

# Prozessverwaltung

Ein **Prozess** ist ein Programm in Ausführung.

## Prozesse erzeugen

Bei der Erzeugung eines Prozesses bekommt dieser eine eindeutige ID.

### Windows

- Die createProcess-Methode wird aufgerufen
- In 6 Phasen wird der Prozess initialisiert

### Linux

- Die fork-Methode erzeugt eine exakte Kopie des aktuellen Prozesses
- Der Kindprozess wird mit eigenen Inhalten gefüllt, bekommt eine eigene P-ID → [aushöhlen und neu befüllen](#)

## Prozesskontrollblock

In einem Prozesskontrollblock fasst das Betriebssystem alle zu einem einzelnen Prozess gehörenden Informationen zusammen.

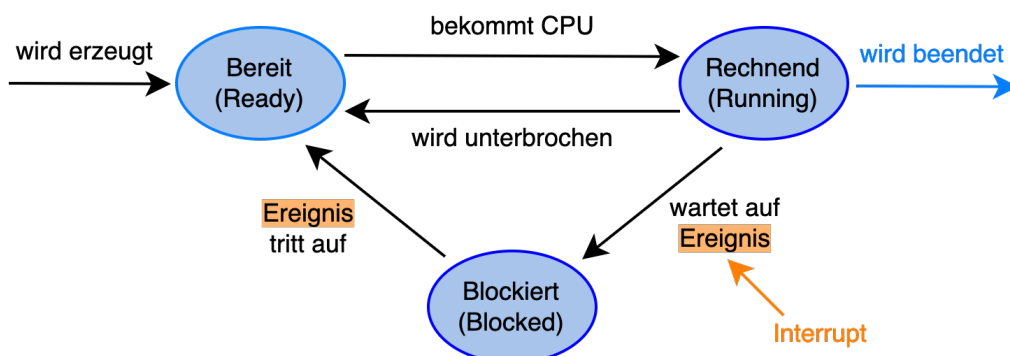
[ID](#), [Besitzer](#), [Programm-Ausführdatei](#), [Zustand](#), [Liste der geöffneten Dateien](#)

## Prozestabelle

In einer Prozesstabelle fasst das Betriebssystem alle Informationen aller erzeugten Prozesse zusammen. Es stehen alle Prozesskontrollblöcke untereinander. (z.B. Task-Manager)

## Prozesszustände

Wir unterscheiden drei Prozesszustände.



Der Scheduler

entscheidet, welcher Prozess die CPU als nächstes bekommt.

---

## Thread

Unter einem Thread versteht man einen Teil eines Prozesses, der einen unabhängigen Kontrollfluss repräsentiert.

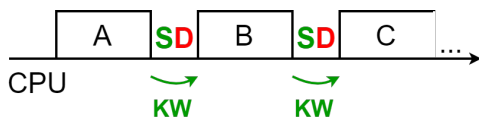
Threads wechseln sich durch **Threadkontextwechsel** immer wieder auf der CPU ab.

Dabei gibt es auch einen Threadscheduler.

Jeder Prozess hat mindestens einen Thread.

## Scheduling

Unter Scheduling versteht man die Tätigkeit des Aufteilens der verfügbaren Prozessorzeit (CPU-Zeit) auf alle Prozesse.



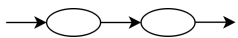
Der **Scheduler** führt die Scheduling-Tätigkeit durch.

Der **Dispatcher** entzieht bei einem Kontextwechsel dem derzeit aktiven Prozess die CPU und teilt sie dem nächsten Prozess zu.

Die Zeit, die ein Prozess zusammenhängend auf der CPU hat, wird als **Quantum** bezeichnet. Als Programmierer hat man keine Ahnung, wie lang ein solches Quantum auf einer jeweiligen CPU dauert und wie viele Befehle man innerhalb von einem Quantum abarbeiten kann.

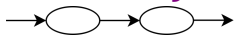
## Scheduling-Verfahren

### First Come-First Serve



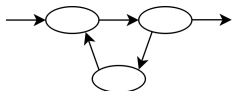
→ Die Prozesse laufen nacheinander vollständig auf der CPU ab.

### Shortest Job First



→ Der kürzeste Prozess wird als erstes abgearbeitet. Die Zeit des Prozesses muss dafür vom Programmierer angegeben werden.

