Universität Bamberg

GRABS: Grundlagen der Rechnerarchitektur und Betriebssysteme

Literatur:
Arpaci-Dusseau [1]
Kap. 4 – 7

Vorlesung 11:
Prozesse und das Betriebssystem

Michael Engel (michael.engel@uni-bamberg.de)
Lehrstuhl für Praktische Informatik, insbes. Systemnahe Programmierung
https://www.uni-bamberg.de/sysnap

Prozess–Betriebssysteminteraktion

- Wie fordert ein Anwendungsprogramm einen Dienst des Betriebssystems an?
- Aus Sicht der Anwendung sieht ein **Systemaufruf** aus wie ein gewöhnlicher Funktionsaufruf, z.B.:

```
pid = fork();
```

- Das wahlfreie Aufrufen von Code innerhalb des Betriebssystemkerns ist aber gefährlich:
  - Keine Überprüfung der Rechte, eine Funktion auszuführen
  - Keine Überprüfung auf Korrektheit der Parameter
    ➛ ***Sicherheitslücken!***
- Der Übergang von Code, der in einer Anwendung ausgeführt wird zu Code, der im Kern läuft, muss geschützt werden!

Prozess–Betriebssysteminteraktion

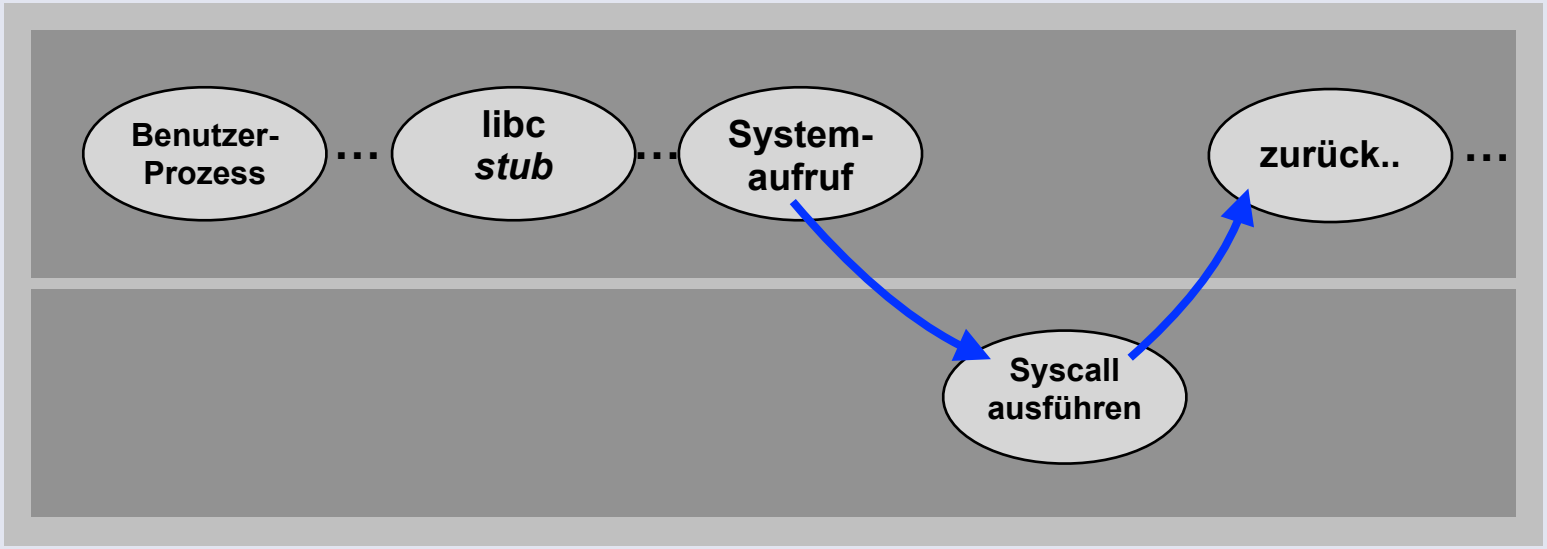
Universität Bamberg

- Der Übergang von Code, der in einer Anwendung ausgeführt wird zu Code, der im Kern läuft, muss geschützt werden!
- Wir haben schon die *Privilegienmodi* kennengelernt*:*
  - “Benutzermodus”: nur eingeschränkte Funktionalität
  - “Kern-” oder “Supervisormodus”: voller Zugriff auf alle Hardware-Ressourcen
- Besondere Instruktionen ermöglichen den Übergang vom Benutzermodus zum Kermodus:
  - int 0x80 (intel x86), syscall/sysenter (intel/AMD64)
  - trap (Motorola 68k), SVC (ARM), **ecall** (RISC-V)
- Ausführung einer solchen Instruktion resultiert im Wechsel des Ausführungsmodus der CPU zum privilegierten Modus und Sprung an eine von der CPU-Hardware vorgebene Adresse: ***Systemaufruf***

Viele Prozessoren unterstützten weitere Abstufungen der Privilegien. So besitzt die x86-Architektur vier sog. "Schutzringe", RISC-V besitzt neben Supervisor und User Mode noch den Machine Mode

Prozess–Betriebssysteminteraktion

Universität Bamberg



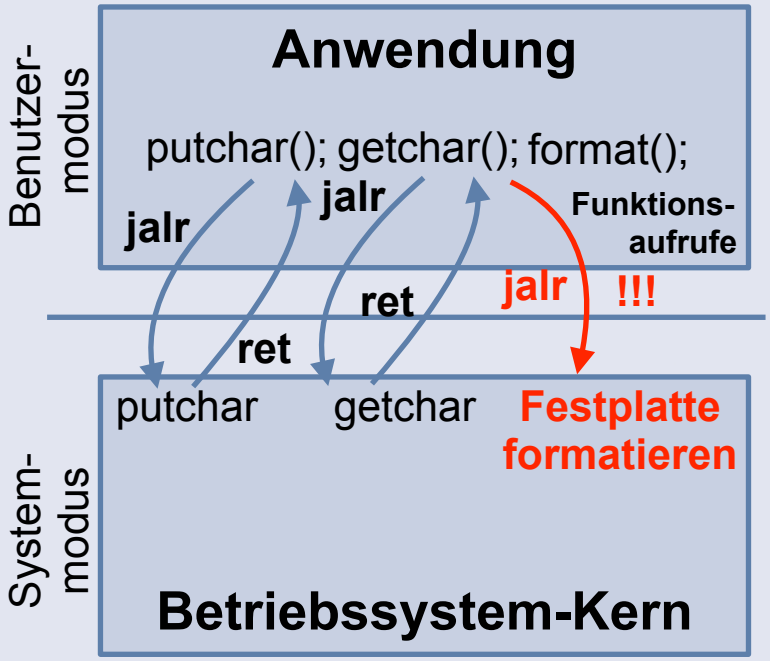
- Anwendungen können einen Systemaufruf direkt ausführen, aber:
  - Funktioniert nicht mehr, wenn Syscall-Schnittstelle geändert
- In modernen Systemen stellt die C-Library (libc) ***stubs*** (Adapter-funktionen bereit), die den eigentlichen Syscall aufrufen
  - Die Stubfunktion ist eine gewöhnliche gelinkte Funktion

OpenBSD verbietet den direkten Aufruf von Syscalls seit 2023 komplett und erfordert den Aufruf via libc!
siehe Theo de Raadt (openbsd-tech vom 27.10.2023): "Removing syscall(2) from libc and kernel"

