

ÜBUNG 2

Prozesse, Threads und Scheduling

Max Schrötter

schroetter@cs.uni-potsdam.de

Institute for Computational Science
University of Potsdam

08.11.2024

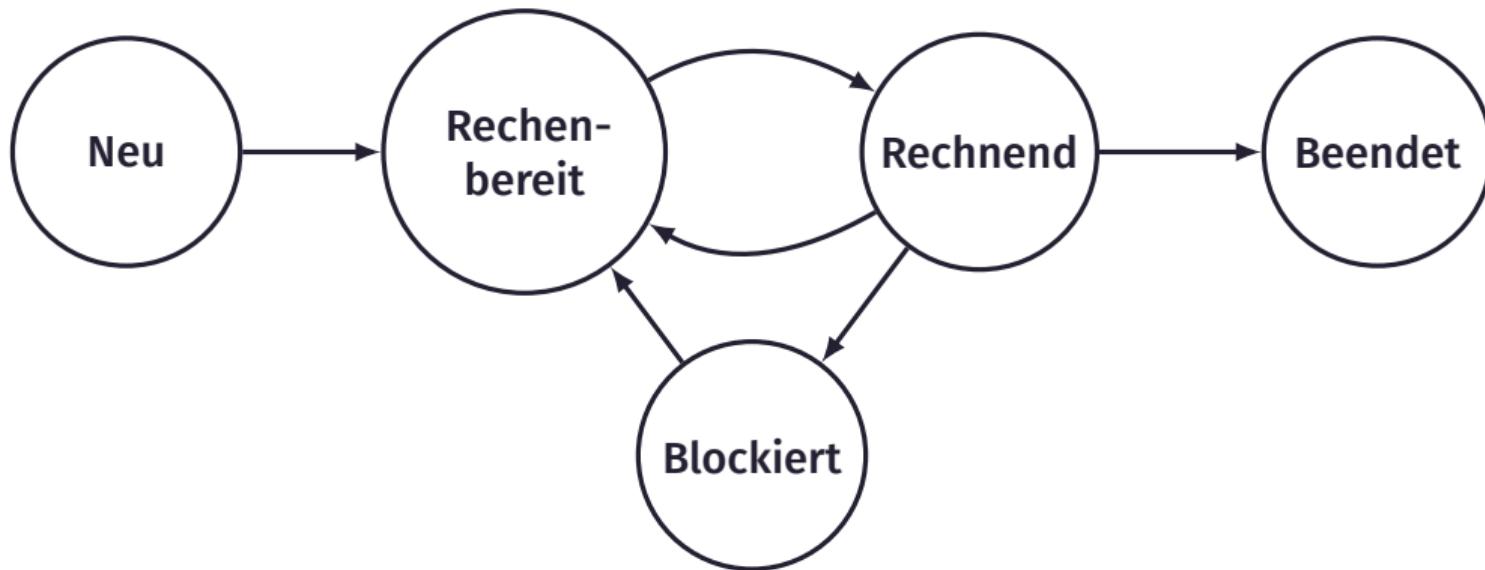


AGENDA

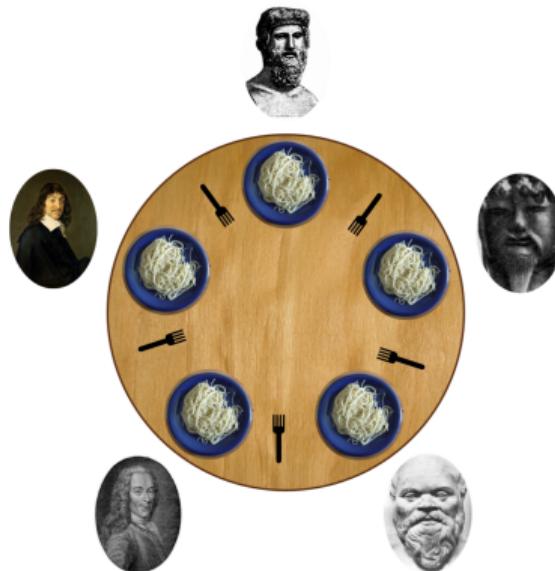
1. Prozesse & Prozesserzeugung
2. Scheduling



PROZESSZUSTÄNDE



ABNORMALE ZUSTÄNDE



Deadlock: alle Phil. greifen sich das rechte Stäbchen und **warten** darauf, dass das linke Stäbchen frei wird.

ABNORMALE ZUSTÄNDE



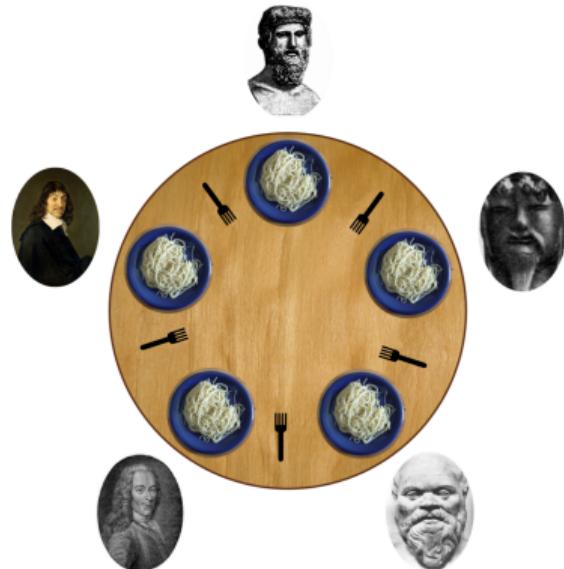
Livelock:

ABNORMALE ZUSTÄNDE



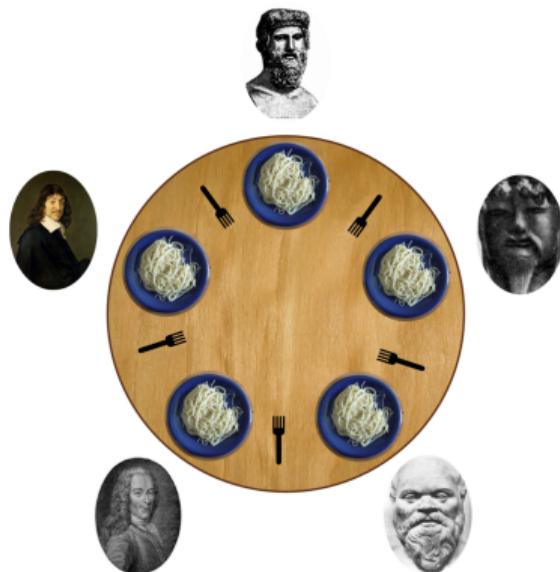
Livelock: alle Phil. greifen sich das rechte Stäbchen, wenn das Linke nicht vorhanden ist, legen sie es wieder hin und **versuchen es erneut**.

ABNORMALE ZUSTÄNDE



Starvation:

ABNORMALE ZUSTÄNDE



Starvation: Phil. prüft ständig, ob beide Stäbchen vorhanden sind. Sind beide Stäbchen für einen Phil. vorhanden, dann greift er beide und isst. Ein Phil. wird verhungern, wenn zu jedem Zeitpunkt mindestens einer seiner Nachbarn isst.

PID

- pid: Identifikator eines Prozesses

```
pid_t pid = getpid();
```

- ppid: Identifikator des Elternprozesses

```
pid_t ppid = getppid();
```

PID

- pid: Identifikator eines Prozesses
`pid_t pid = getpid();`
- ppid: Identifikator des Elternprozesses
`pid_t ppid = getppid();`
- es existieren **nur Syscalls** für die PID und PPID

PROZESSBAUM

```
$ps -ef --forest
UID      PID  PPID  C STIME TTY          TIME CMD
mschroe+ 144583     1  0 14:35 ?
mschroe+ 144593 144583  0 14:35 ?
mschroe+ 144595 144593  1 14:35 pts/7  00:00:03 /usr/bin/zsh
mschroe+ 145281 144595  9 14:37 pts/7  00:00:06
mschroe+ 145290 145281  0 14:37 pts/7  00:00:00
mschroe+ 145315 145290  1 14:37 pts/7  00:00:01
mschroe+ 145291 145281  0 14:37 pts/7  00:00:00
mschroe+ 145293 145291  0 14:37 pts/7  00:00:00
mschroe+ 145321 145293  0 14:37 pts/7  00:00:00
mschroe+ 145346 145293  4 14:37 pts/7  00:00:02
mschroe+ 145318 145281  1 14:37 pts/7  00:00:00
mschroe+ 145665 145281  0 14:37 pts/7  00:00:00
mschroe+ 146017 144595 10 14:37 pts/7  00:00:05
mschroe+ 146103 146017  0 14:37 pts/7  00:00:00
mschroe+ 146122 146017  0 14:37 pts/7  00:00:00
mschroe+ 146271 146017  0 14:37 pts/7  00:00:00
mschroe+ 146317 144595  0 14:38 pts/7  00:00:00
mschroe+ 146320 146317  4 14:38 ?    00:00:01 nvim test.c
mschroe+ 146335 146320  8 14:38 ?    00:00:02   nvim --embed test.c
                                         \ node
→ /home/mschroetter/.local/share/nvim/lazy/copilot.lua/copilot/index.js$
```

PROZESSBAUM

```
$ps -ef --forest
UID      PID  PPID  C STIME TTY          TIME CMD
mschroe+ 144583     1  0 14:35 ?
mschroe+ 144593 144583  0 14:35 ?
mschroe+ 144595 144593  1 14:35 pts/7  00:00:03 /usr/bin/zsh
mschroe+ 145281 144595  9 14:37 pts/7  00:00:06
mschroe+ 145290 145281  0 14:37 pts/7  00:00:00
mschroe+ 145315 145290  1 14:37 pts/7  00:00:01
mschroe+ 145291 145281  0 14:37 pts/7  00:00:00
mschroe+ 145293 145291  0 14:37 pts/7  00:00:00
mschroe+ 145321 145293  0 14:37 pts/7  00:00:00
mschroe+ 145346 145293  4 14:37 pts/7  00:00:02
mschroe+ 145318 145281  1 14:37 pts/7  00:00:00
mschroe+ 145665 145281  0 14:37 pts/7  00:00:00
mschroe+ 146017 144595 10 14:37 pts/7  00:00:05
mschroe+ 146103 146017  0 14:37 pts/7  00:00:00
mschroe+ 146122 146017  0 14:37 pts/7  00:00:00
mschroe+ 146271 146017  0 14:37 pts/7  00:00:00
mschroe+ 146317 144595  0 14:38 pts/7  00:00:00
mschroe+ 146320 146317  4 14:38 ?   00:00:01 nvim test.c
mschroe+ 146335 146320  8 14:38 ?   00:00:02   nvim --embed test.c
                                                \ node
→ /home/mschroetter/.local/share/nvim/lazy/copilot.lua/copilot/index.js$
```

→ Prozessbeziehungen sind “tree-like”

PROZESSZUSTÄNDE

PROCESS STATE CODES

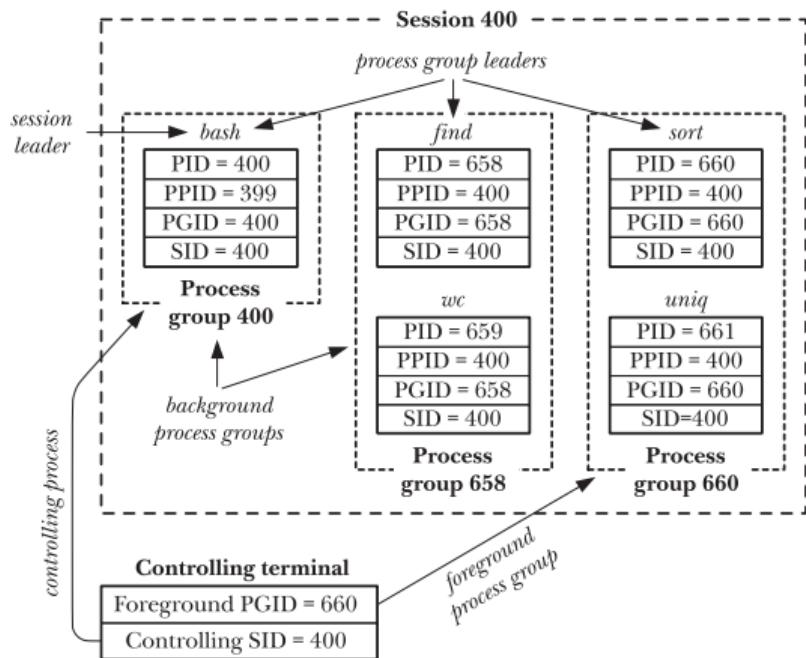
Here are the different values that the s, stat and state output specifiers (header "STAT" or "S") will
→ display to describe the state of a process:

- D uninterrupted sleep (usually I/O)
- I idle kernel thread
- R running or runnable (on run queue)
- S interruptible sleep (waiting **for** an event to **complete**)
- T stopped by job control signal
- t stopped by debugger during the tracing
- W paging (not valid since Linux 2.6)
- X dead (should never be seen)
- Z defunct ("zombie") process, terminated but not reaped by its parent

For BSD formats and when the stat keyword is used, additional characters may be displayed:

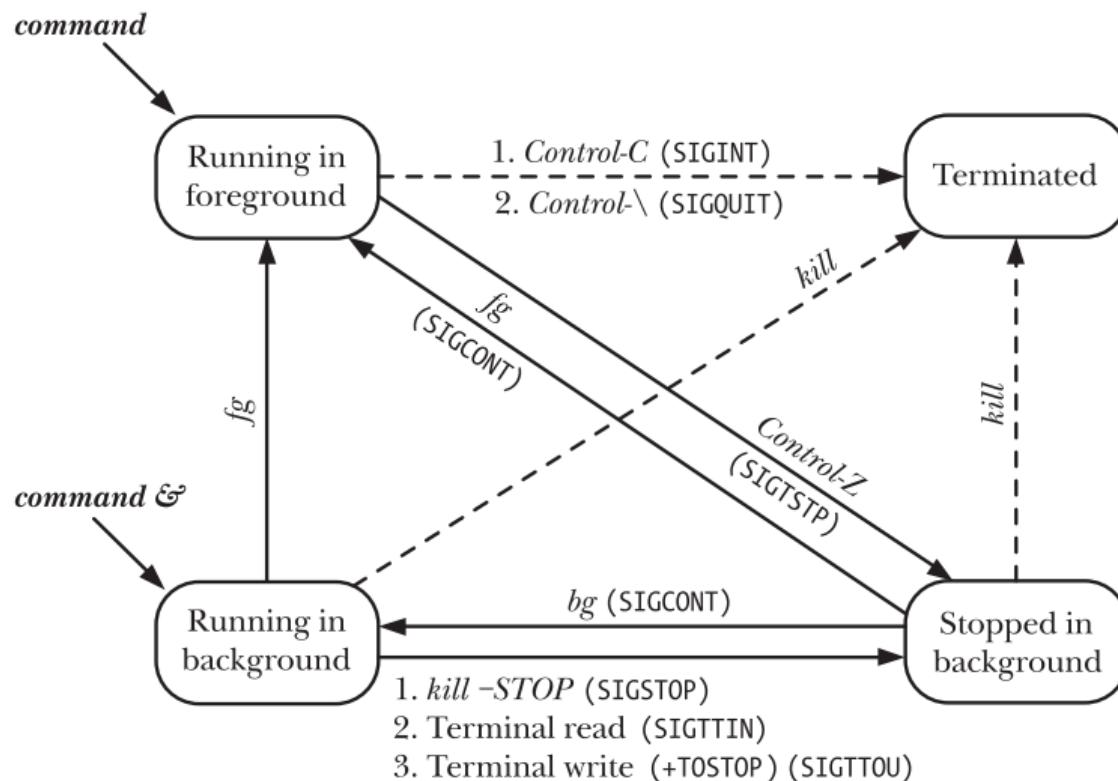
- < high-priority (not nice to other users)
- N low-priority (nice to other users)
- L has pages locked into memory (**for** real-time and custom I/O)
- s is a session leader
- l is multi-threaded (using CLONE_THREAD, like NPTL pthreads **do**)
- + is **in** the foreground process group

JOB CONTROL



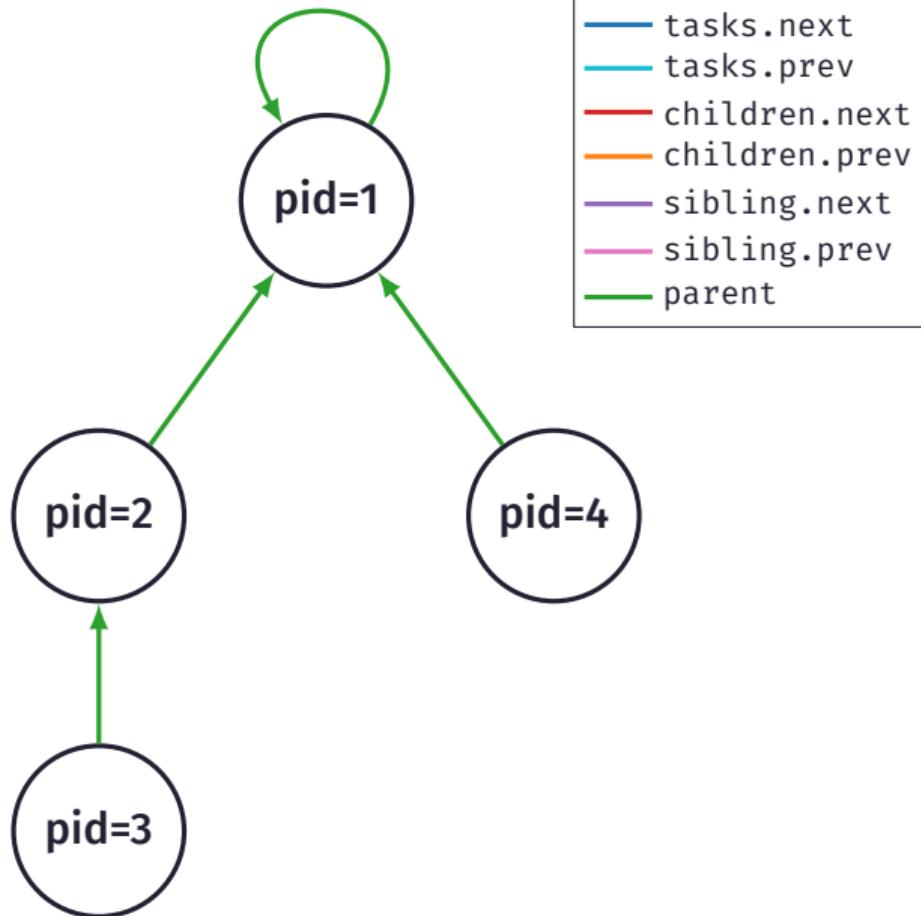
```
$ echo $$  
400  
$ find / 2> /dev/null | wc -l &  
[1] 659  
$ sort < longlist | uniq -c
```

PROCESS STATES IN JOB CONTROL



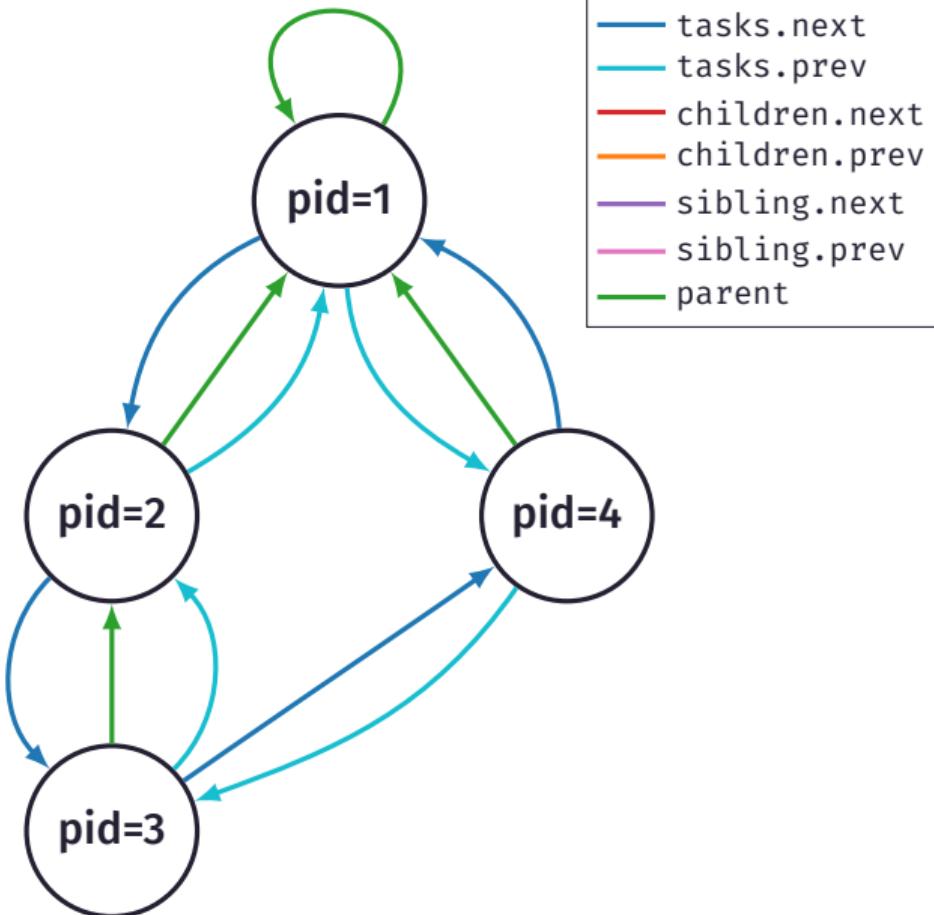
LINUX PROZESSTABELLE

```
struct task_struct {  
    ...  
    struct list_head tasks;  
    pid_t pid;  
    struct task_struct *parent;  
    struct list_head children;  
    struct list_head sibling;  
    ...  
}  
  
struct list_head {  
    struct list_head *next, *prev;  
};
```



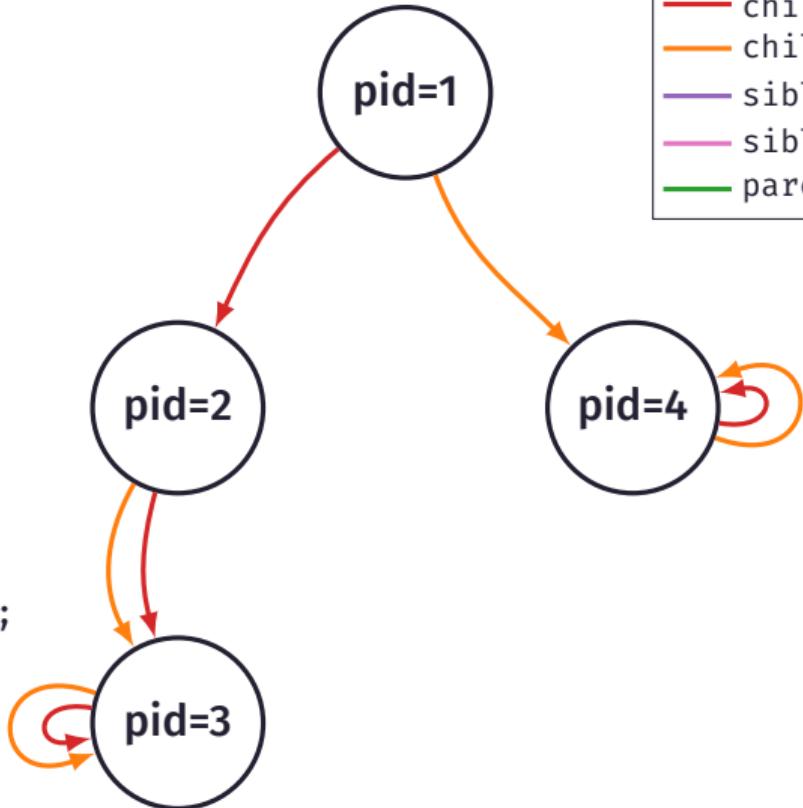
LINUX PROZESSTABELLE

```
struct task_struct {  
    ...  
    struct list_head tasks;  
    pid_t pid;  
    struct task_struct *parent;  
    struct list_head children;  
    struct list_head sibling;  
    ...  
}  
  
struct list_head {  
    struct list_head *next, *prev;  
};
```



