Block 2: Prozesse

Was ist ein Prozess?

Ein Prozess ist ein **Programm in Ausführung**. Er besteht aus:

1. Programmcode (Textsegment)
2. Daten (globale/statische Variablen)
3. Heap (dynamischer Speicher, z. B. über `malloc`)
4. Stack (Funktionsaufrufe, lokale Variablen)
5. CPU-Zustand (Program Counter, Register)
6. Vom Betriebssystem verwaltete Metadaten (PID, Zustand, offene Dateien usw.)

Ein Programm ist **passiv** (Datei auf der Festplatte); ein Prozess ist **aktiv** (laufende Instanz im Speicher).

Programm vs. Prozess

Ein Programm ist statisch und auf der Festplatte gespeichert.
Ein Prozess ist dynamisch, im Speicher geladen, und aus einem Programm können mehrere Prozesse entstehen.

Warum Prozesse existieren

- **Isolation**: ein Prozess kann keinen anderen beschädigen
- **Multiplexing**: mehrere Programme teilen sich eine CPU
- **Schutz**: durch Benutzer- und Kernelmodus
- **Abstraktion**: jeder Prozess verhält sich, als würde er die Maschine exklusiv besitzen

Prozesszustände und Übergänge

$$\text{created} \to \text{ready} \to \text{running} \to \text{blocked} \to \text{ready} \to \text{running} \to \text{exited}$$

- **ready**: lauffähig, wartet auf die CPU
- **running**: wird aktuell auf der CPU ausgeführt
- **blocked**: wartet auf ein Ereignis (I/O, Signal, Kindprozess)
- **exited**: Prozess ist beendet

`fork()` und `exec()`

`fork()` erzeugt einen neuen Prozess durch Duplizieren des aufrufenden Prozesses. Rückgabewerte:

- `0` im Kindprozess
- PID des Kindprozesses (> 0) im Elternprozess
- `-1` bei Fehler (kein Kind erzeugt)

Nach `fork()` laufen Eltern- und Kindprozess unabhängig weiter; die Ausführungsreihenfolge ist undefiniert.
Nach n aufeinanderfolgenden `fork()`-Aufrufen existieren bis zu 2^n Prozesse.
`exec()` ersetzt das aktuelle Programm im Prozess, behält jedoch PID und Prozessmetadaten bei. Nach erfolgreichem `exec()` existiert das alte Programm nicht mehr.

Prozessbeendigung, Zombies, Waisen

Ein Prozess endet durch `exit()`, Rückkehr aus `main` oder durch ein Signal.

- **Zombie**: Kindprozess ist beendet, Elternprozess hat noch kein `wait()` aufgerufen

- Zombies werden problematisch, wenn Elternprozesse nie `wait()` aufrufen und viele Kinder erzeugen
- **Waisenprozess**: Elternprozess endet, während das Kind noch läuft; Waisen werden von `init/systemd` adoptiert

Blocked vs. Ready

Blockierte Prozesse können selbst bei freier CPU nicht laufen, da sie auf ein Ereignis warten.
Bereite Prozesse können sofort laufen, warten jedoch ausschließlich auf die CPU.

Prozessgruppen

Eine Prozessgruppe ist eine Menge zusammengehöriger Prozesse mit einer PGID, die der PID des Gruppenleiters entspricht.
Prozessgruppen erlauben dem Betriebssystem, zusammengehörige Prozesse (z. B. Pipelines) gemeinsam zu steuern.

Grundlagen der Ablaufplanung (Scheduling)

Scheduling entscheidet, welcher bereite Prozess wann und wie lange die CPU erhält.

- **Scheduler**: wählt den nächsten Prozess aus
- **Dispatcher**: führt den Kontextwechsel aus

Präemptives Scheduling

UNIX/Linux verwendet präemptives Scheduling, das durch Timer-Interrupts erzwungen wird.
Round-Robin-Scheduling sorgt durch Zeitscheiben für faire CPU-Verteilung.