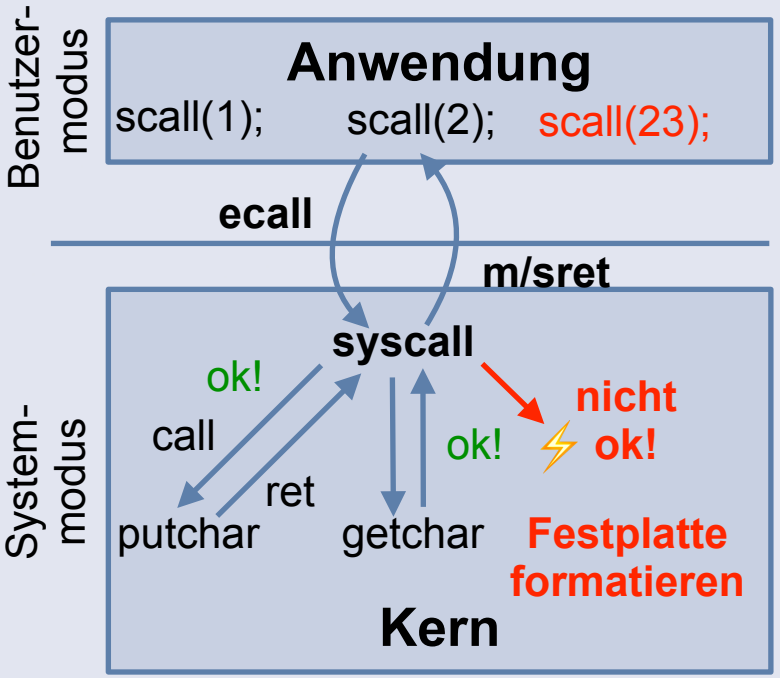
Systemaufrufe vs. Funktionsaufrufe

- **Sicherstellen eines *kontrollierten Übergangs* vom Benutzermodus zum Systemmodus (Kernel)**
  - Benutzerprogramme (Anwendungen) dürfen nur zugelassene Funktionen aufrufen
  - Betriebssystem-Kern überprüft aufgerufene Funktionen und Parameter

Direkte Funktionsaufrufe an BS



Verwendung von Systemaufrufen



Systemaufrufe bei RISC-V

- Wie können wir Systemaufrufe realisieren?
  - Die C-Library (libc) stellt ***Stub-Funktionen*** zur Verfügung
    - Diese sind die Schnittstelle zu den Systemaufrufen
    - Rufen die `ecall`-Instruktion mit passenden Parametern auf
  - Werden wie normale Funktionen aufgerufen
    - Also: Parameter in `a0`, `a1`, …, Rückgabewert in `a0`
- **Besonderheiten:**
  - **Nummer des Systemaufrufs** *(system call* oder *syscall)* in Register `a7`
  - Also nur sieben Parameter in Registern übergebbar (Rest: Stack)

```
#define SYS_sleep 13
    .global sleep
sleep:
    li a7, SYS_sleep
    ecall
    ret
```

Typische Systemaufrufe

- Beispiel aus dem Lehrbetriebs-system xv6 vom MIT [5]
- Diese Aufrufe entsprechen ungefähr den Systemaufrufen von 6th Edition Unix von 1976
- Die meisten auch in aktuellem Unix/Linux noch vorhanden
- Blau: Syscalls für Prozessverwaltung

Nr	Name	Funktion
1	fork	Neuen Prozess erzeugen
2	exit	Aktuellen Prozess beenden
3	wait	Auf Prozess warten
4	pipe	Interprozesskommunikation
5	read	Daten von Dateideskriptor lesen
6	kill	Prozess terminieren (Signal senden)
7	exec	Neuen Prozess laden
8	fstat	Status einer Datei prüfen
9	chdir	Aktuelles Verzeichnis ändern
10	dup	Dateideskriptor duplizieren
11	getpid	Prozess-ID des aktuellen Prozesse holen
12	sbrk	Mehr Speicher anfordern
13	sleep	Prozess eine Zeit lang schlafen legen
14	upZeit	Laufzeit des Systems abfragen
15	open	Datei öffnen, gibt Deskriptor zurück
16	write	Daten in Dateideskriptor schreiben
17	mknod	Neue Spezialdatei anlegen
18	unlink	Datei löschen
19	link	Verweis auf Datei anlegen
20	mkdir	Verzeichnis anlegen
21	close	Datei schließen

Vom Programm zum Prozess

- Das Betriebssystem startet unser Programm an Adresse `_start`
  - …die Start-Funktion ruft danach `main` auf
  - Ein **Programm in Ausführung** bezeichnen wir als ***Prozess***
- Das Programm läuft jetzt und führt aus:
  - Variablenzuweisungen, Kontrollstrukturen, Funktionsaufrufe…
  - Programm **ändert** seinen **Zustand**
    - Wir kennen schon: **Prozessorregister**
  - Außerdem:
    - Hauptspeicher (RAM) – `.data`, `.bss`, Stack und Heap

- **Mehr als ein Prozess?**
  - Wie kann ein Betriebssystem mehrere Prozesse verwalten?

Was ist ein Prozess?

- Horning/Randell, *Process Structuring*

> A *process* is a triple (S, f, s), where S is a state space, f is an action function in that space, and s is the subset of S which defines the initial states of the process. A process generates all the computations generated by its action function from its initial states.

- Dennis/van Horn, *Programming Semantics for Multiprogrammed Computations*

> A *process* is a locus of control within an instruction sequence. That is, a process is that abstract entity which moves through the instructions of a procedure as the procedure is executed by a processor.

- Habermann, *Introduction to Operating System Design*

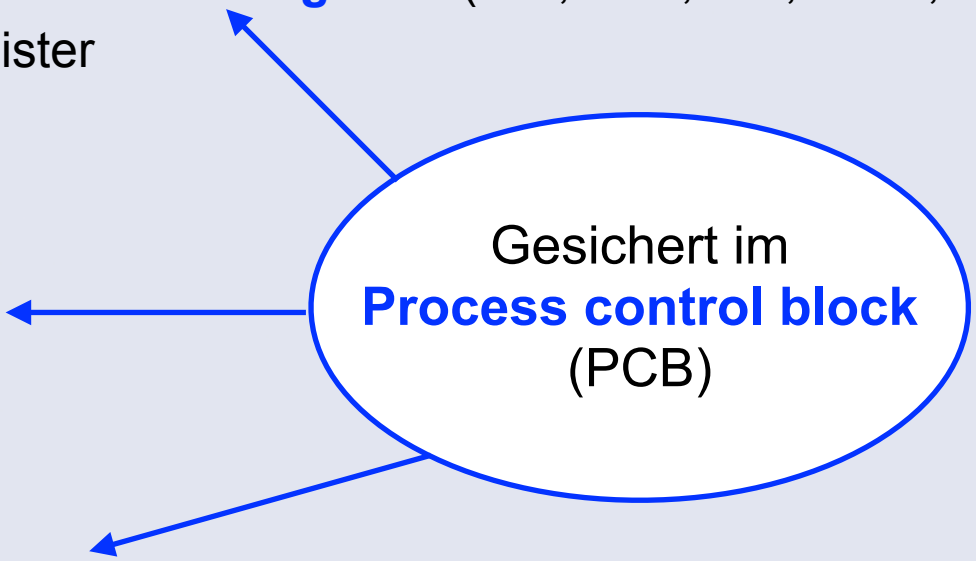
> A *process* is controlled by a program and requires a processor to execute that program.

Ein Prozess…

- „***…ist ein Programm in Ausführung*** “ [unbekannte Quelle]

Dies erfordert einen **Prozesskontext**, bestehend aus...

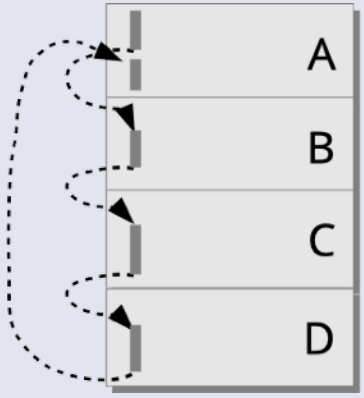
- Speicher: **Code-**, **Daten-** and **Stacksegment** (text, data, bss, stack, heap)
- Inhalte der Prozessorregister
  - Instruction pointer
  - Stack pointer
  - Registers
  - …
- Prozesszustand
- Benutzer-ID
- Zugriffsrechte
- Aktuell verwendete Ressourcen
  - Files, I/O devices, etc.
- ...