LINUX PROZESSTABELLE

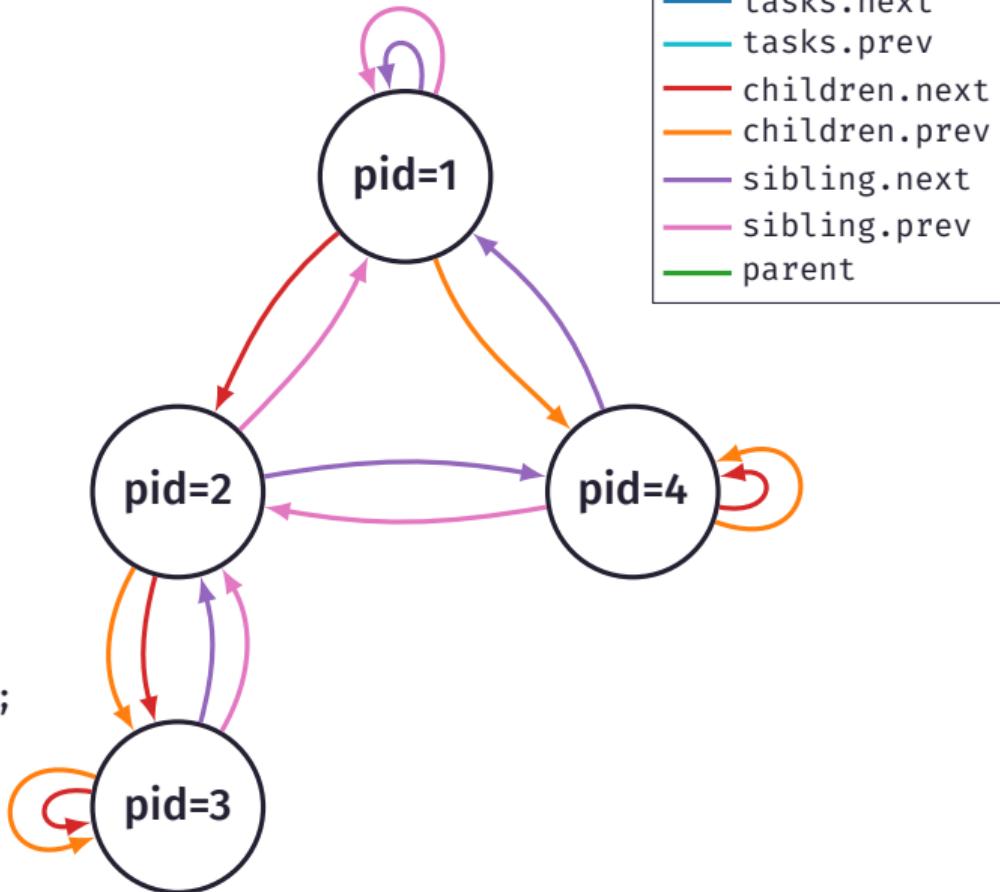
```
struct task_struct {  
    ...  
    struct list_head tasks;  
    pid_t pid;  
    struct task_struct *parent;  
    struct list_head children;  
    struct list_head sibling;  
    ...  
}  
  
struct list_head {  
    struct list_head *next, *prev;  
};
```



—	tasks.next
—	tasks.prev
—	children.next
—	children.prev
—	sibling.next
—	sibling.prev
—	parent

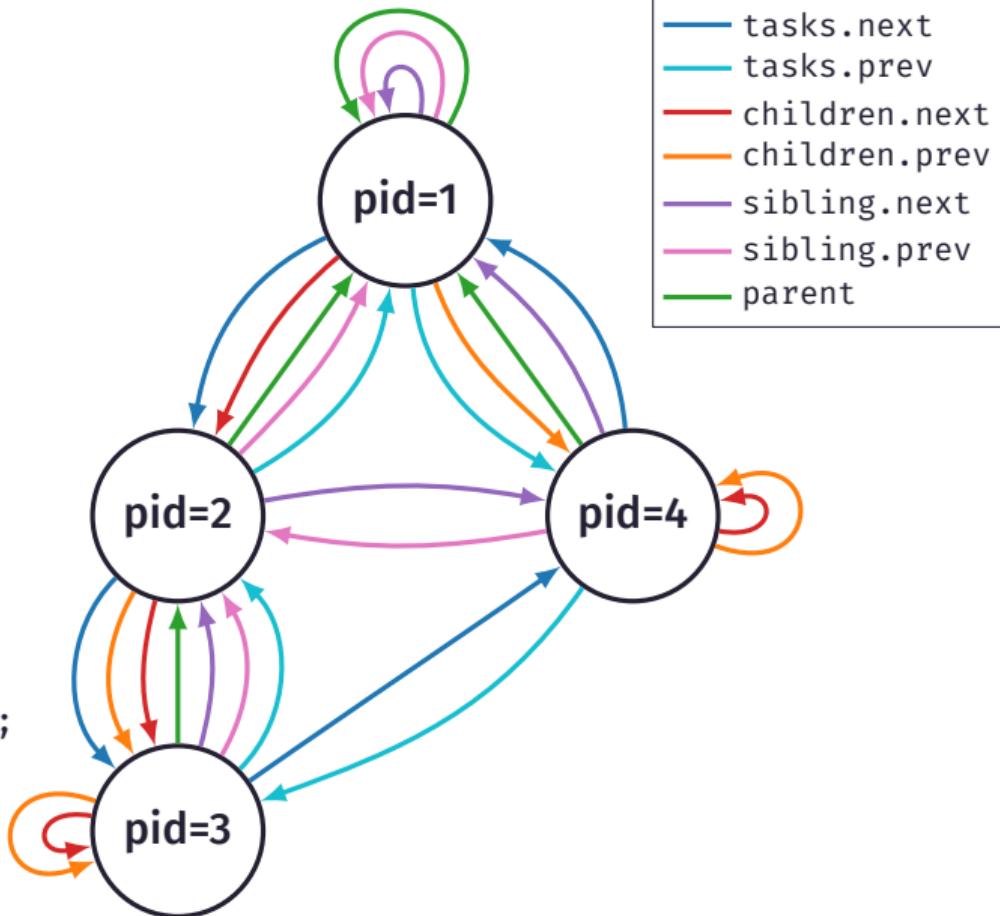
LINUX PROZESSTABELLE

```
struct task_struct {  
    ...  
    struct list_head tasks;  
    pid_t pid;  
    struct task_struct *parent;  
    struct list_head children;  
    struct list_head sibling;  
    ...  
}  
  
struct list_head {  
    struct list_head *next, *prev;  
};
```

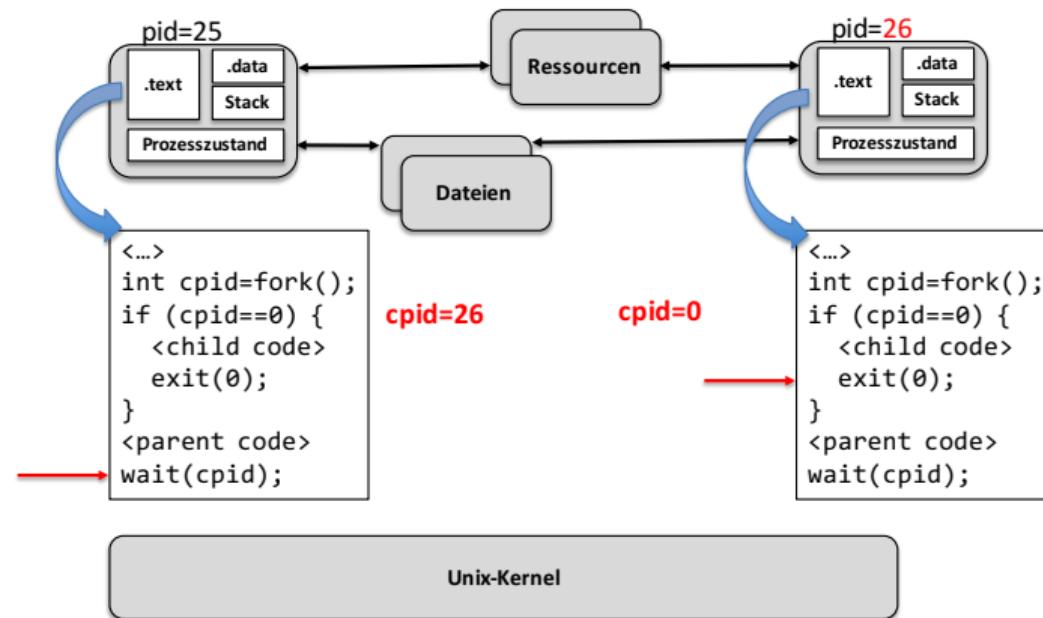


LINUX PROZESSTABELLE

```
struct task_struct {  
    ...  
    struct list_head tasks;  
    pid_t pid;  
    struct task_struct *parent;  
    struct list_head children;  
    struct list_head sibling;  
    ...  
}  
  
struct list_head {  
    struct list_head *next, *prev;  
};
```