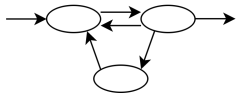
### Shortest Remaining Time Next



→ Es bekommt der Prozess die CPU, der die kürzeste Restlaufzeit besitzt (Kontextwechsel, falls

Prozess warten muss).

## Round-Robin



→ Nach Ablauf eines Quantums wird ein Kontextwechsel zu einem Prozess mit der nächst-höheren P-ID und dem Zustand "Ready" gemacht.

→ Basiert auf einer ringförmig verketteten Liste (Next des letzten Elements zeigt wieder auf das erste).

## Priority Scheduling

→ gleiches Verhalten wie beim Round Robin, nur werden zusätzlich Prioritäten eingeführt.

→ Es werden zuerst Prozesse mit hoher Priorität abgearbeitet.

→ Prioritäten werden nach bestimmter Zeit runtergestuft.

## Synchronisation

Bei der Synchronisation geht es um Mechanismen zur Vermeidung von Problemen, die bei der nebenläufigen Ausführung von Prozessen oder Threads in Verbindung mit gemeinsam genutzten Betriebsmitteln entstehen können.

Ein **Betriebsmittel** ist eine beliebige Hard- oder Software-Ressource, z.B. eine Variable, ein Drucker oder eine Datei.

Unter **Nebenläufigkeit** versteht man die quasi-parallele Ausführung von Befehlen unterschiedlicher Prozesse oder Threads auf der CPU.

Unter **Race Conditions** versteht man Situationen, bei denen zwei oder mehr Prozesse ein oder mehrere Betriebsmittel gemeinsam nutzen, und das Ergebnis der Ausführung von der zeitlichen Reihenfolge der Zugriffe der beteiligten Prozesse oder Threads auf das Betriebsmittel abhängt.

→ z.B. nutzen einer gleichen Variable (Kap. 3.2.11.1.2)

Unter einem **kritischen Abschnitt** versteht man Programmteile, die während ihrer Ausführung auf der CPU nicht durch kritische Abschnitte anderer Prozesse oder Threads unterbrochen werden dürfen, sofern die beteiligten Prozesse oder Threads auf gemeinsam genutzte Betriebsmittel zugreifen.

## Aktives Warten

Aktives Warten ist das ständige Abfragen eines Sperrkennzeichens am Eingang eines kritischen Abschnitts.

```
while (lock == 1); // Tue nichts
```

```
lock = 1;  
// kritischer Abschnitt!  
lock = 0;
```

### Problem:

Ein Kontextwechsel im ungünstigsten Moment (nach `while (lock == 1);` und vor Setzen der lock-Variable `lock = 1;`) führt dazu, dass es trotzdem zu Fehlern kommen kann. Der kritische Abschnitt ist entsperrt, die Variable aber noch nicht.

## TSL

Mit dem TSL-Befehl passieren zwei Dinge als **atomare Aktion**:

**Atomare Aktion:** Nicht teilbar! KW kann entweder vor oder nach TSL Befehl passieren. Nicht zwischen dem Holen des Werts einer Speicherzelle und dem Überschreiben der Zelle mit dem Wert 1.

- Wert aus Speicherzelle X in Akkumulator kopieren
- Wert an Speicherzelle X mit 1 überschreiben (`lock = 1`)

Das Problem des **ungünstigsten Moments** ist gelöst.

Beim aktiven Warten wird CPU-Zeit vergeudet: Der Prozess ist 100% damit beschäftigt, die Bedingung zu prüfen und nichts zu tun.

## Semaphore

Unter einem Semaphor versteht man eine Datenstruktur, welche einen ganzzahligen Zähler sowie eine Warteschlange bereitstellt. Zusätzlich sind zwei **atomare** Operationen `P()` und `V()` auf diese Datenstruktur definiert.