Das Prozessmodell

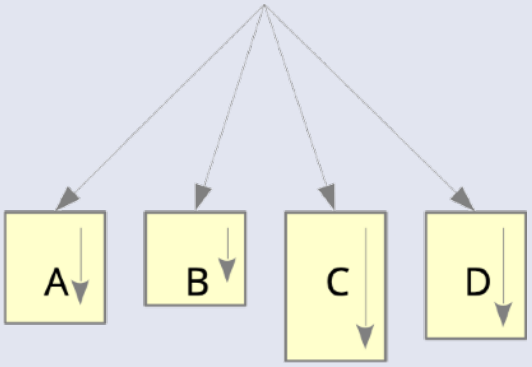
Universität Bamberg

Multiprogrammierung



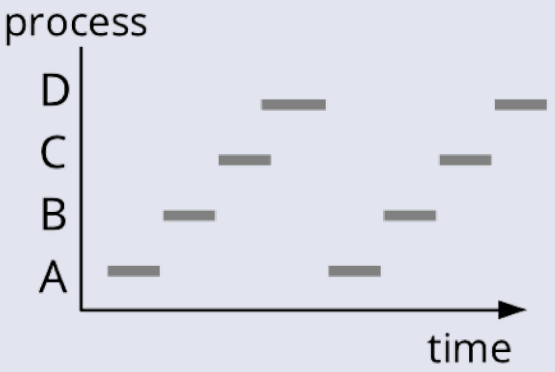
Technische Sicht:
- 1 Instruction pointer
- Kontextumschaltung

Nebenläufige Prozesse



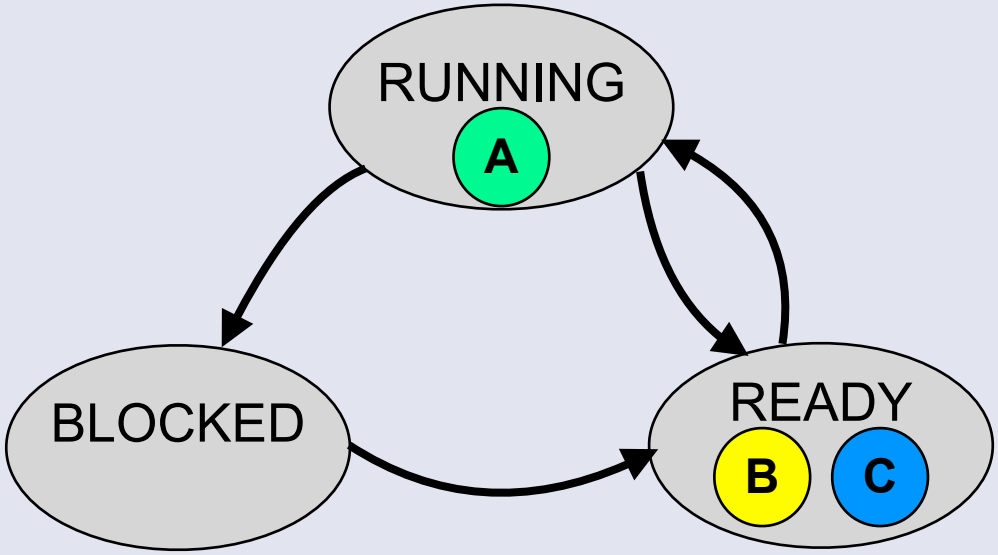
Konzeptionelle Sicht:
- 4 unabhängige sequentielle Kontrollflüsse

CPU-Multiplexing



Echtzeit-Sicht:
(Gantt-Diagramm)
- Nur ein Prozess ist zu jedem gegebenen Zeitpunkt aktiv (bei einem Einprozessor-system)

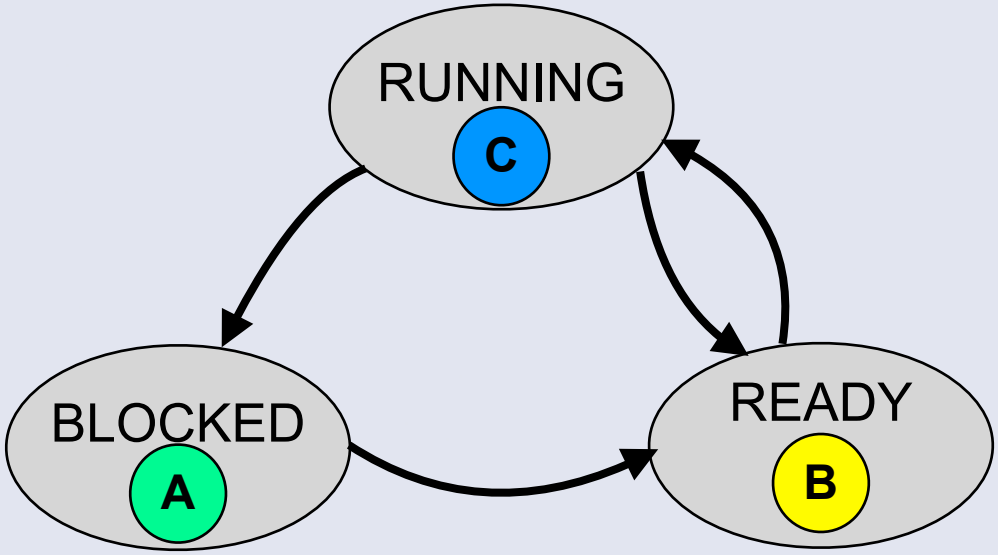
Prozessverhalten und Zustände (1)



Prozesszustände:

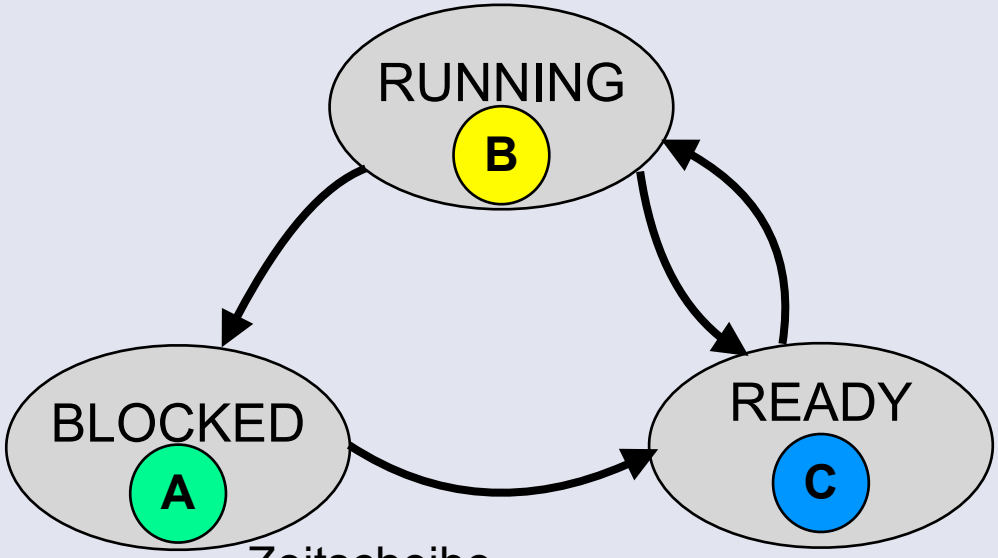
- **RUNNING** (laufend)
  - Prozess wird gerade ausgeführt
- **READY** (bereit)
  - Prozess ist zur Ausführung bereit und wartet nur auf die CPU
- **BLOCKED** (blockiert)
  - Prozess wartet auf das Ende einer I/O-Operation

