufgesammelt

- Noch aktive Kindprozesse, und Vater war „Controlling“ Prozess (z.B. Shell des betreffenden Bildschirmfensters)

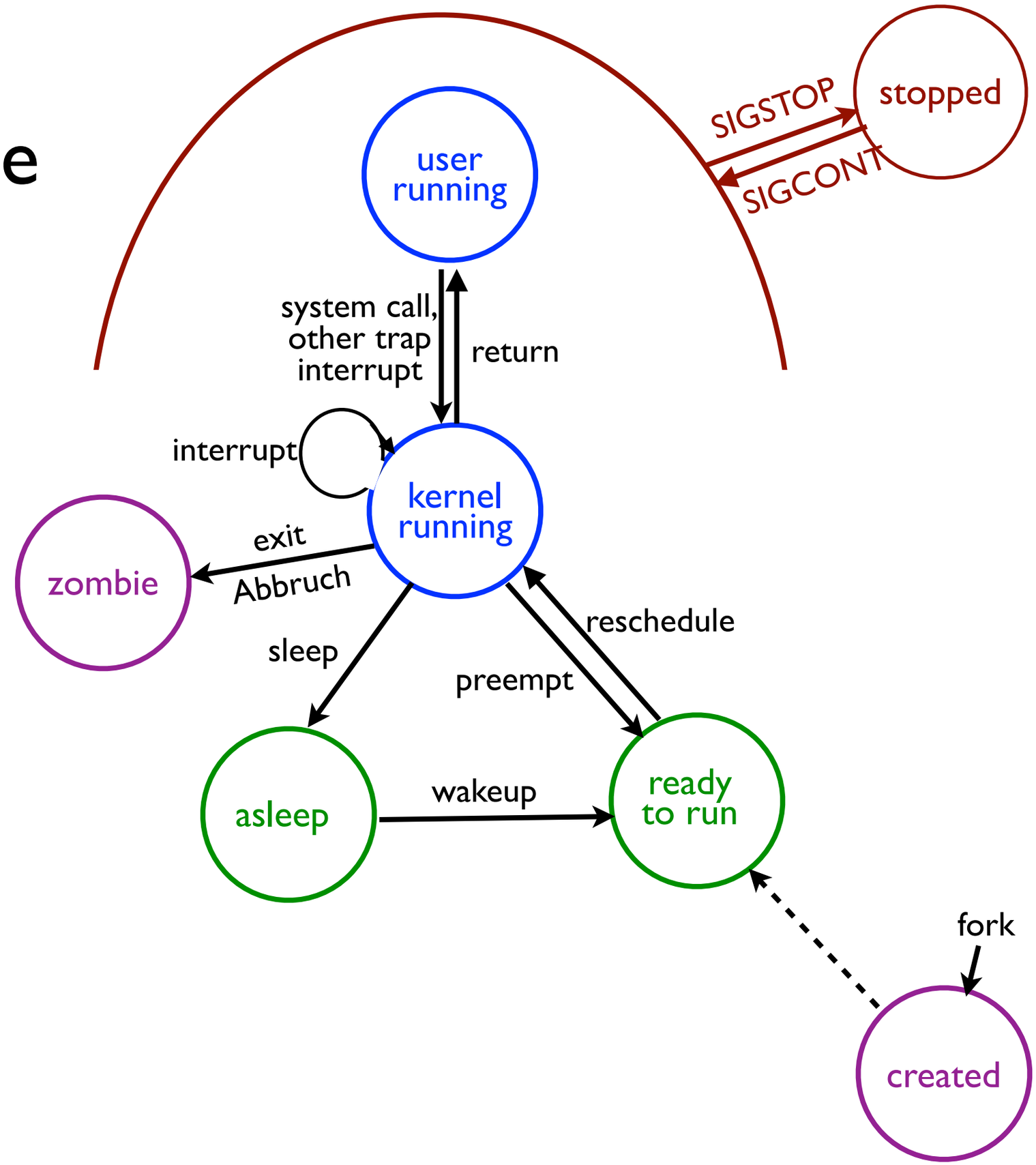
  ⇒ aktive Kindprozesse im Hintergrund laufen i.d.R. weiter

  ⇒ „Gestoppte“ Kindprozesse und Kindprozesse im Vordergrund bekommen (Hangup-)Signal (SIGHUP)