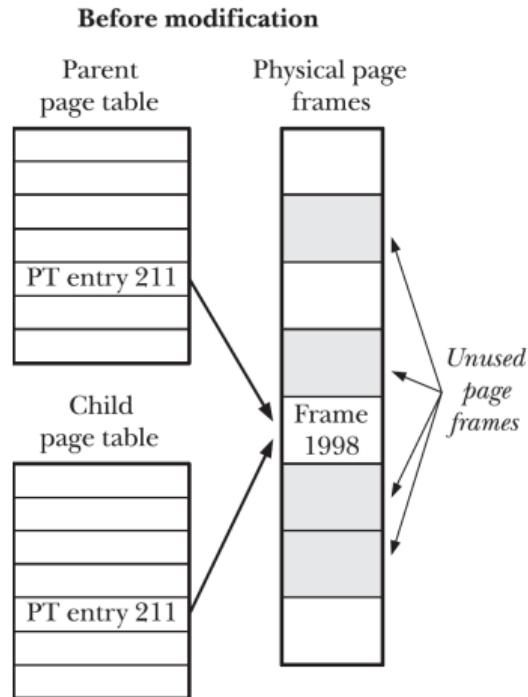
ABLAUF VON FORK



pid25 wartet auf Ende von pid26, **pid26 führt exit(0) aus und beendet sich**

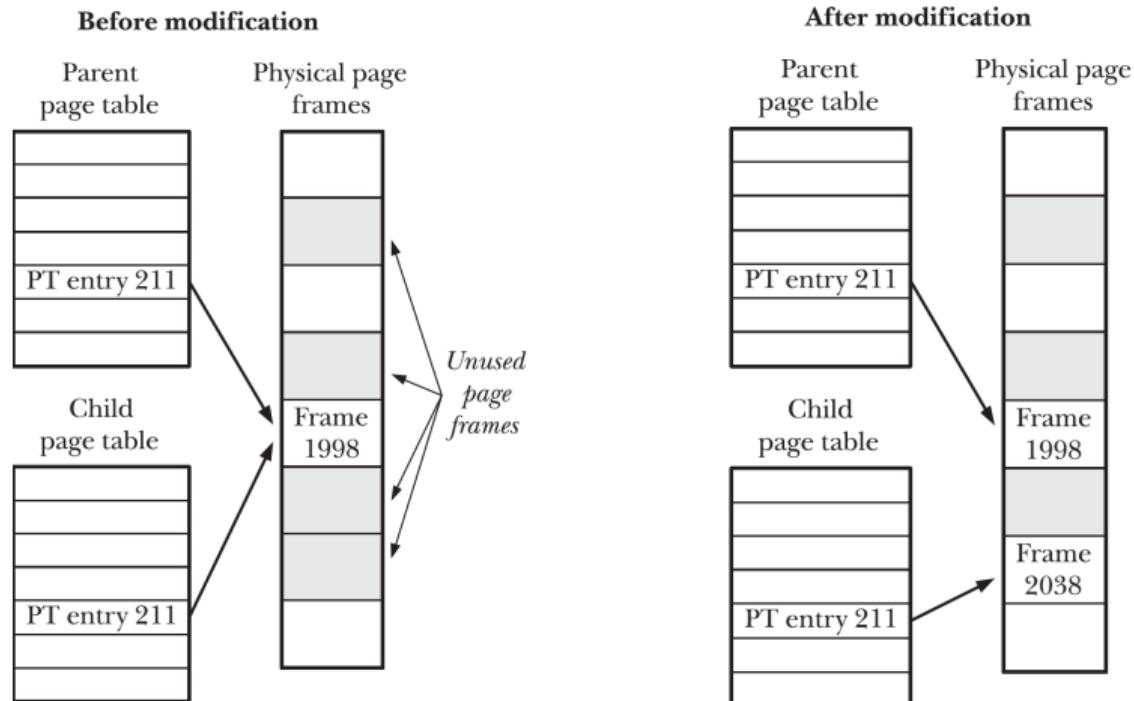
MEMORY AFTER FORK

- `fork()` kopiert den **Adressraum** des Elternprozesses
- Copy-on-Write Optimierung **reduziert** Prozess-erzeugungszeit und Speicherverbrauch
- Adressraum ist kopiert nach **Modifizierung**



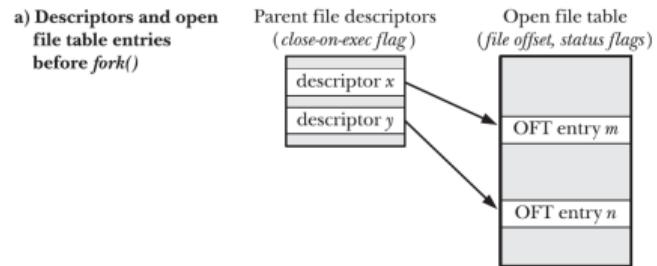
MEMORY AFTER FORK

- `fork()` kopiert den **Adressraum** des Elternprozesses
- Copy-on-Write Optimierung **reduziert** Prozess-erzeugungszeit und Speicherverbrauch
- Adressraum ist kopiert nach **Modifizierung**



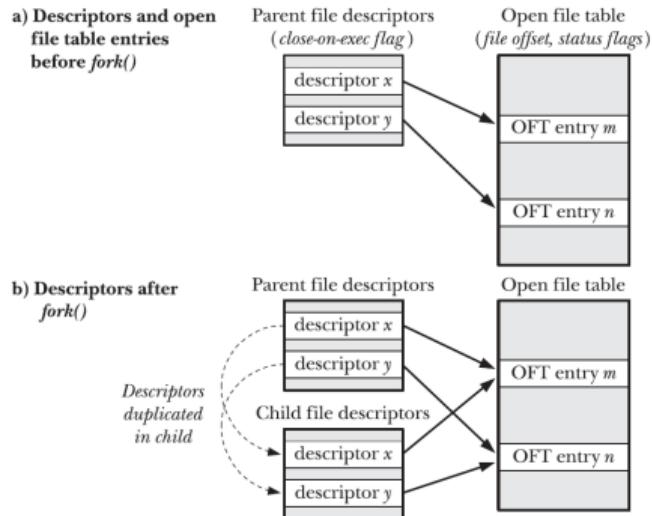
FILES AFTER FORK

- `fork()` kopiert **alle** offenen File-Descriptoren (Dateireferenzen)
- inkludiert **Datei-Offset**
- schließen eines File-Descriptor im Kind hat **keine Auswirkung** auf den Elternprozess (und andersherum)



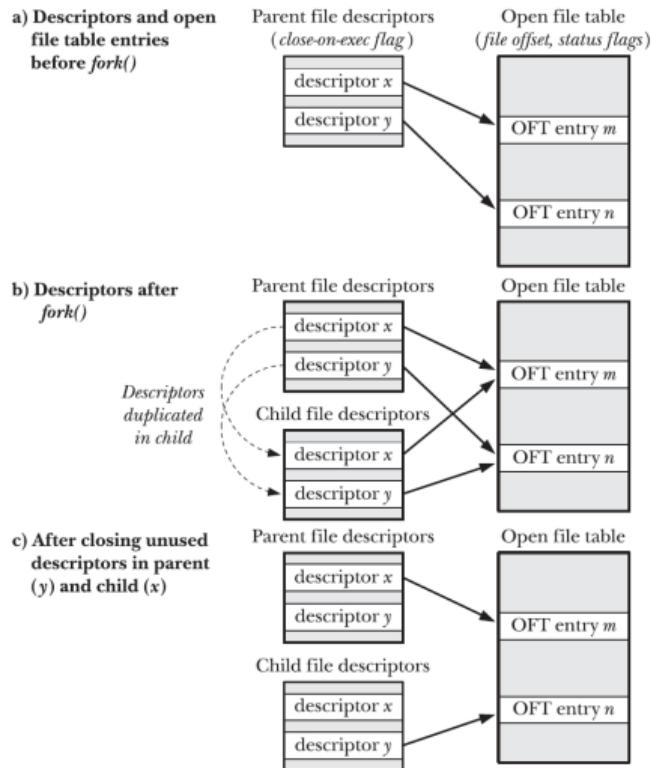
FILES AFTER FORK

- `fork()` kopiert alle offenen File-Descriptoren (Dateireferenzen)
- inkludiert Datei-Offset
- schließen eines File-Descriptor im Kind hat keine Auswirkung auf den Elternprozess (und andersherum)

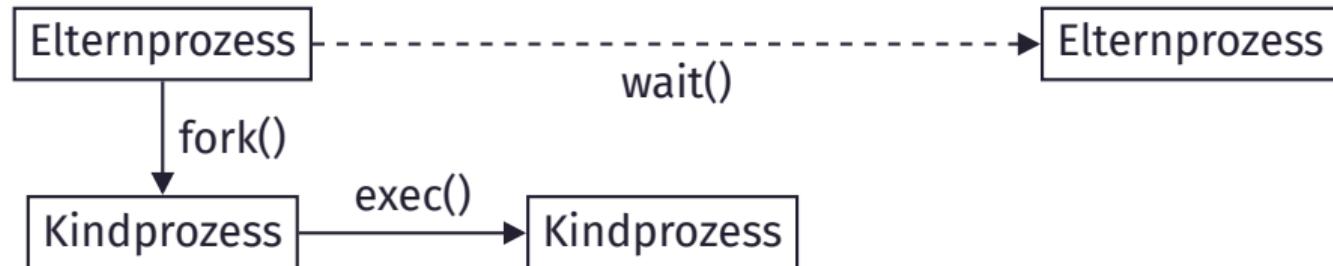


FILES AFTER FORK

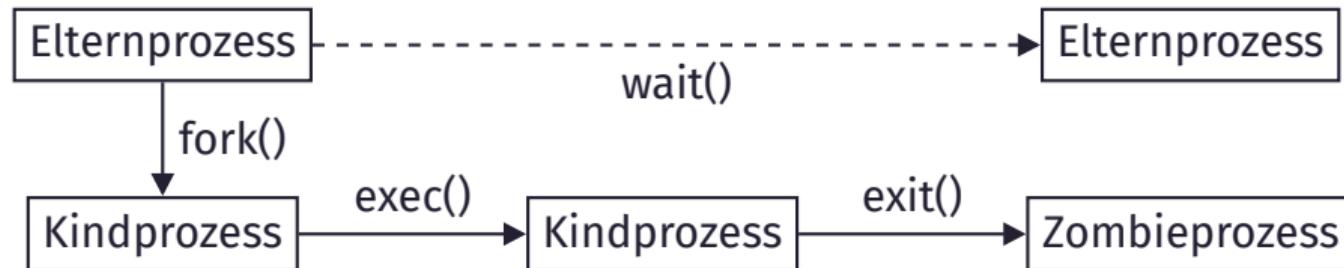
- `fork()` kopiert alle offenen File-Descriptoren (Dateireferenzen)
- inkludiert Datei-Offset
- schließen eines File-Descriptor im Kind hat keine Auswirkung auf den Elternprozess (und andersherum)



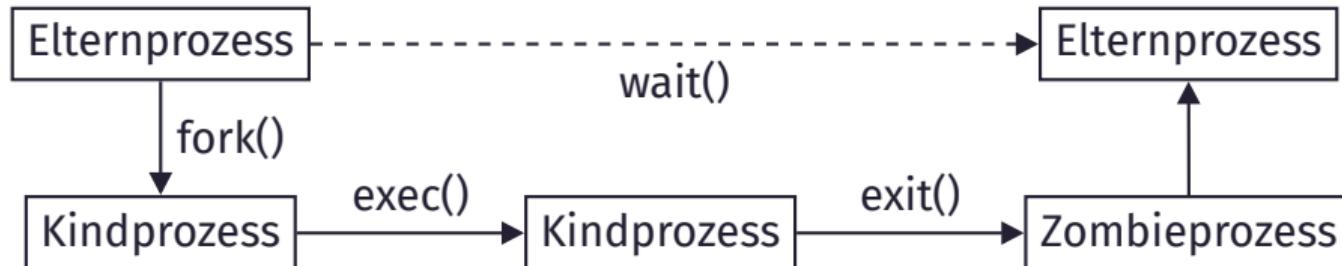
PROZESSZYKLUS



PROZESSZYKLUS



PROZESSZYKLUS



Funktion von `exec()`:

- **ersetzen des Speicherinhaltes** des aktuellen Prozesses durch Sektionen anderer ausführbarer Datei (Parameter von `exec`)
- Methode **neue Programme** zu starten, denn `fork()` erstellt nur **Kopie** des aufgerufenen Prozesses

ZOMBIES



- wenn `wait()` oder `waitpid()` vom Elternprozess nicht aufgerufen wird, verweilen Kinder in einem sogenannten „Zombie“-Zustand

ZOMBIES



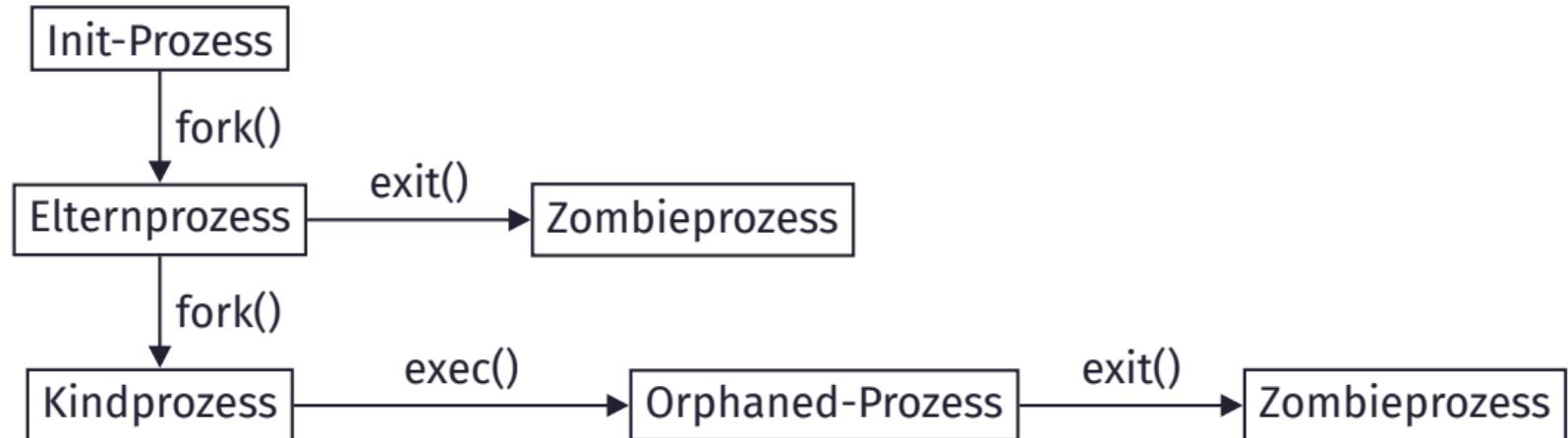
- wenn **wait()** oder **waitpid()** vom Elternprozess nicht aufgerufen wird, verweilen Kinder in einem sogenannten „Zombie“-Zustand
- **wait()** muss aufgerufen werden, um die Ressourcen des Kindprozesses freizugeben