Prozessverhalten und Zustände (2)

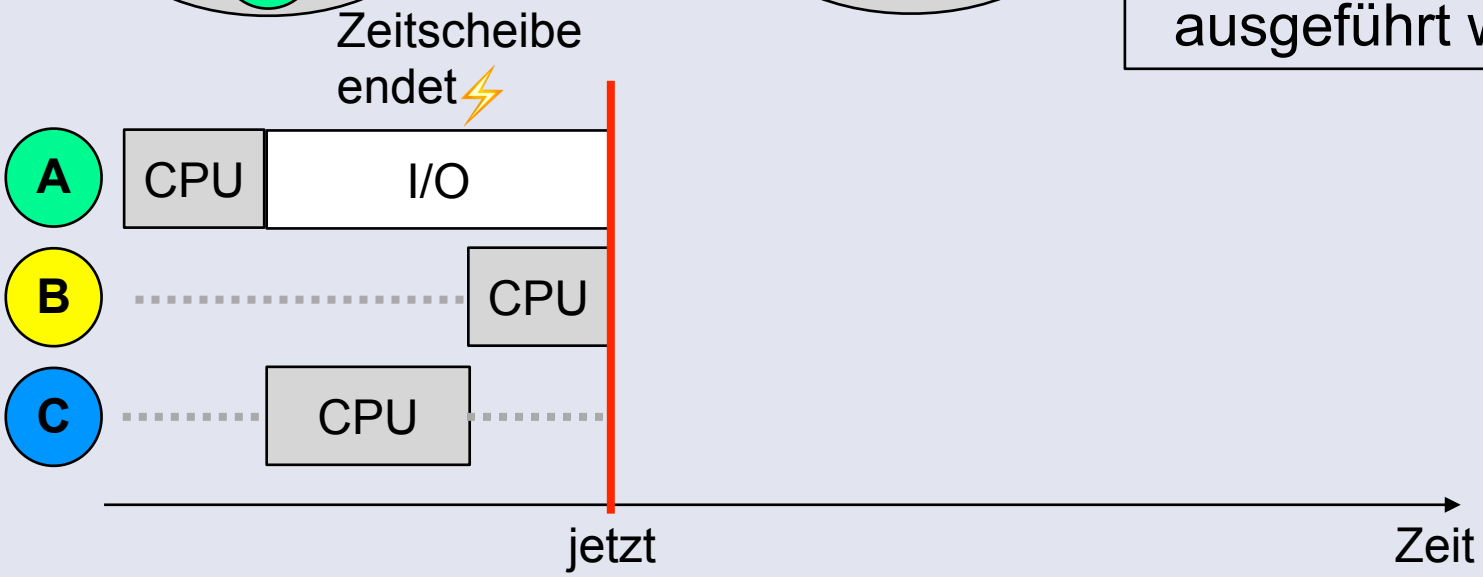


- Prozess A hat eine I/O-Operation gestartet und wurde in Zustand BLOCKED versetzt
- Da A jetzt die CPU nicht mehr nutzen kann, wählt das OS Prozess C und setzt ihn von READY nach RUNNING.
- Dies ist eine **Kontext-umschaltung** (*context switch*) von A zu C

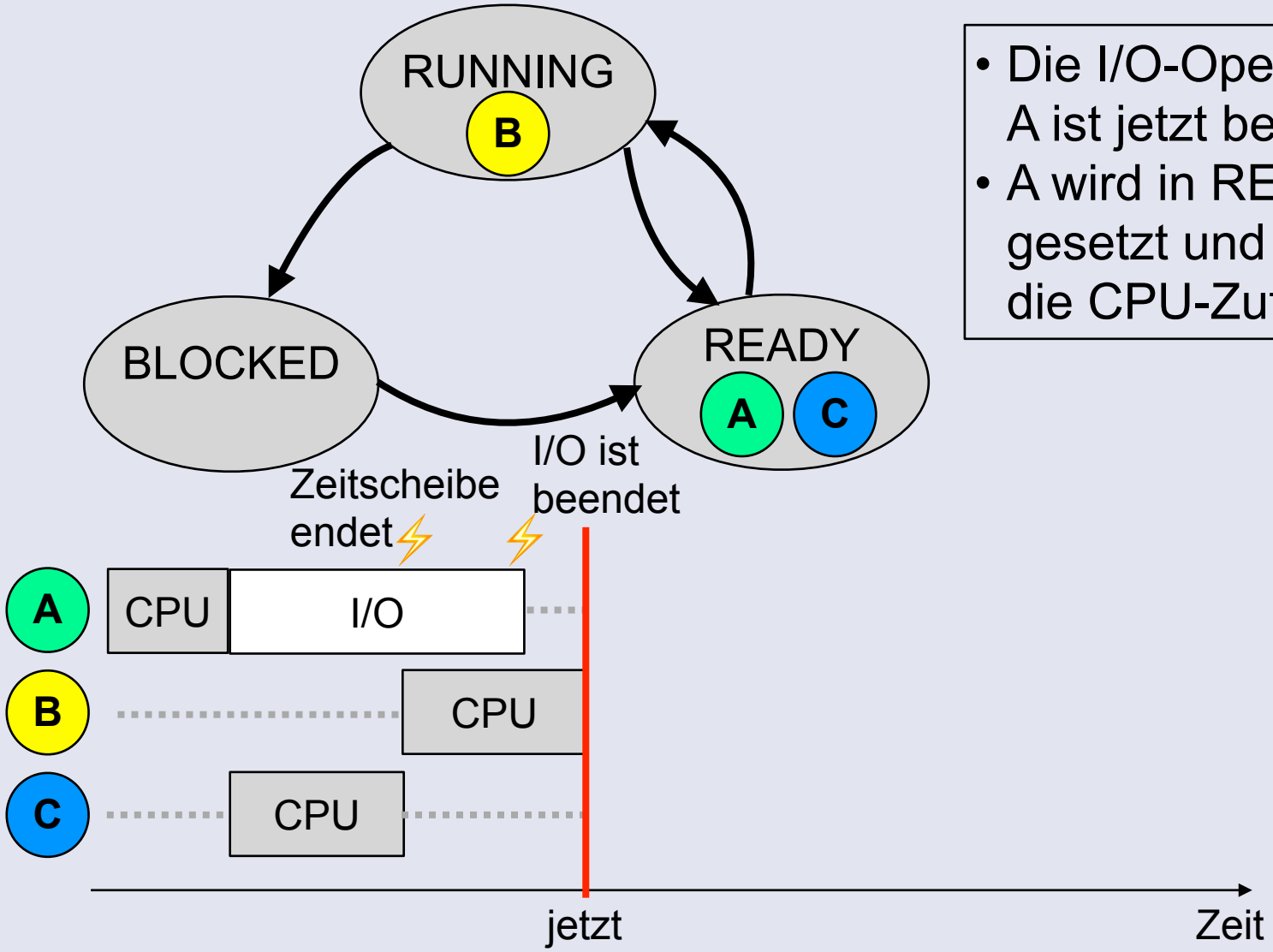
Prozessverhalten und Zustände (3)



- Prozess C hat die ihm zugeteilte CPU-Zeit verbraucht und wird **verdrängt** (*preempted*)
- Schließlich wird Prozess B in Zustand RUNNING gesetzt und kann ausgeführt werden