  ⇒ führt i.d.R. zur Termination

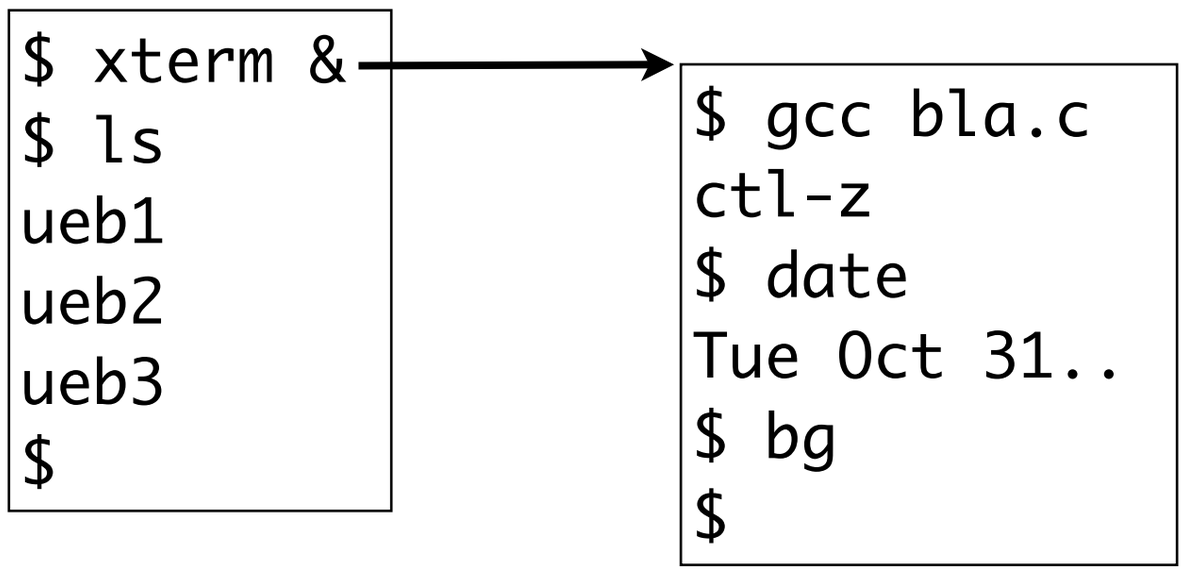
Prozesszustände

(vereinfacht)