ZOMBIES

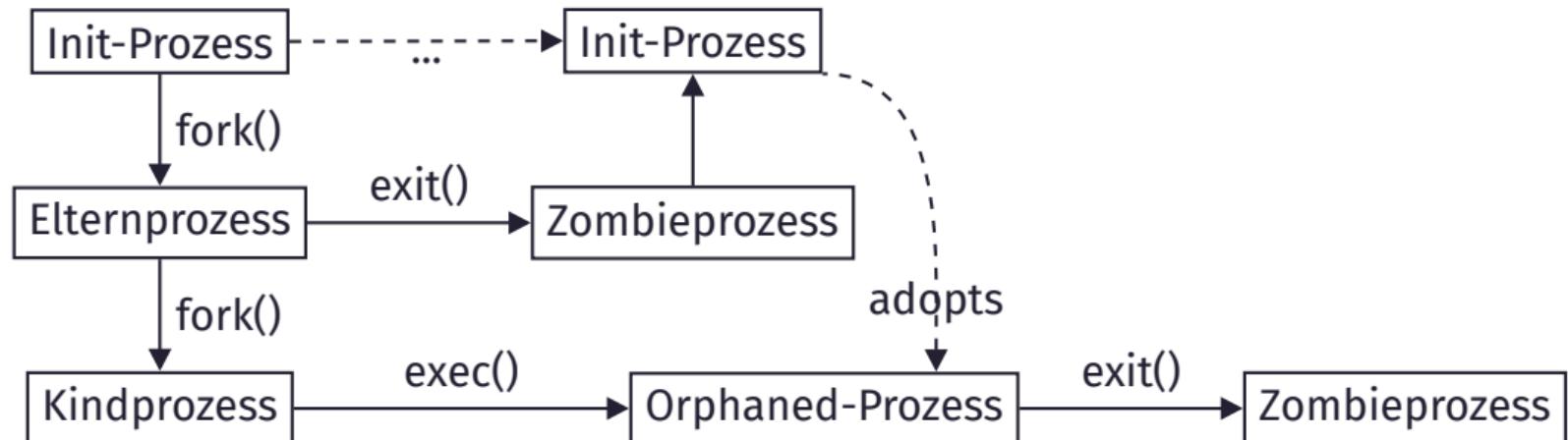


- wenn `wait()` oder `waitpid()` vom Elternprozess nicht aufgerufen wird, verweilen Kinder in einem sogenannten „Zombie“-Zustand
- `wait()` muss aufgerufen werden, um die Ressourcen des Kindprozesses freizugeben
- Was passiert, wenn der Elternprozess vor dem Kind beendet wird?

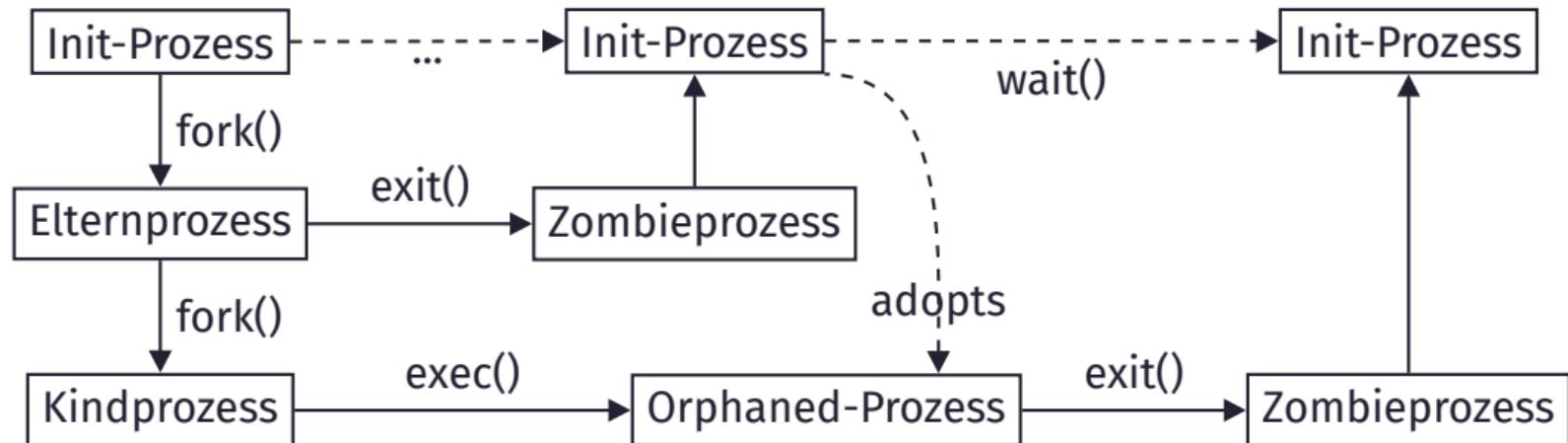
ORPHANED-PROZESSE



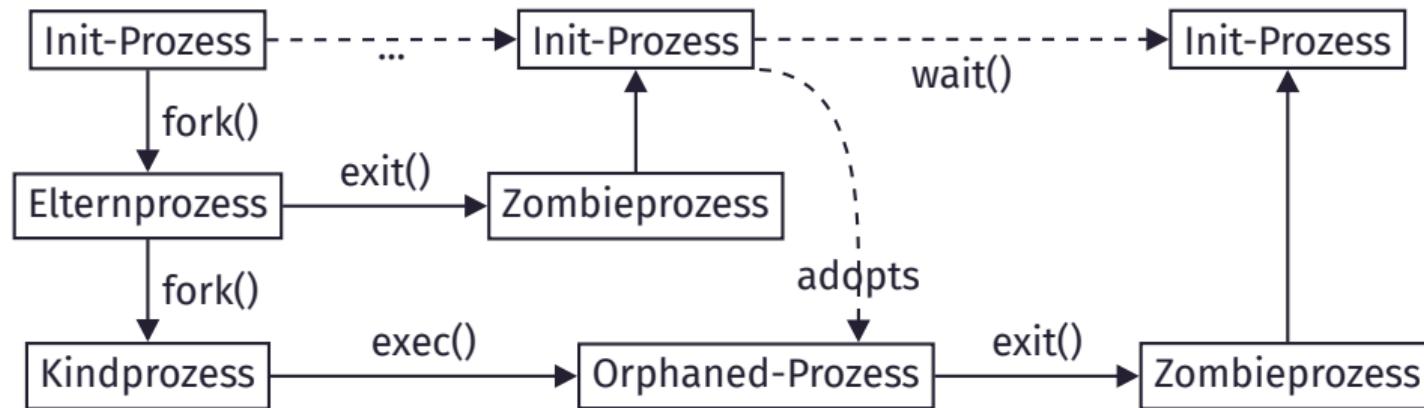
ORPHANED-PROZESSE



ORPHANED-PROZESSE

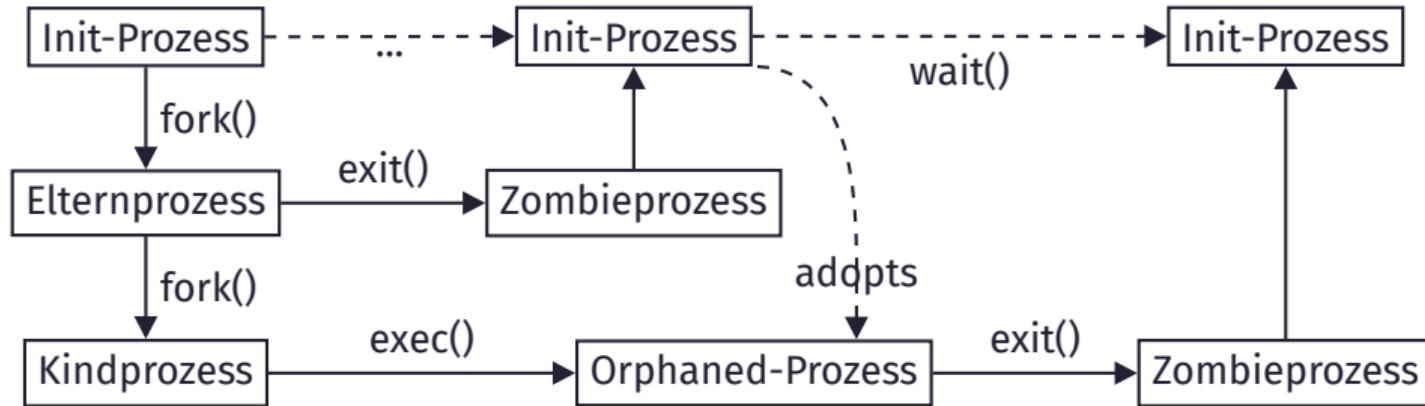


ORPHANED-PROZESSE



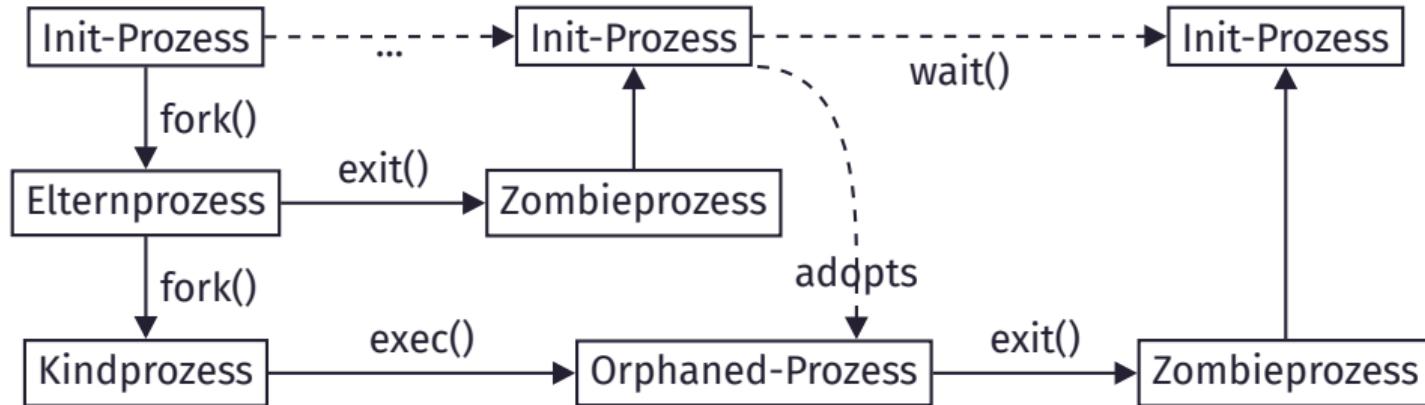
- Orphaned-Prozesse werden vom Init-Prozess „adoptiert“