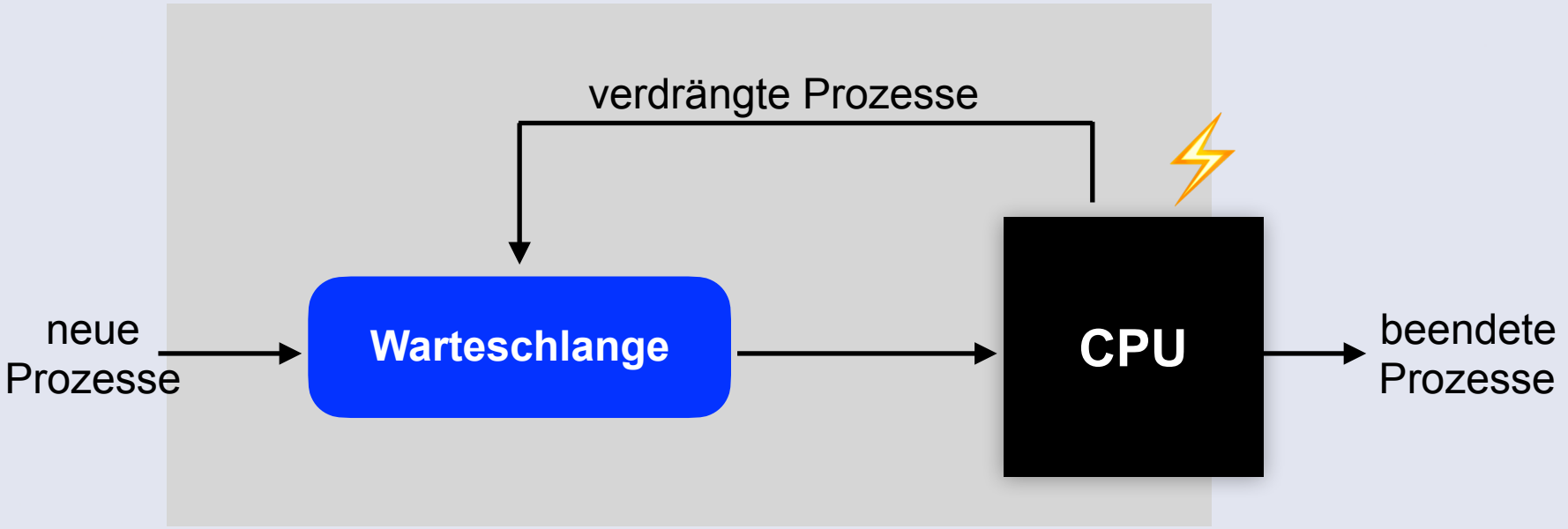


Prozessverhalten und Zustände (4)



- Die I/O-Operation von A ist jetzt beendet
- A wird in READY gesetzt und wartet auf die CPU-Zuteilung

CPU-Zuteilung (*scheduling*)



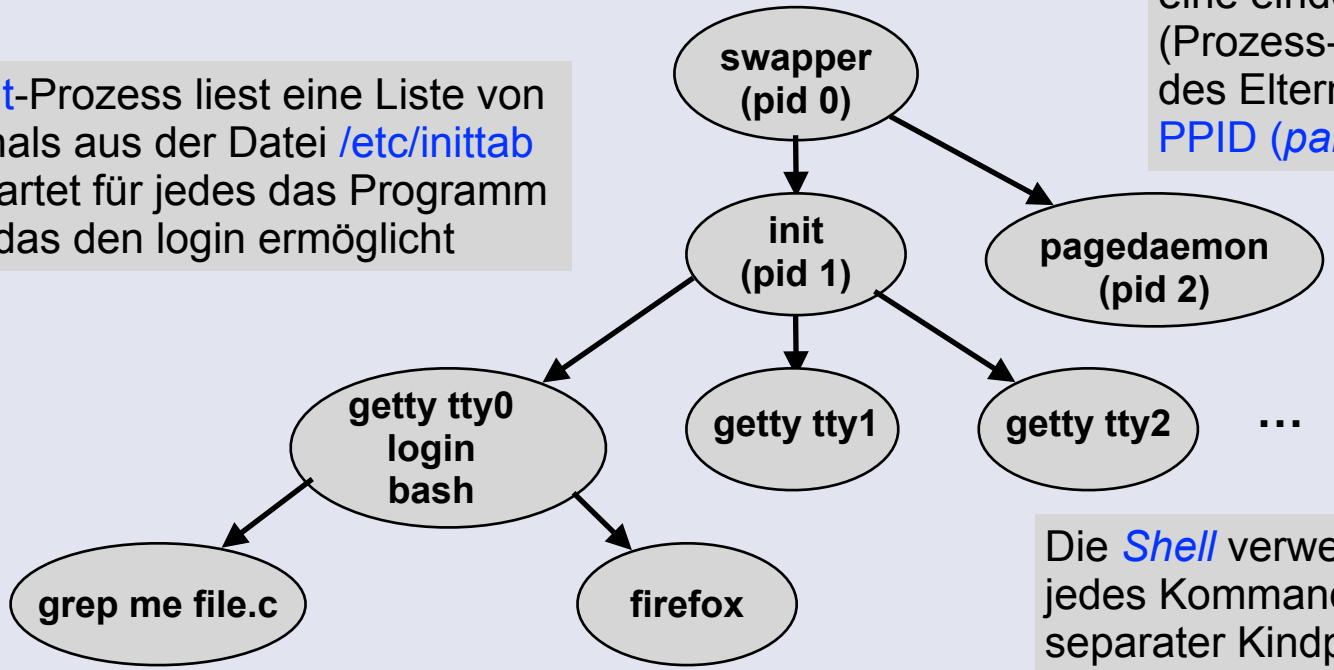
Ein **Zuteilungsalgorithmus** (*scheduling algorithm*) ist gekennzeichnet durch die Reihenfolge der Prozesse in der Warteschlange und die Bedingungen, unter denen Prozesse zur Warteschlange hinzugefügt werden

Unix Prozesse

- …sind die Methode zur **Strukturierung von Aktivitäten**
  - sowohl für Anwendungen wie für Systemprozesse
- …können **neue Prozesse** schnell und einfach **erzeugen**
  - Elternprozess ➛ Kindprozess
- …formen eine **Prozesshierarchie**:

Jeder Unixprozess besitzt eine eindeutige Nummer (Prozess-ID, PID). Die PID des Elternprozesses wird PPID (*parent PID*) genannt

Der init-Prozess liest eine Liste von Terminals aus der Datei /etc/inittab und startet für jedes das Programm getty, das den login ermöglicht



Die *Shell* verwendet Prozesse: jedes Kommando wird als separater Kindprozess ausgeführt

Unix-Systemaufrufe zur Prozesssteuerung

- Die folgenden Systemaufrufe dienen der Prozessverwaltung [4]

`getpid` (2)	Liefert PID des aufrufenden Prozesses
`getppid` (2)	Liefert PID des Elternprozesses (PPID)
`getuid` (2)	Liefert UID des aufrufenden Prozesses
`fork` (2)	Erzeugt einen neuen Kindprozess
`exit` (3), `_exit` (2)	Beendet den aufrufenden Prozess
`wait` (2)	Wartet auf das Ende eines Kindprozesses
`execve` (2)	Lädt und startet ein Programm im Kontext des aufrufenden Prozesses

Die Zahl in Klammern gibt den **Abschnitt** der Unix-Handbuchseiten (*manual pages*) an, in der der Syscall beschrieben ist.
Anzeige z.B. mit `man 2 wait`

Unix-Prozesse in detail: `fork()`

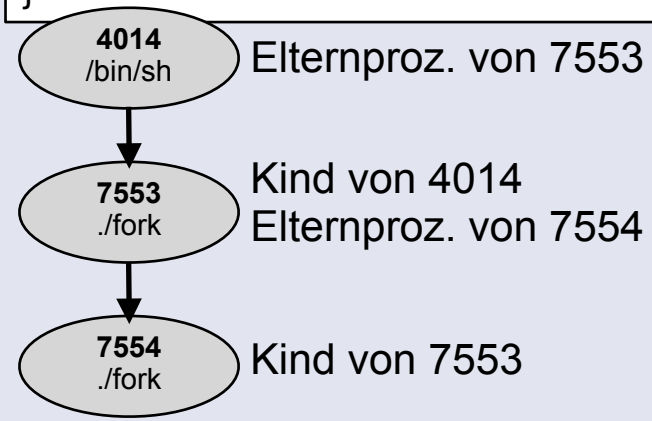
Systemaufruf: `pid_t fork (void)`

Dies ist der C-Prototyp für fork:
keine Parameter (void),
liefert einen pid_t-Wert