Kleine Aufgabe

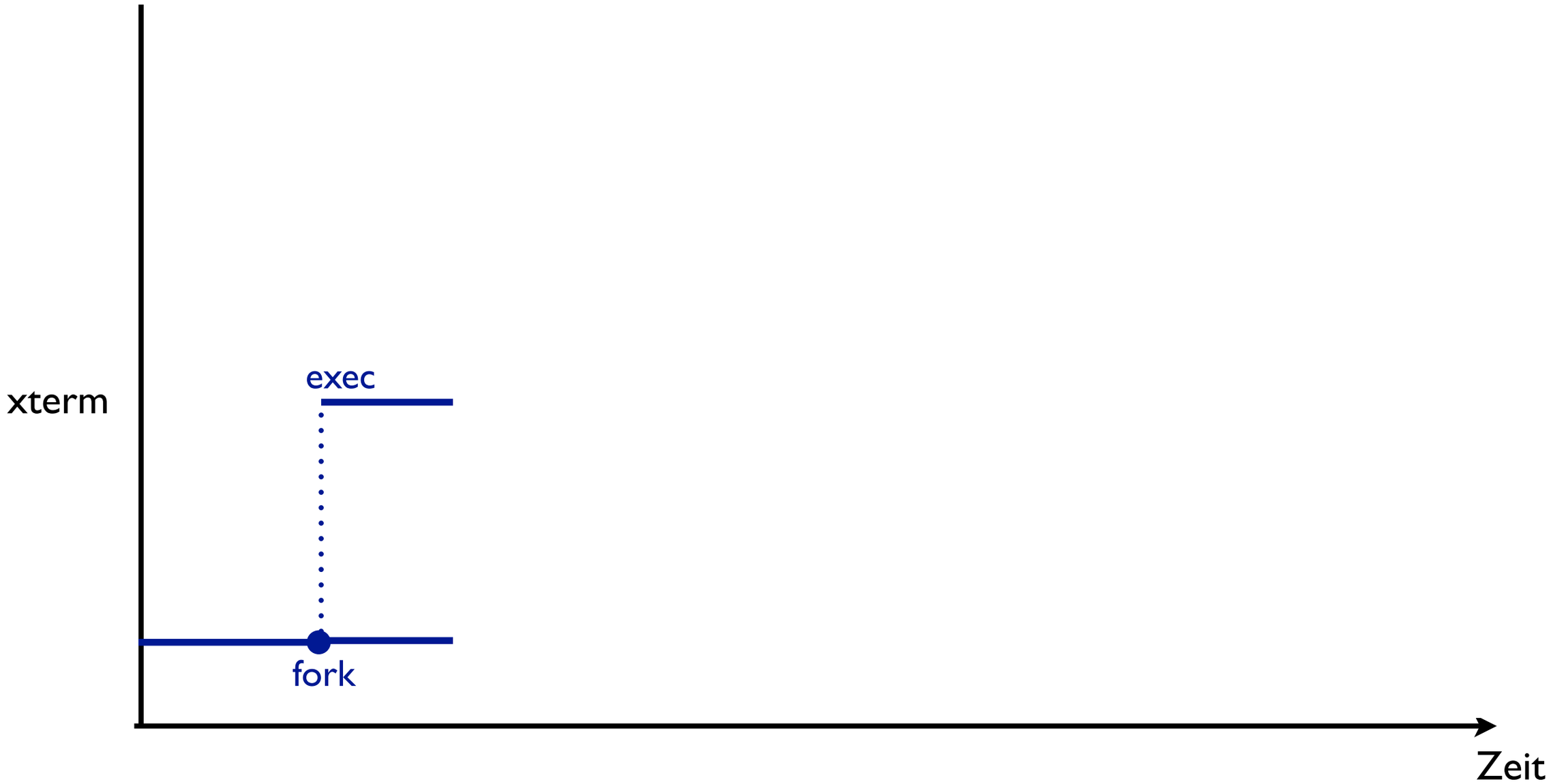
- Gegeben sei die angegebene Abfolge von Shell-Kommandos.
- Visualisiere Prozesserzeugungen/-terminationen und Signale wie in den Folien zur Prozesstermination.



Kleine Aufgabe

```
$ xterm &
$ ls
ueb1
ueb2
ueb3
$
```

```
$ gcc bla.c
ctl-z
$ date
Tue Oct 31..
$ bg
$
```



Fragen – Teil 2

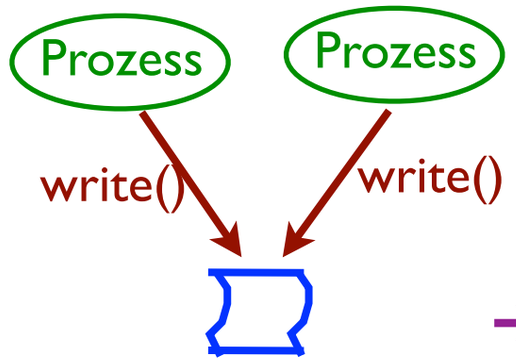
- Wie erfährt ein Unix-Prozess, ob ein Kindprozess terminiert ist?
- Kann ein Kindprozess weiterlaufen, wenn sein Vaterprozess bereits terminiert ist?
- Wozu gibt es in Unix den Prozesszustand SZOMB („Zombie “)?