ORPHANED-PROZESSE



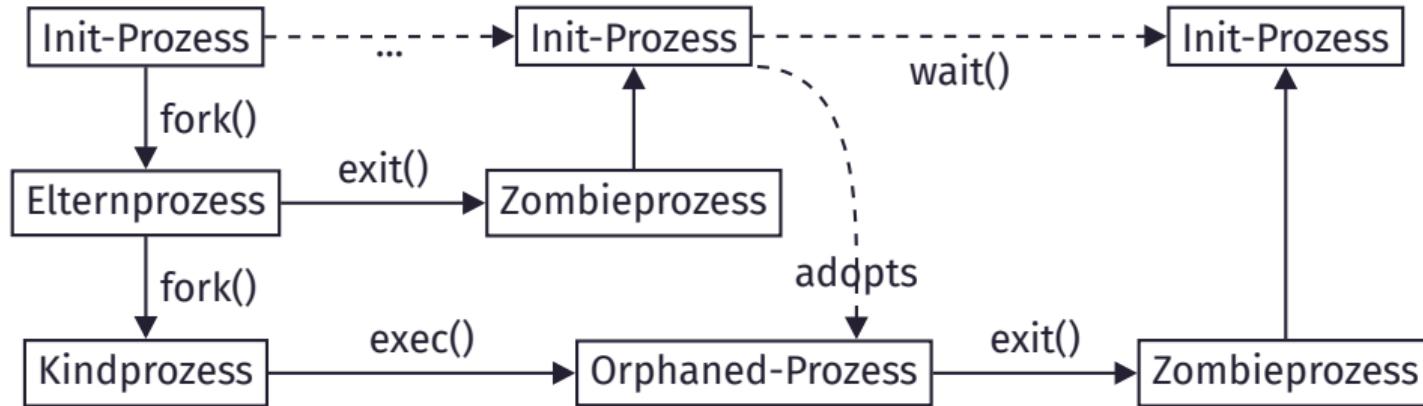
- Orphaned-Prozesse werden vom Init-Prozess „adoptiert“
- Prozessergebnisse/Exit-Codes können **nicht evaluiert** werden (**Fehler** können nicht erkannt werden)

ORPHANED-PROZESSE



- Orphaned-Prozesse werden vom Init-Prozess „adoptiert“
- Prozessergebnisse/Exit-Codes können **nicht evaluiert** werden (**Fehler** können nicht erkannt werden)
- **Eltern sollten immer auf ihre Kinder warten!**

ORPHANED-PROZESSE



- Orphaned-Prozesse werden vom Init-Prozess „adoptiert“
- Prozessergebnisse/Exit-Codes können **nicht evaluiert** werden (Fehler können nicht erkannt werden)
- **Eltern sollten immer auf ihre Kinder warten!**
- aufrufen bei unserem Beispiel mithilfe von **wait(NULL)**, denn Informationen sind nicht relevant → siehe **man wait**

AUFGABE 1: PROZESSERZEUGUNG

```
#include <stdio.h>
#include <time.h>
#include "animal.h"

int main() {
    char *words[] = {"tiger", "river", "otter", "coffee",
                     "walrus", "router", "yak", "algorithm",
                     "dog", "cat", "math", "bike",
                     "elephant", "table", "guitar", "piano",
                     "lion", "desk", "window", "book",
                     "fish", "banana", "computer", "phone",
                     "car", "take", "monkey", "cash", NULL};

    for (int i = 0; words[i] != NULL; i++) {
        printf("%s:\t%lf\n", words[i], animal_probability(words[i]));
    }
}
```

AUFGABE 1: PROZESSERZEUGUNG

- Nutzen sie `fork()` !!!
- Achten sie darauf, dass Kindprozesse terminieren und nicht Code nach dem `if/else` Block ausführen! siehe man `3 exit`

Kein `_exit()` oder `_Exit()` verwenden

- Warten sie im Elternprozess auf die Kindprozesse!
- `fork()` kann fehlschlagen! Betriebssysteme können die Anzahl an Prozessen limitieren. Siehe cgroups in Übung 3.

→Fehlerbehandlung nicht vergessen!

AGENDA

1. Prozesse & Prozesserzeugung
2. Scheduling



SCHEDULING ALGORITHMEN

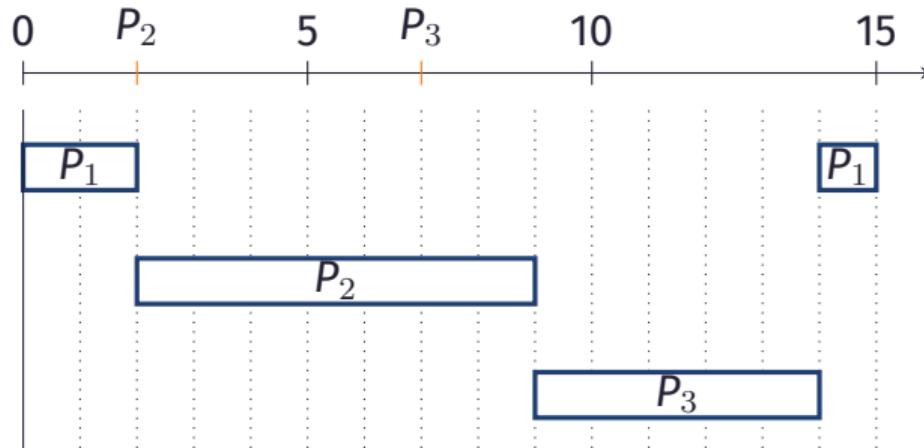
In einem Rechensystem liegen die folgenden Tasks vor:

Prozess	Ankunftszeit	Rechenzeit	Priorität
P_1	0	3	1
P_2	2	7	3
P_3	7	5	2

1. Erstellen Sie ein Gantt-Diagramm für einen **preemptive Scheduling-Algorithmus mit fixed priorities**.
2. Berechnen Sie die **mittlere Wartezeit**, die **mittlere Verweildauer** sowie die **mittlere Antwortzeit**.

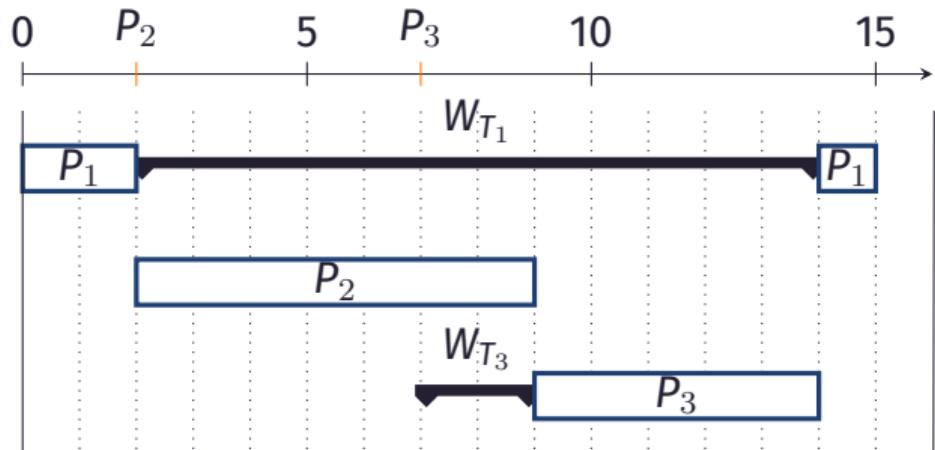
GANNT-DIAGRAMM

Prozess	Ankunftszeit	Rechenzeit	Priorität
P_1	0	3	1
P_2	2	7	3
P_3	7	5	2



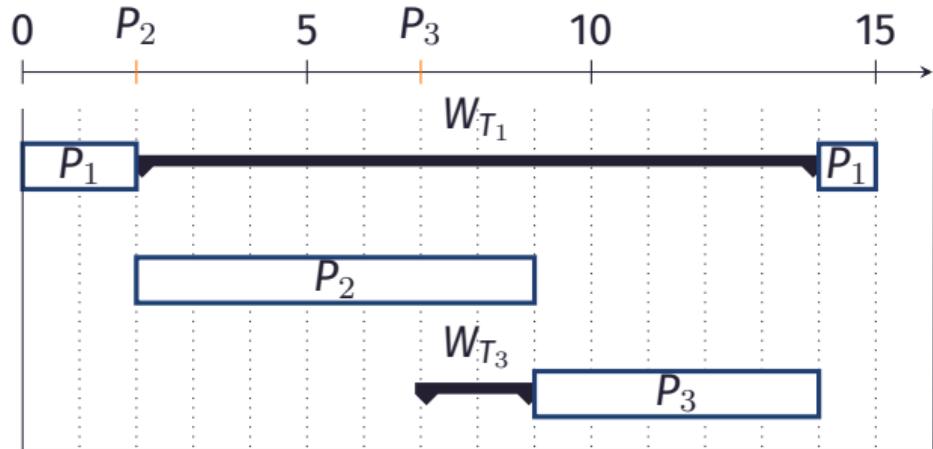
MITTLERE WARTEZEIT

Prozess	Ankunftszeit	Rechenzeit	Priorität
P_1	0	3	1
P_2	2	7	3
P_3	7	5	2



MITTLERE WARTEZEIT

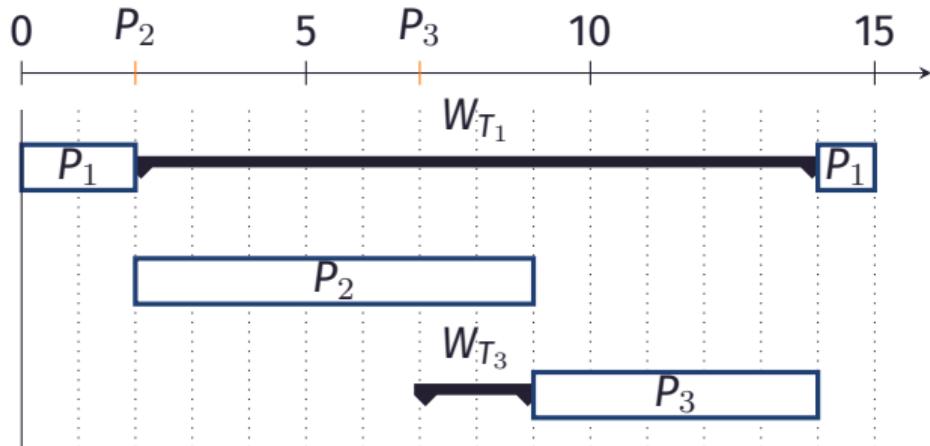
Prozess	Ankunftszeit	Rechenzeit	Priorität
P_1	0	3	1
P_2	2	7	3
P_3	7	5	2



$$\begin{aligned}
 W_T &= \frac{W_{T_1} + W_{T_2} + W_{T_3}}{3} \\
 &= \frac{12 + 0 + 2}{3} \\
 &= \frac{14}{3}
 \end{aligned}$$

MITTLERE WARTEZEIT

Prozess	Ankunftszeit	Rechenzeit	Priorität
P_1	0	3	1
P_2	2	7	3
P_3	7	5	2

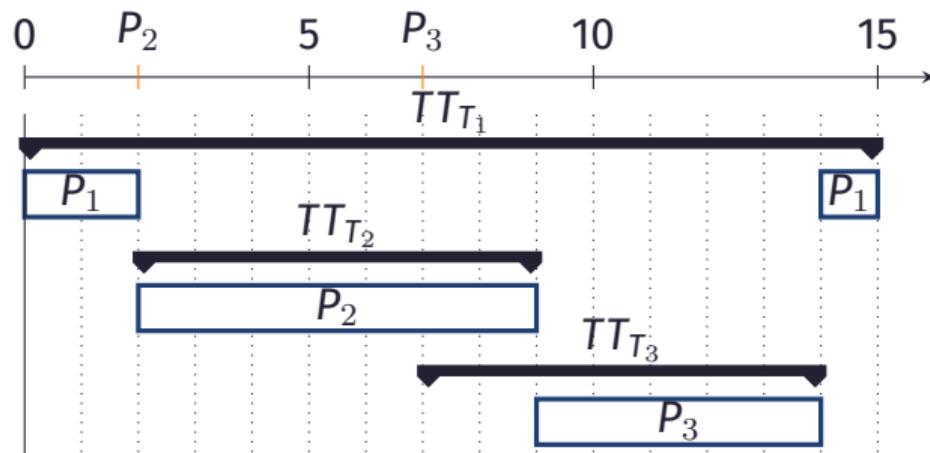


$$\begin{aligned} W_T &= \frac{W_{T_1} + W_{T_2} + W_{T_3}}{3} \\ &= \frac{12 + 0 + 2}{3} \\ &= \frac{14}{3} \end{aligned}$$

Welche Formel kann für die Wartezeit W_{T_i} angegeben werden?