- **Dupliziert** den aufrufenden Prozess
  (so erzeugt man neue Prozesse in Unix!)
- Der Kindprozess "erbt"…
  - Addressraum (code, data, bss, stack-Segmente)
  - Benutzer und Gruppen-ID
  - Standard I/O-Kanäle
  - Prozessgruppe, Signaltabelle (mehr dazu später)
  - Offene Dateien, aktuelles Arbeitsverzeichnis
- ***Nicht*** übernommen werden:
  - Prozess-ID (PID), Elternprozess-ID (PPID)
  - Ausstehende Signale, Abrechnungsdaten, ...
- **Ein Prozess ruft fork auf, aber zwei Prozesse kehren zurück!**

Verwendung von `fork()`


```
... /* includes */
int main () {
    int pid;
    printf("In parent: pid %d PPID %d\n", getpid(), getppid());
    pid = fork();  /* Prozess wird hier dupliziert!
                      Beide laufen von hier aus weiter */
    if (pid > 0)
        printf("In the parent process, child PID %d\n", pid);
    else if (pid == 0)
        printf("In the child process, PID %d PPID %d\n",
                getpid(), getppid());
    else
        printf("Oh, an error!\n"); /* mehr in den Übungen…*/
}
```




```
me@unix:~> gcc -o fork fork.c
me@unix:~> ./fork
In parent: pid 7553 PPID 4014
In the child process,  PID 7554
PPID 7553
In the parent process, child PID
7554
```

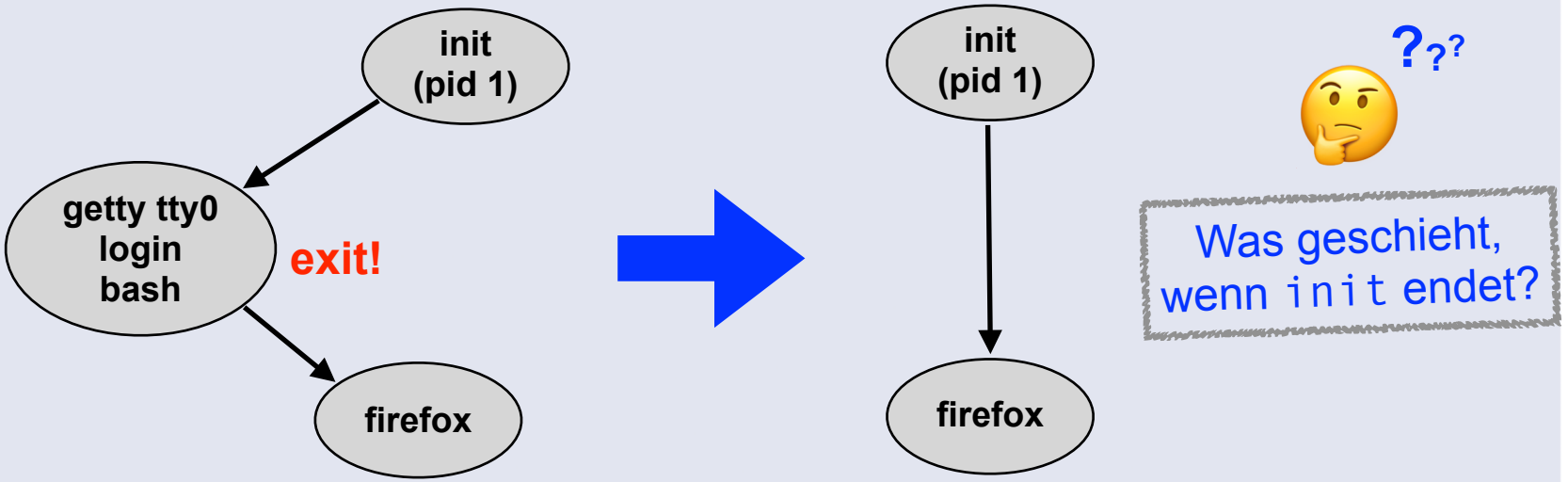
Unix Prozesse in detail: `_exit()`

Systemaufruf: `void _exit (int)`

- Beendet den aufrufenden Prozess und liefert das Argument als "*exit status*" an den Elternprozess
  - Dieser Aufruf ***kehrt nicht zurück***!
- Gibt die vom Prozess belegten Ressources frei
  - offene Dateien, belegten Speicher, …
- Benachrichtigt den Elternprozess vom "Ableben" des Kindes
- Es gibt auch eine Libraryfunktion `exit(3)`, die zusätzlich von der C-Library belegte Ressourcen freigibt
  - U.a. leert (*flush*) dies alle Daten, die sich noch in Ausgabepuffern befinden!
- Normale Prozesse sollten `exit (3)` und nicht `_exit` verwenden

Diskussion: verwaiste Prozesse

- Ein Unix-Prozess ist **verwaist**, wenn sein Elternprozess "stirbt" (beendet wird)
- Wie sieht dann unsere Prozesshierarchie aus??



Der **init**-Prozess (immer pid 1) "adoptiert" alle verwaisten Prozesse. Damit bleibt die Prozesshierarchie weiter intakt.

Unix-Prozesse im Detail: `wait()`

Systemaufruf: `pid_t wait (int *)`

- Blockiert den aufrufenden Prozess, bis **einer** seiner Kindprozesse endet
  - Der Rückgabewert von `wait` ist die PID des beendeten Kindprozesses
  - Über den `int *`-Parameter kann der Aufrufer dann den "exit status" des Kindprozesses (und mehr) übergeben bekommen
- `wait` kehrt sofort zurück, wenn alle Kindprozesse bereits beendet sind

Verwendung von `wait()`


```
... /* includes, main() { … */
pid = fork();   /* Erzeugt Kindprozess */
if (pid > 0) {
   int status;
   sleep(5);    /* Libraryfunktion: 5 Sekunden schlafen */
   if (wait(&status) == pid && WIFEXITED(status))
      printf ("Exit status: %d\n", WEXITSTATUS(status));
}
else if (pid == 0) {
   exit(42);
}
...
```

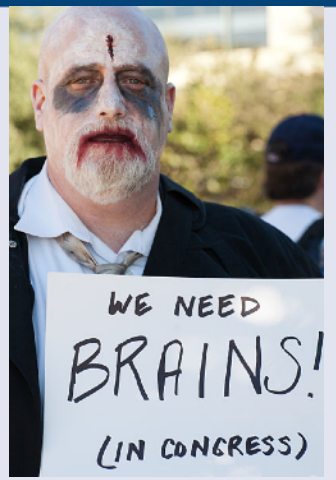
Ein Prozess kann auch von außen ***"getötet"*** werden, d.h. er ruft nicht `exit` auf.
In diesem Fall würde `WIFEXITED` den Wert 0 zurückgeben.

```
me@unix:~> ./wait
Exit status: 42
```

Diskussion: ***Zombies***

Universität Bamberg

- Ein beendeter Prozess wird zum ***“Zombie”***, bis sein exit mit `wait` abgefragt wurde
- Die solchen Prozessen zugeteilten Ressourcen können freigegeben werden, aber die Prozessverwaltung muss sie noch kennen
- Besonders der *exit status* muss erhalten bleiben



[by Charlie Llewellin, CC BY 2.0]

```
me@unix:~> ./wait &
me@unix:~> ps
  PID TTY          Zeit CMD
 4014 pts/4    00:00:00 bash
17892 pts/4    00:00:00 wait
17895 pts/4    00:00:00 wait <defunct>
17897 pts/4    00:00:00 ps
me@unix:~> Exit status: 42
```

Beispielprogramm der the vorigen Folie während der Wartezeit von 5 Sekunden

Zombies werden von `ps` als `<defunct>` gekennzeichnet

Unix-Prozesse im Detail: `execve()`

Systemaufruf:
```
int execve (const char *command,
            const char *args[ ], const char *envp[ ])
```

- Lädt und startet das Programm, das im “command”-Parameter übergeben wird
- Kehrt nur im Fehlerfall zurück
  - z.B. Programm existiert nicht, keine Zugriffsrechte, …
- Ersetzt den kompletten Adressraum des aufrufenden Prozesses
  - ***aber es bleibt der selbe Prozess!***
  - Identische PID, PPID, offene Dateien, ...
- Die C-Library (libc) stellt einige komfortable Funktionen bereit, die intern `execve` aufrufen:
  `execl`, `execv`, `execlp`, `execvp`, … (lest die *manpages*!)