Teil 3:
Überblick Nebenläufigkeit

Überblick Nebenläufigkeit

- Prozesse teilen sich Betriebsmittel

  Beispiel: Zugriff auf Drucker

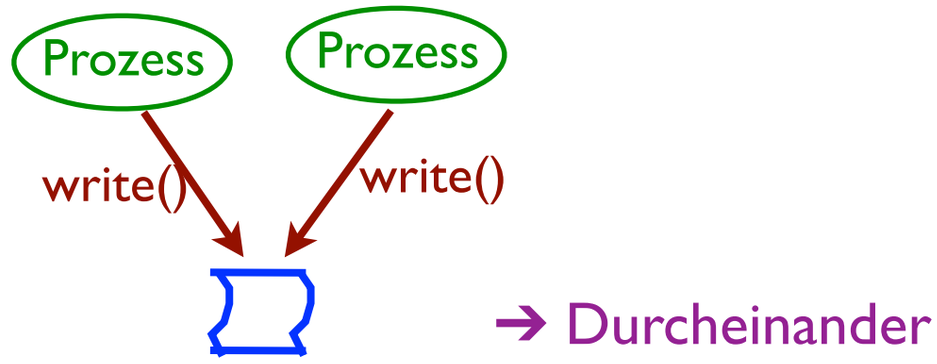


➔ Durcheinander

Überblick Nebenläufigkeit

- Prozesse teilen sich Betriebsmittel

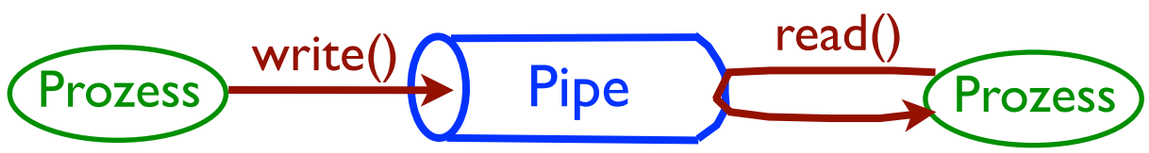
  Beispiel: Zugriff auf Drucker



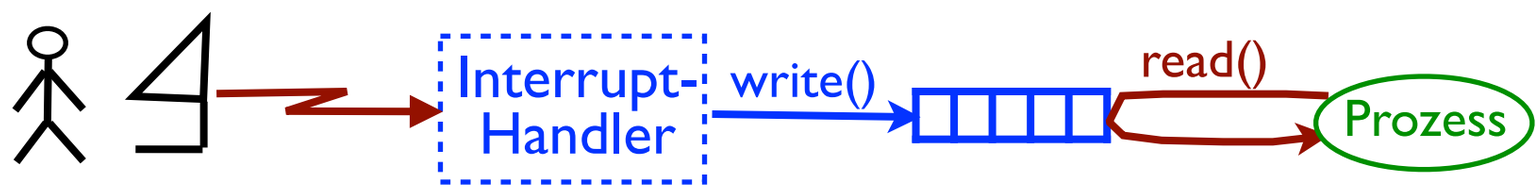
→ Durcheinander

- „Aktivitäten“ greifen auf gemeinsame Datenstrukturen zu

  Beispiel 1: Pipes (z.B. `date | lprx`)



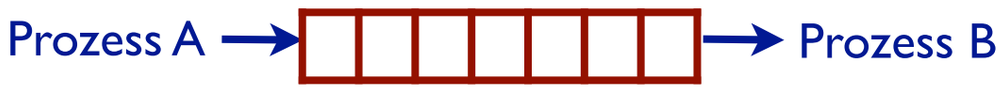
  Beispiel 2: Gerätewarteschlangen



⇒ Wo ist das Problem?

Beispiel: „Realisierung“ einer Pipe (vereinfacht)

⇒ Modellierung einer Warteschlange



- Füllstandszähler count
- maximale Länge maxcount

Füllen (A)

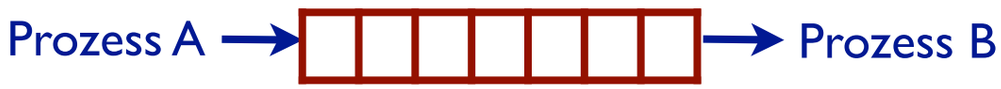
```
while (count == maxcount) {
  /* warten */
};
/* Element anhängen */
count++;
```

Leeren (B)

```
while (count == 0) {
  /* warten */
};
/* 1. Element entfernen */
count--;
```

Beispiel: „Realisierung“ einer Pipe (vereinfacht)

⇒ Modellierung einer Warteschlange



- Füllstandszähler count
- maximale Länge maxcount

Füllen (A)

```
while (count == maxcount) {
  /* warten */
};
/* Element anhängen */
count++;
```

Maschineninstruktionen, z.B.

```
LOAD count
ADD 1
STORE count
```

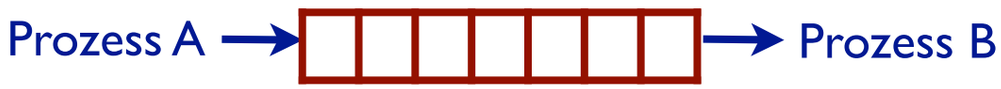
Leeren (B)

```
while (count == 0) {
  /* warten */
};
/* 1. Element entfernen */
count--;
```

```
LOAD count
SUB 1
STORE count
```

Beispiel: „Realisierung“ einer Pipe (vereinfacht)