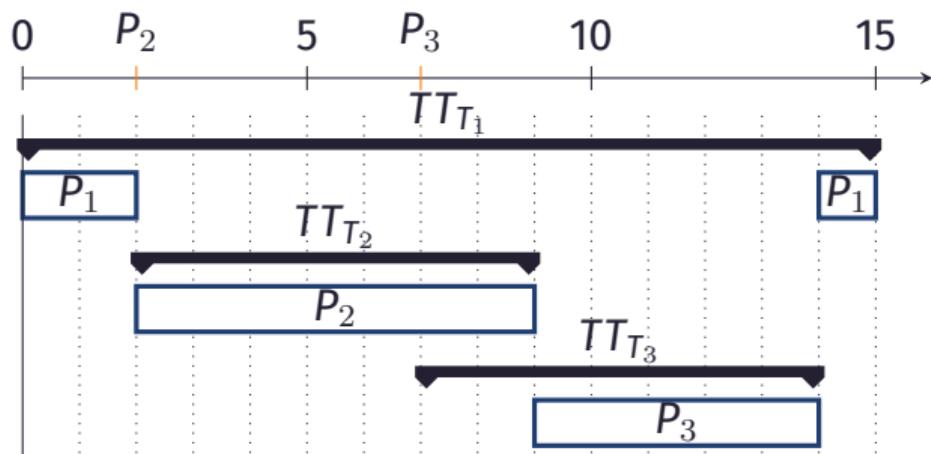
MITTLERE VERWEILDAUER

Prozess	Ankunftszeit	Rechenzeit	Priorität
P_1	0	3	1
P_2	2	7	3
P_3	7	5	2



MITTLERE VERWEILDAUER

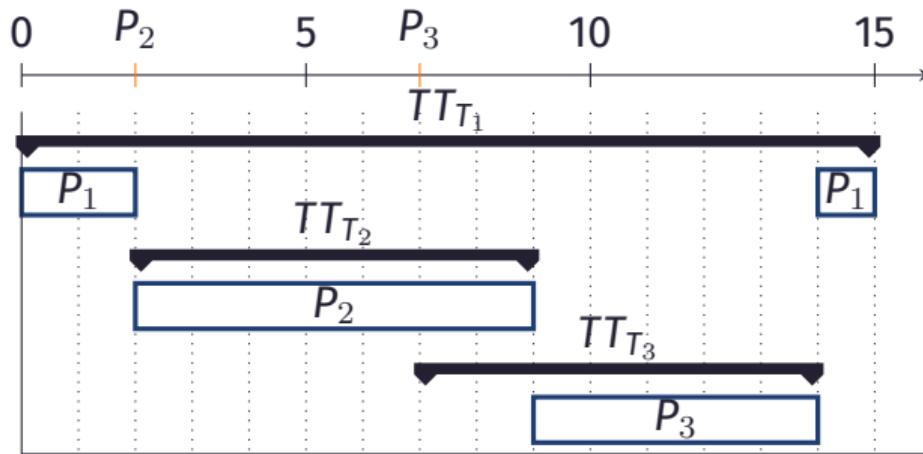
Prozess	Ankunftszeit	Rechenzeit	Priorität
P_1	0	3	1
P_2	2	7	3
P_3	7	5	2



$$\begin{aligned}
 T_T &= \frac{TT_{T_1} + TT_{T_2} + TT_{T_3}}{3} \\
 &= \frac{15 + 7 + 7}{3} \\
 &= \frac{29}{3}
 \end{aligned}$$

MITTLERE VERWEILDAUER

Prozess	Ankunftszeit	Rechenzeit	Priorität
P_1	0	3	1
P_2	2	7	3
P_3	7	5	2

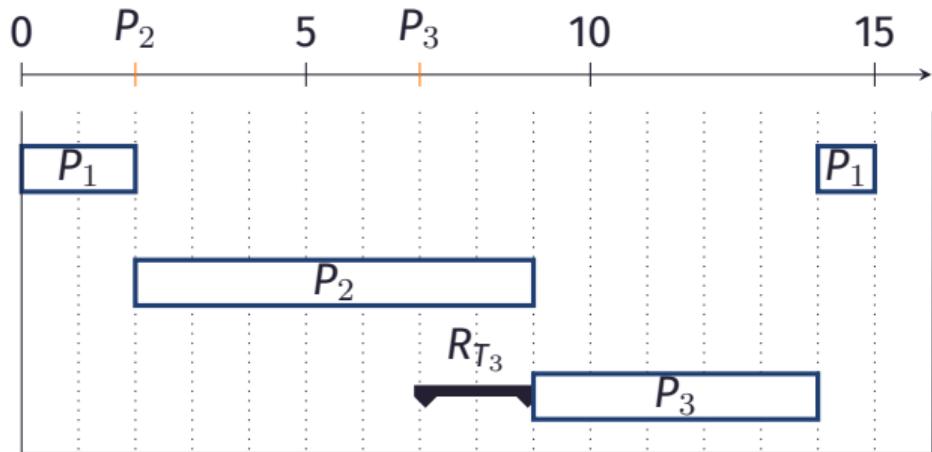


$$\begin{aligned} T_T &= \frac{TT_{T_1} + TT_{T_2} + TT_{T_3}}{3} \\ &= \frac{15 + 7 + 7}{3} \\ &= \frac{29}{3} \end{aligned}$$

Welche Formel kann für die Wartezeit T_{T_i} angegeben werden?

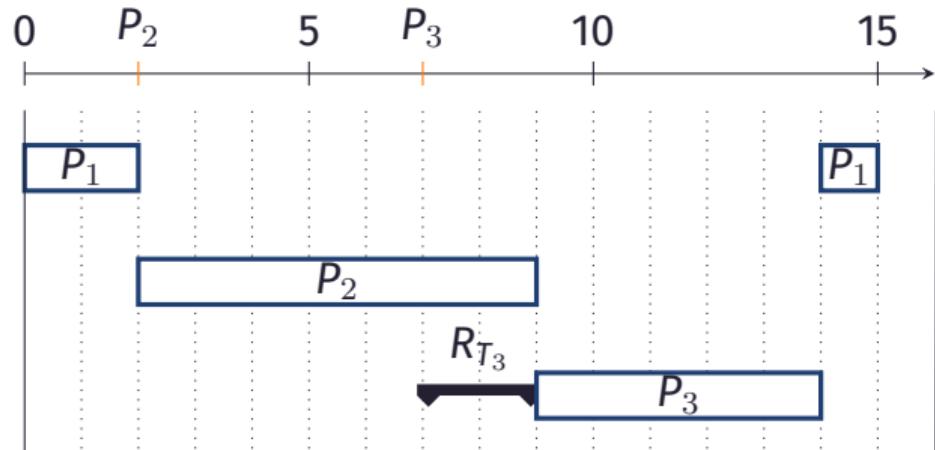
MITTLERE ANTWORTZEIT

Prozess	Ankunftszeit	Rechenzeit	Priorität
P_1	0	3	1
P_2	2	7	3
P_3	7	5	2



MITTLERE ANTWORTZEIT

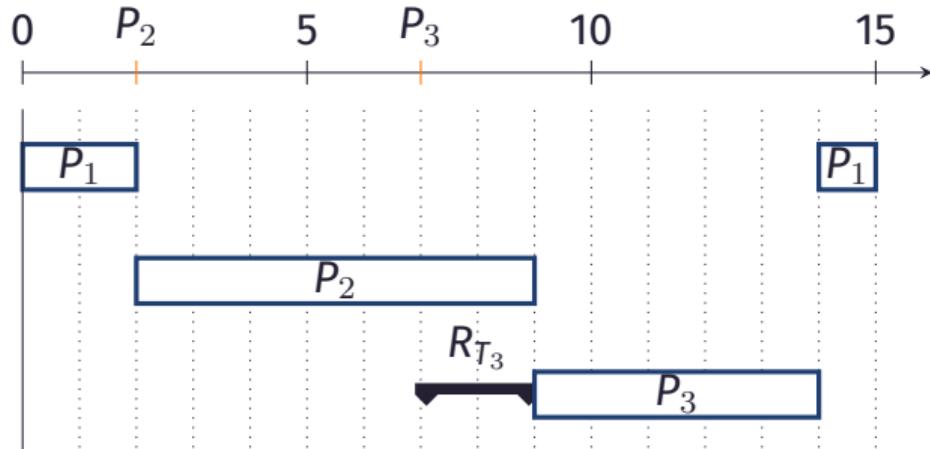
Prozess	Ankunftszeit	Rechenzeit	Priorität
P_1	0	3	1
P_2	2	7	3
P_3	7	5	2



$$\begin{aligned}R_T &= \frac{R_{T_1} + R_{T_2} + R_{T_3}}{3} \\&= \frac{0 + 0 + 2}{3} \\&= \frac{2}{3}\end{aligned}$$

MITTLERE ANTWORTZEIT

Prozess	Ankunftszeit	Rechenzeit	Priorität
P_1	0	3	1
P_2	2	7	3
P_3	7	5	2



$$\begin{aligned} R_T &= \frac{R_{T_1} + R_{T_2} + R_{T_3}}{3} \\ &= \frac{0 + 0 + 2}{3} \\ &= \frac{2}{3} \end{aligned}$$

Welche Formel kann für die Wartezeit R_{T_i} angegeben werden?

SCHED_SYM

* Dateien, die für Aufgabe a) angepasst werden müssen

```
/  
  main.c ..... Spezifikation der Prozesse  
  process.h ..... Deklaration des Prozess-Structs  
  queue.c ..... Implementation der Queue *  
  queue.h ..... Deklaration der Queue *  
  schedule.c ..... Scheduling-Algorithmen *  
  schedule.h ..... Deklaration der Scheduling-Algorithmen  
  statistics.c ..... Implementation der Statistik-Funktionen  
  internal ..... Verzeichnis für interne Funktionen  
    dispatch.c ..... Implementation des Dispatchers und der Loop  
    misc.c ..... Implementation des ncurses Interfaces  
    misc.h ..... Deklaration des global States  
    queue.c ..... Fallback Queue  
    schedule.c ..... Fallback Scheduling
```

BAUEN VON SCHED_SYM

Sched_Sym benutzt **cmake** als Build-System.

1. **Installieren** von CMake
2. `cmake -S . -B build`
3. `cmake --build build`
4. `./build/schedule_simulator -help`
5. `./build/schedule_simulator -scheduler rr`

Hinweis: Nutzen Sie **Linux** oder **WSL** (Windows Subsystem for Linux)!

OS X Nutzer: Beschwert euch bei Apple ☺ → nutzt **Coll und Tiree**

AUFGABE A)

Signaturen:

```
int schedule_fifo(process_t *current_process, process_t **next_process);
int schedule_rr(process_t *current_process, process_t **next_process);
int schedule_sjf(process_t *current_process, process_t **next_process);
int schedule_prio(process_t *current_process, process_t **next_process);
```

AUFGABE A)

Signaturen:

```
int schedule_fifo(process_t *current_process, process_t **next_process);  
int schedule_rr(process_t *current_process, process_t **next_process);  
int schedule_sjf(process_t *current_process, process_t **next_process);  
int schedule_prio(process_t *current_process, process_t **next_process);
```

Description:

1. **schedule_*** wählt den nächsten Prozess aus der Run-Queue aus.
2. Schedule wird bei **jedem Timer-Interrupt** aufgerufen, entsprechend kann *current_process* = *next_process* gelten.
3. Die scheduling Funktionen fügt den *current_process* je nach Scheduling-Strategie in die **Run-Queue ein**.