Verwendung von exec()

Universität Bamberg

```
... /* includes, main() { … */
char cmd[100], arg[100];

while (1) {
  printf ("Command?\n");
  scanf ("%99s %99s", cmd, arg);
  pid = fork(); /* Prozess wird dupliziert!
                   Beide laufen von hier aus weiter. */
  if (pid > 0) {
    int status;
    if (wait(&status) == pid && WIFEXITED(status))
      printf ("Exit Status: %d\n", WEXITSTATUS(status));
  }
  else if (pid == 0) {
    /* Jetzt wird der Kindprozess durch ein
       anderes Programm ersetzt! */
    execlp(cmd, cmd, arg, NULL);
    printf ("exec failed\n");
  }
...
}
```

Diskussion: schnelle Prozesserzeugung

- Kopieren des Adressraums benötigt viel Zeit
  - Besonders, wenn direkt nach `fork()` dann `exec..()` aufgerufen wird
    ➛ ***komplette Zeitverschwendung!***
- Historische Lösung: `vfork`
  - Der Elternprozess wird suspendiert, bis der Kindprozess `exec..()` aufruft oder mit `_exit()` terminiert
- Der Kindprozess verwendet einfach direkt Code und Daten seines Elternprozesses (keine Kopie!)
  - Dabei darf der Kindprozess ***keine Daten ändern!***
  - Manchmal nicht so einfach: z.B. nicht `exit()`, sondern `_exit()` aufrufen
- Moderne Lösung: ***copy on write***
  - Eltern- und Kindprozess ***nutzen die selben Code- und Datensegmente*** mit Hilfe von virtuellem Speicher und der *memory management unit* (MMU)
  - Ein Segment wird nur kopiert, wenn der Kindprozess Daten darin ändert
  - **Nicht** der Fall, wenn `exec..()` direkt nach `fork()` aufgerufen wird
  - `fork()` mit *copy on write* ist *fast* so schnell wie `vfork()`

Diskussion: warum kein `forkexec()`?

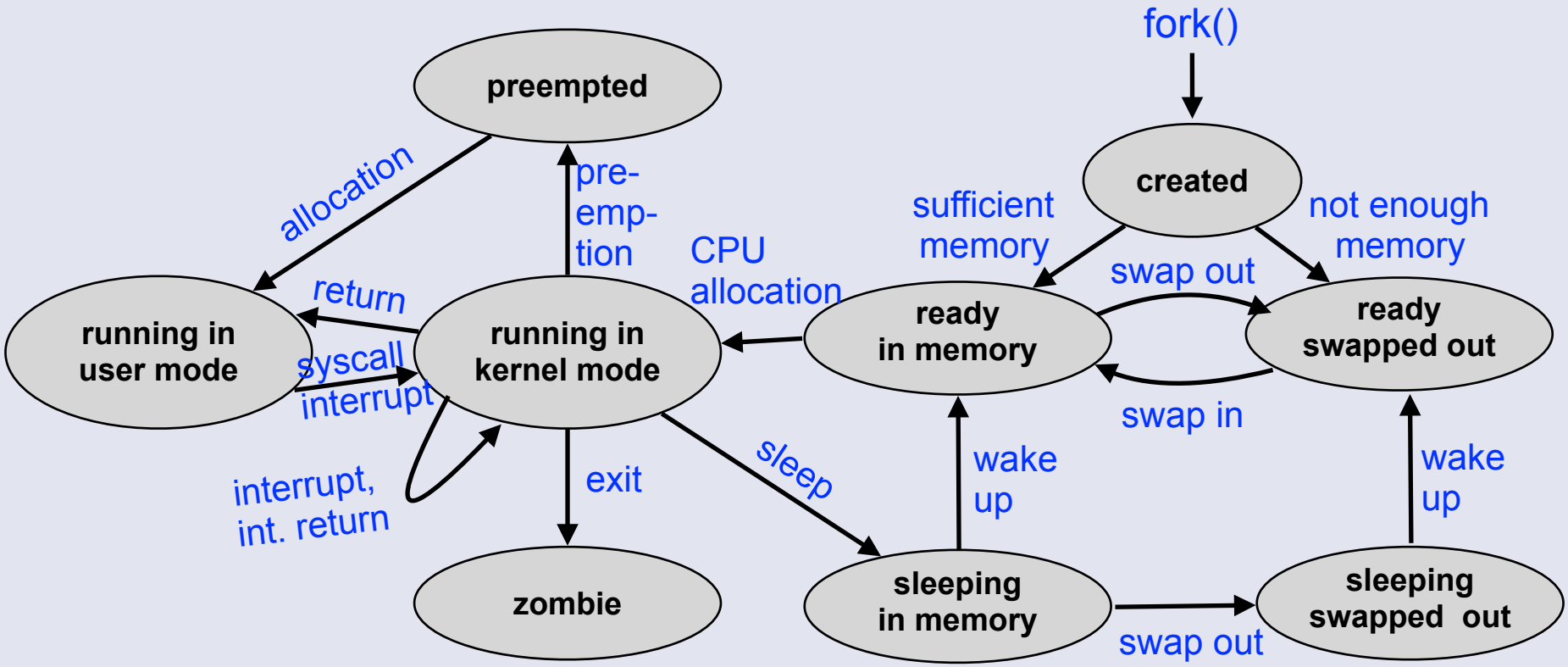
- Der Elternprozess hat eine bessere Kontrolle, wenn man separate Aufrufe von `fork` und `execve` bereitstellt:
  - Kann Operationen im Kontext des Kindprozesses ausführen (z.B. Dateien schließen)
  - Voller Zugriff auf die Daten des Elternprozesses

- Unix-Shells nutzen dies z.B. zum…
  - Umleiten von Ein-/Ausgabekanälen
  - Konfiguration von Interprozesskommunikation mit *pipes*

- Was sind die Alternativen zu `fork`?
  - Diskussion in einem HotOS-Papier von Microsoft aus dem Jahr 2019 [6]

Unix-Prozesszustände

- "Etwas" komplexer als unser ursprüngliches einfaches Modell…



CPU-Zuteilung

- Zuteilung (*scheduling*) ermöglicht die Koordination nebenläufiger Prozesse

- Grundsätzliche Fragen:
  - Welche Ereignisse sollten zu Verdrängung führen?
  - In welcher Reihenfolge sollten Prozesse ausgeführt werden?

- Optimierungsziele eines ***Zuteilungsalgorithmus***:
  - Benutzerorientiert ➛ kurze Reaktionszeiten
  - Systemorientiert ➛ optimale CPU-Ausnutzung

- Kein einzelner Zuteilungsalgorithmus erfüllt alle Anforderungen!