⇒ Modellierung einer Warteschlange



- Füllstandszähler count
- maximale Länge maxcount

Füllen (A)

```
while (count == maxcount) {
  /* warten */
};
/* Element anhängen */
count++;
```

Leeren (B)

```
while (count == 0) {
  /* warten */
};
/* 1. Element entfernen */
count--;
```

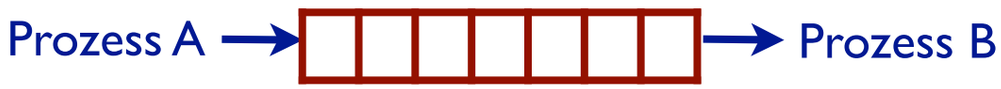
Maschineninstruktionen, z.B.

```
LOAD count    =7
                          LOAD count    =7
ADD 1         =8
                          SUB 1         =6
STORE count   =8
                          STORE count   =6
```

⇒ Wert von count falsch

Beispiel: „Realisierung“ einer Pipe (vereinfacht)

⇒ Modellierung einer Warteschlange



- Füllstandszähler count
- maximale Länge maxcount

Füllen (A)

```
while (count == maxcount) {
  /* warten */
};
/* Element anhängen */
count++;
```

Leeren (B)

```
while (count == 0) {
  /* warten */
};
/* 1. Element entfernen */
count--;
```

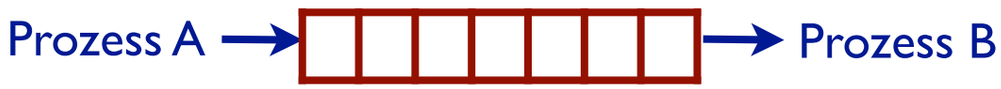
Maschineninstruktionen, z.B.

```
LOAD count     =7
                                  LOAD count     =7
                                  SUB 1          =6
                                  STORE count    =6
ADD 1          =8
STORE count    =8
```

⇒ Wert von count falsch

Beispiel: „Realisierung“ einer Pipe (vereinfacht)

⇒ Modellierung einer Warteschlange



- Füllstandszähler count
- maximale Länge maxcount

Füllen (A)

```
while (count == maxcount) {
  /* warten */
};
/* Element anhängen */
count++;
```

Leeren (B)

```
while (count == 0) {
  /* warten */
};
/* 1. Element entfernen */
count--;
```

Maschineninstruktionen, z.B.

```
LOAD count     =7
                          LOAD count     =7
                          SUB 1          =6
                          STORE count    =6
ADD 1          =8
STORE count    =8
```

⇒ Wert von count falsch

Algorithmus hat noch weitere Nebenläufigkeitsprobleme ➔ später

- Programmablauf von A darf u.U. nicht an beliebigen Stellen durch B unterbrochen werden
  (und umgekehrt)

  ⇒ Kritische Abschnitte schützen

```
A:                    B:
count++;              count--;
```

- Programmablauf von A darf u.U. nicht an beliebigen Stellen durch B unterbrochen werden
  (und umgekehrt)

  ⇒ Kritische Abschnitte schützen

```
A: lock();                    B:  lock();
   count++;                       count--;
   unlock();                      unlock();
```

⇒ gewährleistet gegenseitigen Ausschluss

⇒ entweder A oder B

- lock()/unlock() so implementieren, dass:
  - nur einer durchkommt
    (Achtung: lock() ist u.U. auch
    kritischer Abschnitt)
  - keiner davor „verhungert“
  - keine Verklemmungen entstehen
  - kein „After-you-after-you“-Problem entsteht

Fragen – Teil 3

- Beschreibe ein typisches Nebenläufigkeitsproblem.

Teil 4:
Nebenläufigkeit in Unix

Nebenläufigkeit in Unix (Einprozessorsystem)

- Eine CPU wird gemeinsam benutzt von:
  - Prozessen im User-Mode
  - Prozessen im Kernel-Mode
  - Interrupthandlern

(keine Prozesse, leihen sich CPU nur aus)

⇒ kein Prozesskontext, kein PID,...