Semaphor (0, 1)

**Mutex, binär**

**ein** Betriebsmittel

Semaphor ( $x > 1$ )

**Zählsemaphor**

**x** Betriebsmittel

`P()`

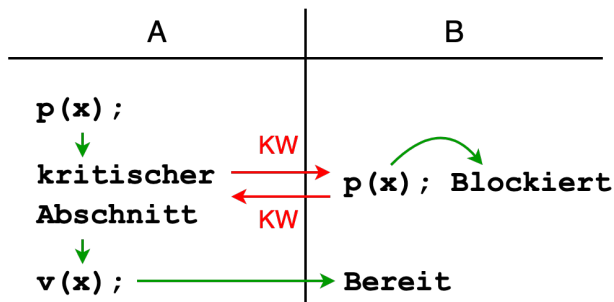
- Hat der Zähler einen positiven Wert, verringere ihn um 1
- Ansonsten blockiere den aufrufenden Prozess und füge ihn der Warteschlange hinzu

**V()**

- Ist die Warteschlange nicht leer, entblockiere den ersten Prozess
- Ansonsten erhöhe den Zähler um 1

## Wechelseitiger Ausschluss

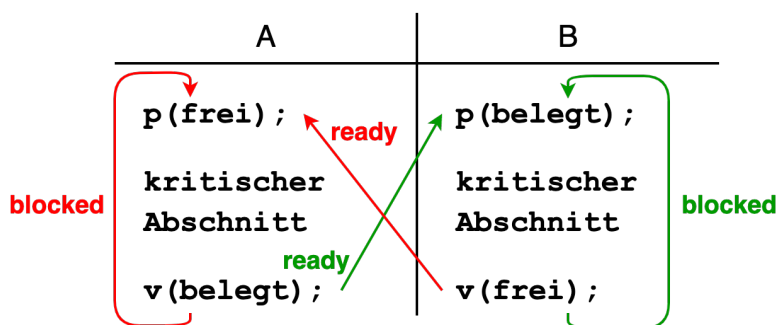
```
Semaphor x = new Semaphore(1);
```



Da A sich im kritischen Abschnitt befindet, wird B durch `x` blockiert (Zähler von `x` ist auf 0). Nachdem A den kritischen Abschnitt verlässt, wird B durch `v(x)` entblockiert.

## Reihenfolge durch Setzung

```
Semaphor frei = new Semaphor(1);
Semaphor belegt = new Semaphor(0);
```



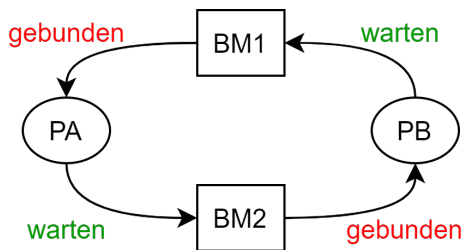
Bei zwei voneinander abhängigen Prozessen werden zwei Semaphore benötigt, sodass sich die Prozesse gegenseitig entblockieren.

## Deadlocks

Eine Menge von Prozessen befindet sich in einem Deadlock-Zustand, wenn jeder Prozess aus der Menge auf ein Ereignis wartet, das nur ein anderer Prozess der Menge auflösen kann.

**Beispiel:** Prozess A und B benötigen BM1 und BM2. A hat bereits BM1 an sich gebunden, B hat BM2. Beide Prozesse warten nun auf das jeweils andere Betriebsmittel.

## Betriebsmittelgraph



## Vier Bedingungen nach Coffman

1. Mutual exclusion condition  
→ Gegenseitiger Ausschluss  
→ es steht nur ein BM zur Verfügung
2. Wait for condition  
→ warten auf ein anderes BM
3. No preemption condition  
→ Den Prozessen können bereits gebundene BM nicht entrissen werden
4. Circular wait condition  
→ siehe Betriebsmittelgraph

Theoretisch könnten Deadlocks vermieden werden, indem jedes BM eine Nummer bekommt und immer das BM mit der niedrigeren Nummer zuerst angefordert wird. (Bsp. Bank: 3.2.12.4 Aufg. 3)

In der Praxis funktioniert das nicht: Zu Beginn ist nicht klar, welche BM vom Prozess benötigt werden.

Deadlocks kommen sehr selten vor und es ist aufwendig, sie zu erkennen (kostet viel Rechenzeit).  
Daher werden sie in gängigen Betriebssystemen ignoriert.

## Interprozesskommunikation

In Betriebssystemen müssen Möglichkeiten geschaffen werden, damit Prozesse und Threads miteinander kommunizieren können:

- gemeinsame Daten- und Speicherbereiche
- Sockets → **Netzwerkverbindung**; Pipes → **Ausgabe von P1 = Eingabe von P2**

Revision #16

Created 12 August 2022 08:27:25 by Martin Tienken

Updated 6 September 2022 09:33:00 by Martin Tienken