Prozesszustände (nochmal)

- **ready**: bereit, von der CPU ausgeführt zu werden
  - ein Prozess ist in der *ready (Warte-)Liste* für die CPU-Zuteilung
  - die Listenposition ist abhängig vom Zuteilungsalgorithmus

- **running**: die Ressource "CPU" ist dem Prozess zugeteilt
  - ein Prozess rechnet: *CPU burst*
  - es gibt *nur einen laufenden Prozess pro CPU* zu jedem gegebenen Zeitpunkt

- **blocked**: wartet auf ein Ereignis
  - ein Prozess führt Ein- oder Ausgabe durch: *I/O burst*
  - wartet auf das Eintreten mindestens einer Bedingung

Zustandsübergangspunkte

- Jeder Übergang zum READY-Zustand aktualisiert die **Warteschlange** der CPU
  - eine Entscheidung über die Einreihung des Prozesses erfolgt
  - das Ergebnis hängt von der ***CPU-Zuteilungsstrategie*** des Systems ab
- ***Zuteilung*** und ***Wiederzuteilung*** werden ausgeführt…
  1. nachdem ein neuer Prozess erzeugt wurde
  2. wenn ein Prozess die CPU abgibt
  3. wenn ein Ereignis, auf das ein Prozess wartet, eintritt

- Ein Prozess kann ***gezwungen werden***, die CPU abzugeben:
  → ***präemptive Zuteilung***, z.B. durch einen Zeitgeber-Interrupt

Zuteilungsstrategie: First-Come First-Served – FCFS

- Ein einfacher und fairer (?) Algorithmus: "first come first served"
- Reihenfolgekriterium ist die *Ankunftszeit* eines Prozesses
- Algorithmus ist *nicht präemptiv* und erwartet *kooperierende Prozesse*

Prozess	Zeiten					
	Ankunft	Benötigte Ausführungszeit T_s	Start	Ende	Laufzeit T_r	T_r/T_s
A	0	1	0	1	1	1,00
B	1	100	1	101	100	1,00
C	2	1	101	102	100	**100,00**
D	3	100	102	202	199	1,99
Durchschnitt						26,00

- Beispiel:
  - Die normalisierte Laufzeit (T_r / T_s) von C ist hoch im Vergleich zu seiner benötigten Ausführungszeit (*service time*) T_s

Diskussion: FCFS – "Konvoi-Effekt"

- Dieses Problem betrifft kurz laufende, **I/O-intensive Prozesse**, die auf lang laufende, **CPU-intensive Prozesse** folgen
  - Prozesse mit **langen** *CPU bursts* profitieren davon
  - Prozesse mit **kurzen** *CPU bursts* sind benachteiligt
- FCFS minimiert die Anzahl von *Kontextumschaltungen*
- Dieser **Konvoi-Effect** erzeugt aber eine Reihe von Problemen:
  - Lange Antwortzeiten
  - Niedriger I/O-Durchsatz
- Wenn das System einen Mix von CPU- und I/O-intensiven Prozessen ausführt, ist FCFS kein brauchbarer Ansatz
  - Typischerweise nur in Systemen mit Stapelverarbeitung (*batch processing)* verwendet

Round Robin (RR)

- Reduziert die Benachteiligung von Prozessen mit kurzen *CPU bursts*:
  - die verfügbare Prozessorzeit wird in ***Zeitscheiben*** aufgeteilt
- Wenn eine Zeitscheibe aufgebraucht ist, ***kann*** eine Prozessumschaltung stattfinden
  - der unterbrochene Prozess wird ans Ende der *ready list* gesetzt
  - der nächste Prozess wird nach FCFS aus der *ready list* ausgewählt
- Basis für garantierten Zugriff auf die CPU:
  Zeitgeber erzwingt einen Interrupt am Ende jeder Zeitscheibe
- Die Effizienz dieses Ansatzes hängt insbesondere von der gewählten Länge der Zeitscheibe ab:
  - zu lang ➛ round robin degeneriert zu FCFS
  - zu kurz ➛ sehr hoher *overhead* für häufige Prozessumschaltungen

Diskussion: RR – Performanceprobleme

Universität Bamberg

- I/O-intensive Prozesse beenden ihren *CPU burst* vor dem Ende ihrer Zeitscheibe
  - sie blockieren und werden wieder ans Ende der *ready list* gesetzt, wenn ihr *I/O burst* beendet ist
- CPU-intensive Prozesse dagegen nutzen ihre Zeitscheibe vollständig
  - sie werden dann verdrängt und direkt wieder ans Ende der *ready list* gesetzt
- Die Menge an CPU-Zeit, die Prozesse erhalten, ist daher ungleich verteilt ➛ CPU-intensive Prozesse erhalten größeren Anteil
  - I/O-intensive Prozesse werden nicht so gut bedient, daher ist die Ausnutzung von I/O-Geräten niedrig
  - Die *Varianz der Antwortzeit* von I/O-intensiven Prozessen steigt

Shortest Process Next (SPN)

- Reduziert die Benachteiligung kurzer *CPU bursts* bei FCFS: "der kürzeste kommt zuerst dran…"
  - dies erfordert Wissen über die Prozesslaufzeiten
  - keine Präemption
- Das Hauptproblem ist die ***Vorhersage der Laufzeiten***
  - Batchverarbeitung:
    der Programmierer annotiert die benötigte *Zeitschranke*
  - Interaktive Verarbeitung:
    Zeitschranke wird abgeschätzt basiert auf vorherigen *CPU burst*-Längen des Prozesses
- Antwortzeiten werden deutlich reduziert und die Gesamtsystemleistung verbessert
  - Aber: Gefahr des ***Verhungerns*** CPU-intensiver Prozesse

Diskussion: SPN – Gewichtung von bursts

- *CPU bursts*, die weiter in der Vergangenheit liegen, sollten geringer gewichtet werden:

$$S_{n+1} = \alpha \cdot T_n + (1-\alpha) \cdot S_n$$

  - Werte des konstanten Gewichtungsfaktors α: 0 < α < 1
  - gibt die *relative Gewichtung* einzelner *CPU bursts* in der Zeitlinie des Prozesses an

Dieser ***statistische Ansatz*** heißt auch **exponentielle Glättung**

- Rekursives Lösen führt zu:

$$S_{n+1} = \alpha T_n + (1-\alpha)\alpha T_{n-1} + \ldots + (1-\alpha)^i \alpha T_{n-i} + \ldots + (1-\alpha)^n S_1$$

$$S_{n+1} = \alpha \cdot \sum_{i=0}^{n-1} (1-\alpha)^i T_{n-i} + (1-\alpha)^n S_1$$

- für α = 0.8: $S_{n+1} = 0.8T_n + 0.16T_{n-1} + 0.032T_{n-2} + 0.0064T_{n-3} + \ldots$

Ein weites Feld…

- Es gibt viele weitere Zuteilungsverfahren…
  - Besonders relevant im Bereich der *Echtzeitsysteme*
  - Hier nicht weiter betrachtet!

- Bei Interesse: mehr Details in [1] Kapitel 7–9

Referenzen

[1] Remzi Arpaci-Dusseau, Andrea Arpaci-Dusseau
Operating Systems: Three Easy Pieces
https://pages.cs.wisc.edu/~remzi/OSTEP/

[2] Peter H. Salus
A Quarter Century of Unix, Addison-Wesley 1995
ISBN-13: 978-0201547771

[3] Brian Kernighan
UNIX: A History and a Memoir
Independently published 2019, ISBN-13: 978-1695978553

[4] W. Stevens, Stephen Rago
Advanced Programming in the UNIX Environment
3rd Edition, Addison-Wesley 2013, ISBN-13: 978-0321637734

[5] Dennis M. Ritchie and Ken Thompson
The UNIX time-sharing system
Commun. ACM 17, 7 (July 1974), 365–375

[6] Andrew Baumann, Jonathan Appavoo, Orran Krieger, Timothy Roscoe
A fork() in the road
ACM HotOS ’19