Mögliche Situationen:

a) User-Mode A ⇝ Interrupthandler

⇒ keine gemeinsamen Datenstrukturen

Mögliche Situationen:

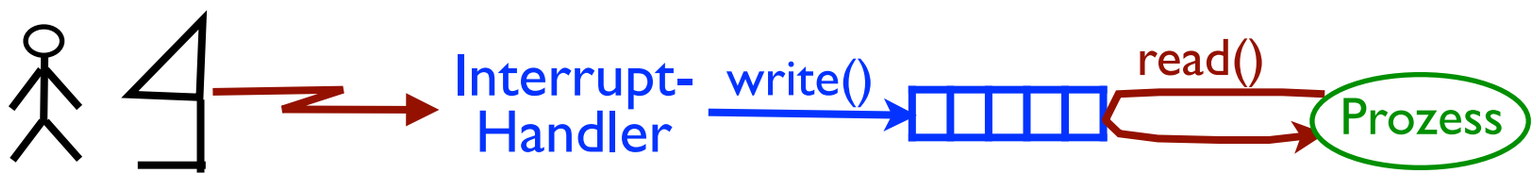
a) User-Mode A ⇝ Interrupthandler

⇒ keine gemeinsamen Datenstrukturen

b) Kernel-Mode A ⇝ Interrupthandler

⇒ Schutz von kritischen Abschnitten durch temporären Ausschluss von Interrupt(s)

Beispiel:



Mögliche Situationen:

a) User-Mode A ⇝ Interrupthandler

⇒ keine gemeinsamen Datenstrukturen

b) Kernel-Mode A ⇝ Interrupthandler

⇒ Schutz von kritischen Abschnitten durch temporären Ausschluss von Interrupt(s)

c) Interrupthandler x ⇝ Interrupthandler y

⇒ Anderer Interrupt (höhere Priorität)

⇒ i.d.R. auch andere Datenstrukturen (oder Interruptausschluss)

d) User-Mode A  User-Mode B

⇒ i.d.R. keine gemeinsamen Datenstrukturen
(getrennte Adressräume)

⇒ andernfalls (Shared Memory) ggf. Schutz durch Locking

d) User-Mode A ⇝ (clock interrupt) ⇝ User-Mode B

⇒ i.d.R. keine gemeinsamen Datenstrukturen (getrennte Adressräume)

⇒ andernfalls (Shared Memory) ggf. Schutz durch Locking

e) User-/Kernel-Mode A —sleep()→ User-/Kernel-Mode B

⇒ geplante Umschaltung

⇒ ggf. Schutz durch Locking

d) User-Mode A ⇝ (clock interrupt) ⇝ User-Mode B

⇒ i.d.R. keine gemeinsamen Datenstrukturen (getrennte Adressräume)