AUFGABE A)

Return-Value:

- 0: Es wurde **erfolgreich** der neue Prozess Pointer in next_process gesetzt
- -1: Es wurde **kein** neuer Prozess gefunden
- andere: **selbst definierbare** Fehlercodes

AUFGABE A)

Return-Value:

- 0: Es wurde **erfolgreich** der neue Prozess Pointer in next_process gesetzt
- -1: Es wurde **kein** neuer Prozess gefunden
- andere: **selbst definierbare** Fehlercodes

Hilfreiche Variablen & Methoden:

- **current_core->run_queue**
- Methoden für die Queue, die in queue.h deklariert und in queue.c implementiert sind

QUEUE FUNCTIONS

In queue.c in der Funktion queue_set_functions können die Queue-Funktionen gesetzt werden:

```
//setting queue functions
global_state.queue_functions.queue_init = queue_init;
global_state.queue_functions.queue_enqueue =
    → queue_enqueue;
global_state.queue_functions.queue_destroy =
    → queue_destroy;

// If you want that new processes are added in a different
    → way for one scheduler
// if (schedule == schedule_sjf) {
//     global_state.queue_functions.queue_enqueue =
        → queue_enqueue_sjf;
// }
```

FIFO

```
int schedule_fifo(process_t *current_process, process_t
→ **next_process) {
    if(current_process && current_process->state == READY) {
        *next_process = current_process;
        return 0;
    }

    if (queue_length(current_core->run_queue) < 1) {
        return -1;
    }

    return queue_dequeue(current_core->run_queue,
→     next_process);
}
```

AUFGABE 3: WINDOWS-SCHEDULER

- nutze modifiziertes Gantt-Diagramm: y-Achse = Priorität

AUFGABE 3: WINDOWS-SCHEDULER

- nutze modifiziertes Gantt-Diagramm: y-Achse = Priorität
- 3 Szenarien für Prozess-Scheduling:
 1. Prozess war **blockiert** (Netzwerk-I/O) und erhält nun die CPU
 2. Prozess ist **GUI-Thread**
 3. Prozess hat schon seit **mehr als 4 Sekunden** nicht mehr gerechnet

AUFGABE 3: WINDOWS-SCHEDULER

- nutze modifiziertes Gantt-Diagramm: y-Achse = Priorität
- 3 Szenarien für Prozess-Scheduling:
 1. Prozess war **blockiert** (Netzwerk-I/O) und erhält nun die CPU
 2. Prozess ist **GUI-Thread**
 3. Prozess hat schon seit **mehr als 4 Sekunden** nicht mehr gerechnet

Zusatzinformation aus Windows Internals Seventh Edition Part 1:

*On client version of Windows, threads run for **two clock intervals by default**. [...] Internally, a quantum unit is represented as **one-third of a clock tick**. That is, one clock tick equals three quantums. This means that on client Windows systems, threads have a **quantum reset value of 6** ($2 * 3$) [...]*

AUFGABE 3: WINDOWS-SCHEDULER

- nutze modifiziertes Gantt-Diagramm: y-Achse = Priorität
- 3 Szenarien für Prozess-Scheduling:
 1. Prozess war **blockiert** (Netzwerk-I/O) und erhält nun die CPU
 2. Prozess ist **GUI-Thread**
 3. Prozess hat schon seit **mehr als 4 Sekunden** nicht mehr gerechnet

Zusatzinformation aus Windows Internals Seventh Edition Part 1:

*On client version of Windows, threads run for **two clock intervals by default**. [...] Internally, a quantum unit is represented as **one-third of a clock tick**. That is, one clock tick equals three quantums. This means that on client Windows systems, threads have a **quantum reset value of 6** ($2 * 3$) [...]*

Hinweis: Gehen Sie von einem Clock-Interval von 10ms aus!

AUFGABE 3: WINDOWS-SCHEDULER

- Einem Thread sind eine **baseprio** und eine **current_prio** zugeordnet:

$$\text{current_prio_neu} := \min\{\text{baseprio} + \text{boost}; 15\}$$

AUFGABE 3: WINDOWS-SCHEDULER

- Einem Thread sind eine **basepri** und eine **current_prio** zugeordnet:

$$\text{current_prio_neu} := \min\{\text{basepri} + \text{boost}; 15\}$$

Standard: **basepri** = 8

AUFGABE 3: WINDOWS-SCHEDULER

- Einem Thread sind eine **baseprio** und eine **current_prio** zugeordnet:

$$\text{current_prio_neu} := \min\{\text{baseprio} + \text{boost}; 15\}$$

Standard: **baseprio = 8**

- Priorität **und** Quantum eines Threads werden in gewissen Situationen kurzfristig verändert, z.B.:

AUFGABE 3: WINDOWS-SCHEDULER

- Einem Thread sind eine **baseprio** und eine **current_prio** zugeordnet:

$$\text{current_prio_neu} := \min\{\text{baseprio} + \text{boost}; 15\}$$

Standard: **baseprio** = 8

- Priorität **und** Quantum eines Threads werden in gewissen Situationen kurzfristig verändert, z.B.:
 1. nach I/O (Network): $\text{boost} = 2$ Nach jeder Zeitscheibe wird die Priorität (**current_prio**) um 1 gesenkt, bis die **baseprio** wieder erreicht ist.

AUFGABE 3: WINDOWS-SCHEDULER

- Einem Thread sind eine **baseprio** und eine **current_prio** zugeordnet:

$$\text{current_prio_neu} := \min\{\text{baseprio} + \text{boost}; 15\}$$

Standard: baseprio = 8

- Priorität **und** Quantum eines Threads werden in gewissen Situationen kurzfristig verändert, z.B.:
 1. **nach I/O (Network):** boost = 2 Nach jeder Zeitscheibe wird die Priorität (**current_prio**) um 1 gesenkt, bis die **baseprio** wieder erreicht ist.
 2. **Threads einer interaktiven Sitzung (z.B. GUI-Threads):** boost = 6, gleichzeitig wird die Anzahl der Quantum-Units verdreifacht

AUFGABE 3: WINDOWS-SCHEDULER

- Einem Thread sind eine **baseprio** und eine **current_prio** zugeordnet:

$$\text{current_prio_neu} := \min\{\text{baseprio} + \text{boost}; 15\}$$

Standard: baseprio = 8

- Priorität und Quantum eines Threads werden in gewissen Situationen kurzfristig verändert, z.B.:

1. **nach I/O (Network):** boost = 2 Nach jeder Zeitscheibe wird die Priorität (**current_prio**) um 1 gesenkt, bis die **baseprio** wieder erreicht ist.
2. **Threads einer interaktiven Sitzung (z.B. GUI-Threads):** boost = 6, gleichzeitig wird die Anzahl der Quantum-Units verdreifacht
3. **Threads, die länger als 4 s im Zustand rechenbereit sind:**
 $\text{current_prio_neu} := 15$, gleichzeitig wird Anzahl der Quantum-Units auf 3 reduziert (→ **Balance Set Manager**)

AUFGABE 3: WINDOWS-SCHEDULER

- Einem Thread sind eine **basepri** und eine **current_prio** zugeordnet:

$$\text{current_prio_neu} := \min\{\text{basepri} + \text{boost}; 15\}$$

Standard: basepri = 8

- Priorität **und** Quantum eines Threads werden in gewissen Situationen kurzfristig verändert, z.B.:

1. **nach I/O (Network):** boost = 2 Nach jeder Zeitscheibe wird die Priorität (**current_prio**) um 1 gesenkt, bis die **basepri** wieder erreicht ist.
2. **Threads einer interaktiven Sitzung (z.B. GUI-Threads):** boost = 6, gleichzeitig wird die Anzahl der Quantum-Units verdreifacht
3. **Threads, die länger als 4 s im Zustand rechenbereit sind:**
 $\text{current_prio_neu} := 15$, gleichzeitig wird Anzahl der Quantum-Units auf 3 reduziert (→ **Balance Set Manager**)

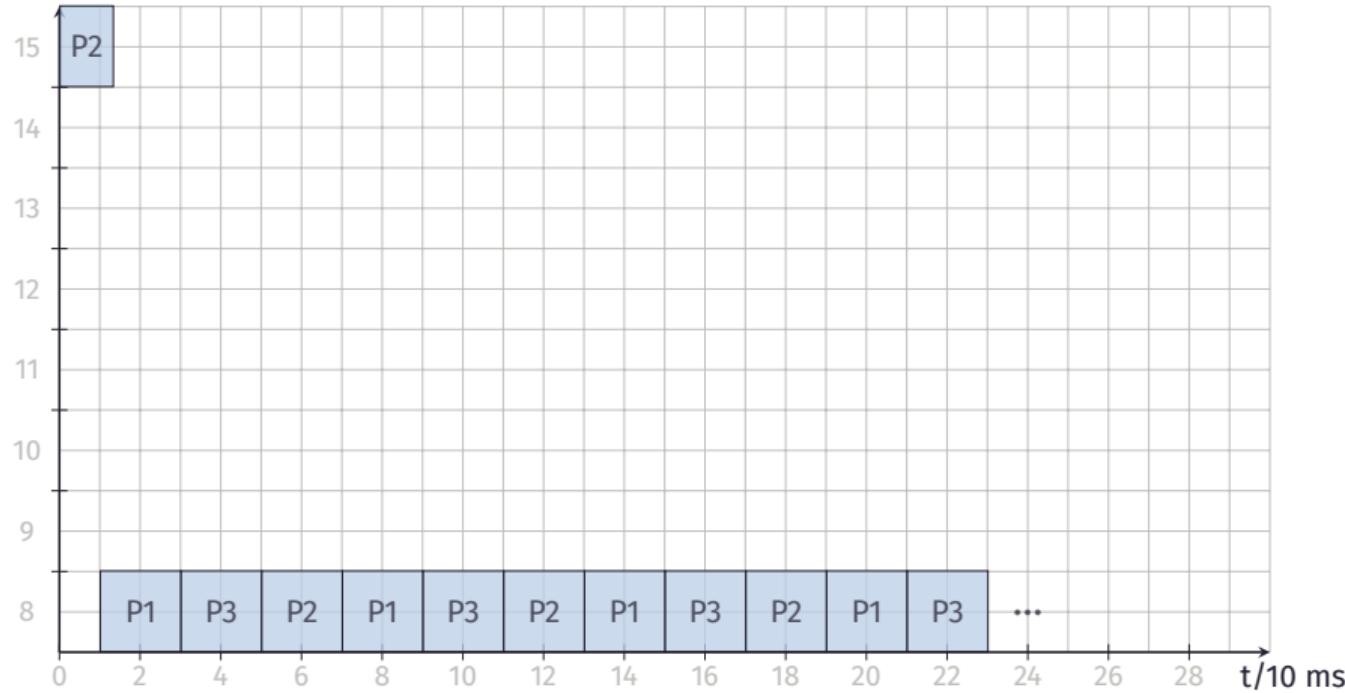
! Achtung: In Fall 2 und 3 verfällt der Boost sofort nach Ablauf eines Quantums!

WINDOWS SCHEDULER C)

Im dritten Szenario hat Prozess 2 schon seit mehr als 4 Sekunden nicht gerechnet.

WINDOWS SCHEDULER C)

Im dritten Szenario hat Prozess 2 schon seit mehr als 4 Sekunden nicht gerechnet.



LITERATUREMPFEHLUNG

- *The Linux Programming Interface*, Michael Kerrisk,
Kapitels 6, 24, 26
- *Windows Internals Seventh Edition Part 1*, Pavel Yosifovich, Alex Ionescu, Mark E. Russinovich, David A. Solomon
Kapitel Threads - Thread Scheduling
- man pages:
 - man 2 fork
 - man 3 exit
 - man 2 wait

Noch Fragen?

Max Schrötter

Potsdam, 08.11.2024

schroetter@cs.uni-potsdam.de

PRÄSENTATIONEN