⇒ andernfalls (Shared Memory) ggf. Schutz durch Locking

e) User-/Kernel-Mode A —sleep()→ User-/Kernel-Mode B

⇒ geplante Umschaltung

⇒ ggf. Schutz durch Locking

f) Kernel-Mode A ⇝ (clock interrupt) ⇝ Kernel-Mode B

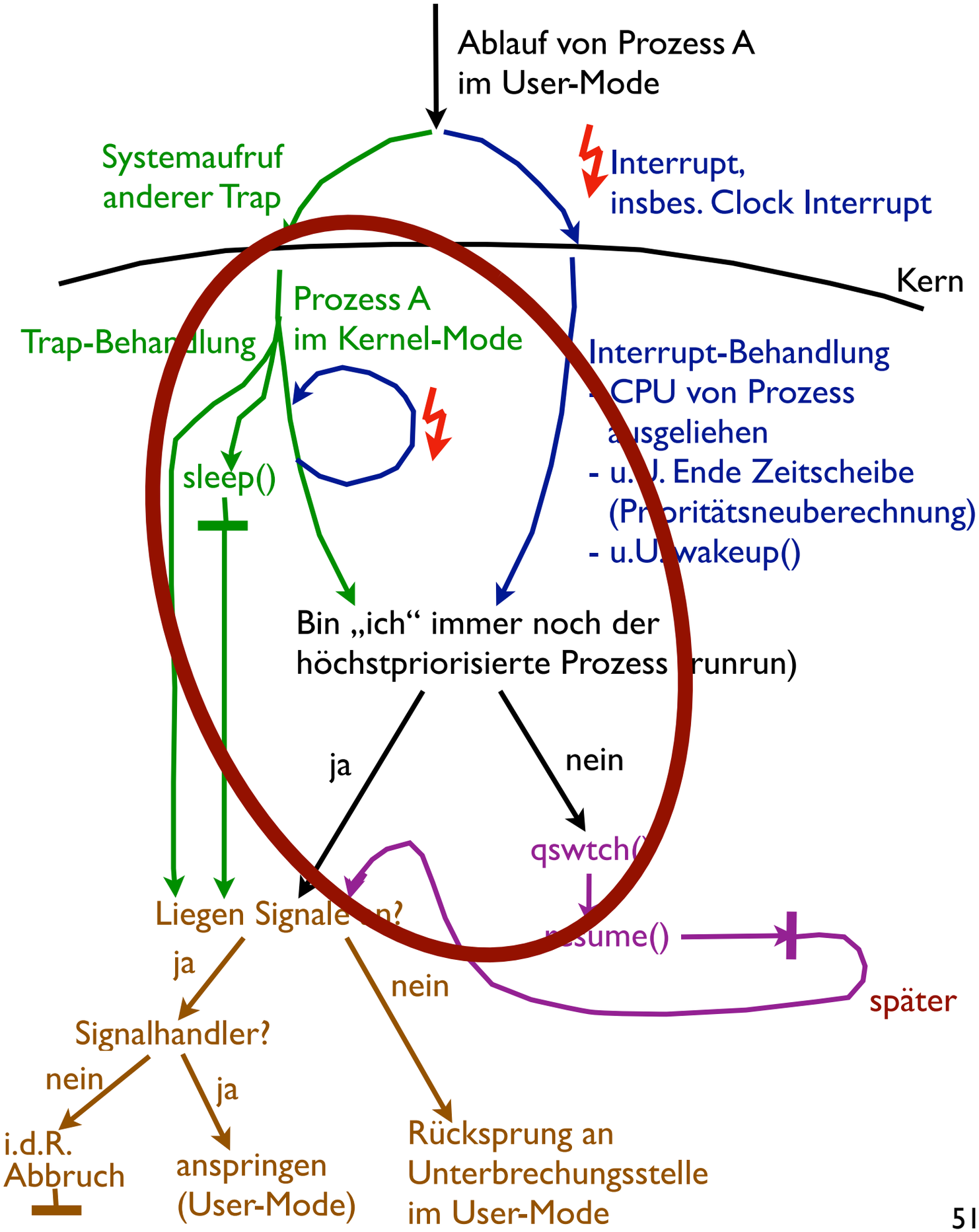
⇒ kein Problem, da Nicht-Präemption im Einprozessor-Unix

⇒ Kernel-Mode-Aktivitäten laufen durch (oder sleep())

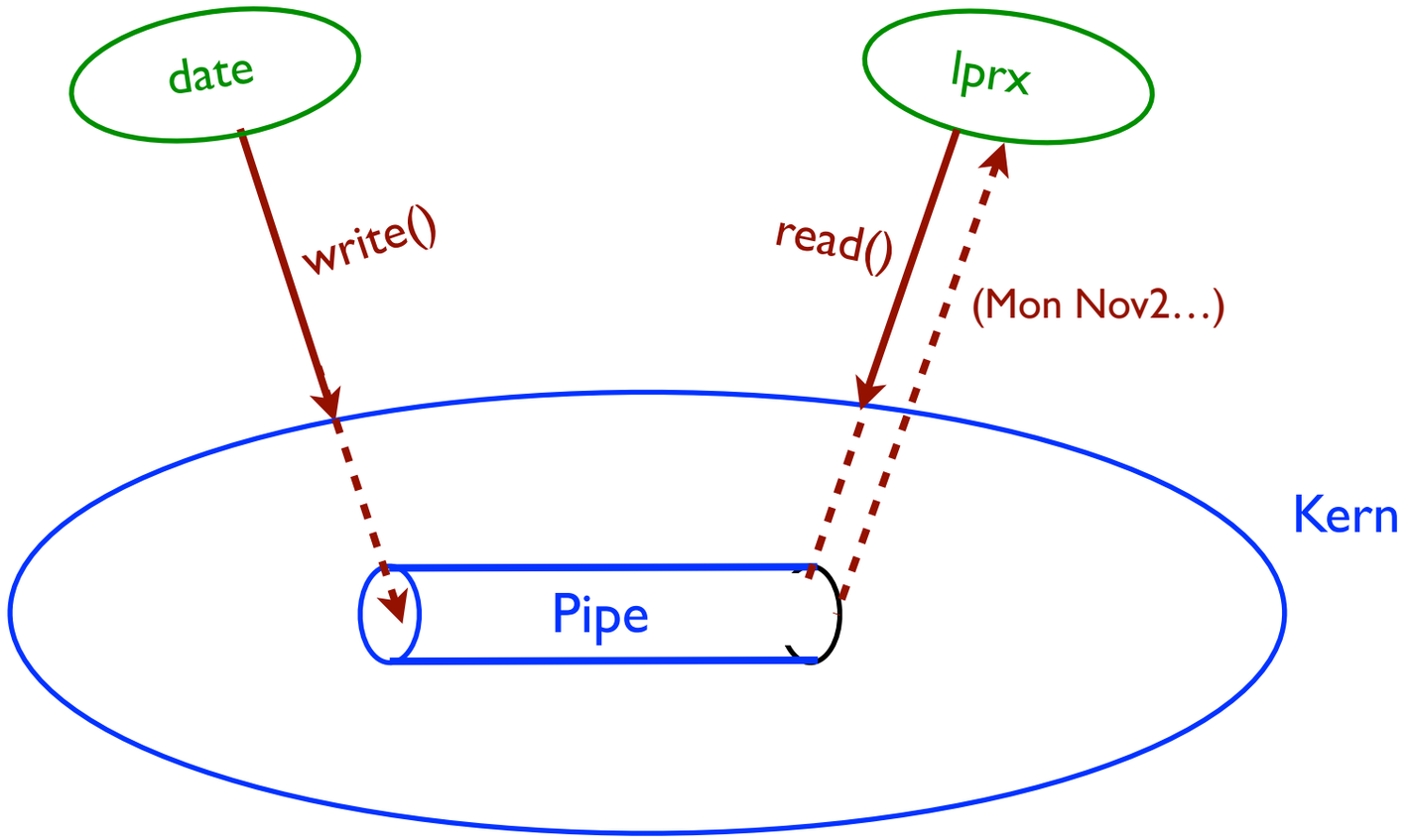
⇒ dann CPU abgeben

Einordnung einer Prozessumschaltung im Prozessablauf

(vereinfacht)



- Zurück zum Beispiel „Pipe“ …
  Verwaltung erfolgt im Kern:



⇒ Bei Einprozessor-Unix:
„count“ war durch Nicht-Präemption sowieso geschützt

Nebenläufigkeit in Unix

- Einprozessorsystem: CPU wird gemeinsam benutzt von
  - Prozessen im User-Mode
  - Prozessen im Kernel-Mode
  - Interrupthandlern

    (keine Prozesse, leihen sich CPU nur aus)
    ⇒ kein Prozesskontext, PID,...

⇒ relativ überschaubare Nebenläufigkeitsprobleme im Einprozessor-Unix

Nebenläufigkeit in Unix

- Einprozessorsystem: CPU wird gemeinsam benutzt von
  - Prozessen im User-Mode
  - Prozessen im Kernel-Mode
  - Interrupthandlern

    (keine Prozesse, leihen sich CPU nur aus)
    ⇒ kein Prozesskontext, PID,...

⇒ relativ überschaubare Nebenläufigkeitsprobleme im Einprozessor-Unix

- Zusätzliche Probleme durch Mehrprozessorsystem
  - Nicht-Präemption funktioniert nicht
  - Interrupts ausschließen reicht nicht

Fragen – Teil 4

- Was versteht man in einem Einprozessor-Unix unter *Nicht-Präemption*? Welche Auswirkung hat dies auf mögliche Nebenläufigkeitsprobleme?

Zusammenfassung

- Prozesserzeugung: fork() vs. exec()
- Prozesstermination: exit(), „Zombies“, Init
- Nebenläufigkeit: kritische Abschnitte
- Nebenläufigkeit in Unix

Prozessverwaltung 2 – Fragen

1. Skizziere kurz die Prozesserzeugung in Unix. Welche Rolle spielen die Systemaufrufe `fork()` und `exec()` dabei?
2. Wie erfährt ein Unix-Prozess, ob ein Kindprozess terminiert ist?
3. Kann ein Kindprozess weiterlaufen, wenn sein Vaterprozess bereits terminiert ist?
4. Wozu gibt es in Unix den Prozesszustand SZOMB („Zombie“)?
5. Beschreibe ein typisches Nebenläufigkeitsproblem.
6. Was versteht man in einem Einprozessor-Unix unter *Nicht-Präemption*? Welche Auswirkung hat dies auf mögliche Nebenläufigkeitsprobleme?