

# Prozessverwaltung

- Prozesskontext
- Prozesse erzeugen
- Prozesskontrollblock
- Prozesstabelle
- Prozesszustände
- Prozessverwaltung
- Threads
- Scheduling
- Synchronisation
- Semaphore
- Deadlocks
- Interprozesskommunikation

# Prozesskontext

**Prozess** ist Programm in Ausführung. **Prozesskontext** hat Informationen, die Prozess bei Ausführung auf CPU benötigt.

- Zum Kontext eines Prozesses gehören unter anderem:
  - Die Werte in den betreffenden Registern der CPU (Program Counter, Instruction Register, Stack Register, Flags, etc.)
  - Die Belegung des Caches mit Befehlen und Daten des Prozesses
  - Die Belegung des Hauptspeichers mit Programmtext und Daten des Prozesses

**Prozess-Kontextwechsel** auf CPU sind alle Tätigkeiten, um von auf CPU aktiven Prozess A auf Prozess B zu kommen

- Dafür werden Tätigkeiten vom Steuerwerk der CPU in Zusammenarbeit mit dem Betriebssystem durchgeführt:
  - Sichere alle notwendigen Registerinformationen des scheidenden Prozesses A an einer bekannten Stelle (damit sie von dort später wiederhergestellt werden können).
  - Lade alle notwendigen Registerinformationen des neuen Prozesses B in die entsprechenden Register auf der CPU.
  - Lade alle notwendigen Befehle und Daten des neuen Prozesses B in den Cache.

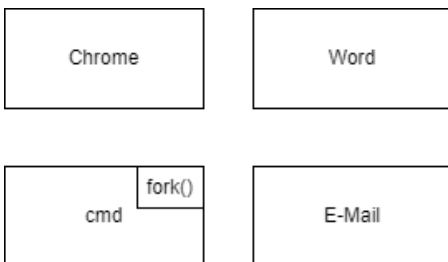
Es ist leicht verständlich, dass jeder Kontextwechsel eine gewisse Zeit für seine Durchführung beansprucht.

# Prozesse erzeugen

Jeder Prozess erhält zur Unterscheidung und Verwaltung direkt bei seiner Erzeugung eine eindeutige **Prozess-ID**

- Während Laufzeit eines BS werden Prozesse erzeugt, abgearbeitet, unterbrochen, weiter abgearbeitet und beendet
- Bei Start eines Rechners passiert folgendes:
  - Wird Rechner eingeschaltet, startet BS, wobei ein Mechanismus den ersten Prozess erzeugt (PID = 0, Leerlaufprozess)
  - Ausgehend vom ersten Prozess starten weitere Prozesse, die zum BS gehören für Erfüllung der Aufgabe des BS
  - **dann startet Anwender Anwendungsprogramme, die jeweils als eigener Prozess vom BS verwaltet werden**
- allgemein hängt es vom Betriebssystem ab, wie genau die Erzeugung eines Prozesses erfolgt:
  - Unter Unix/Linux gibt es dazu einen Systemaufruf *fork* und ist sehr einfach
  - unter Windows heißt dieser Systemaufruf *CreateProcess* und ist sehr kompliziert

## Fork - Linux



- Systemaufruf **fork** macht von aufrufenden Prozess (**Elternprozess**) Kopie (**Kindprozess**)
- **Kindprozess** erhält eigene Prozess-ID, übernimmt Informationen des **Elternprozesses**
- **Kindprozess** ist nach **fork** unabhängig vom **Elternprozess** eigenständige Systeminstanz
- Sowohl **Eltern**- als auch **Kindprozess** laufen nach dem "klonen" an gleicher Stelle weiter
  - **fork** liefert Rückgabewert und auf das **fork** folgende Anweisung wird ausgeführt
- Rückgabewert von **fork** > 0; **fork** = 0: **Kindprozess**; **fork** < 0: Es ist Fehler aufgetreten.

## CreateProcess - Windows

- Der Systemaufruf *CreateProcess* besteht aus sechs Hauptphasen bei der Prozesserzeugung.

- Anschließend ist der neue Prozess komplett erstellt und wartet auf die Zuteilung der CPU.

1.	<b>beginnt</b> mit dem <b>Öffnen</b> der EXE-Datei.
2.	<b>erstellt</b> notwendige <b>Verwaltungsobjekte</b> .
3.	<b>erstellt</b> notwendige <b>Verwaltungsobjekte</b> .
4.	<b>informiert</b> das Windows-Subsystem über den hier neu <b>erstellten Prozess</b> .
5.	<b>leitet</b> die <b>Ausführung</b> des neu erstellten Prozesses ein.
6.	nimmt alle abschließenden <b>Initialisierungen</b> vor.

# Prozesskontrollblock

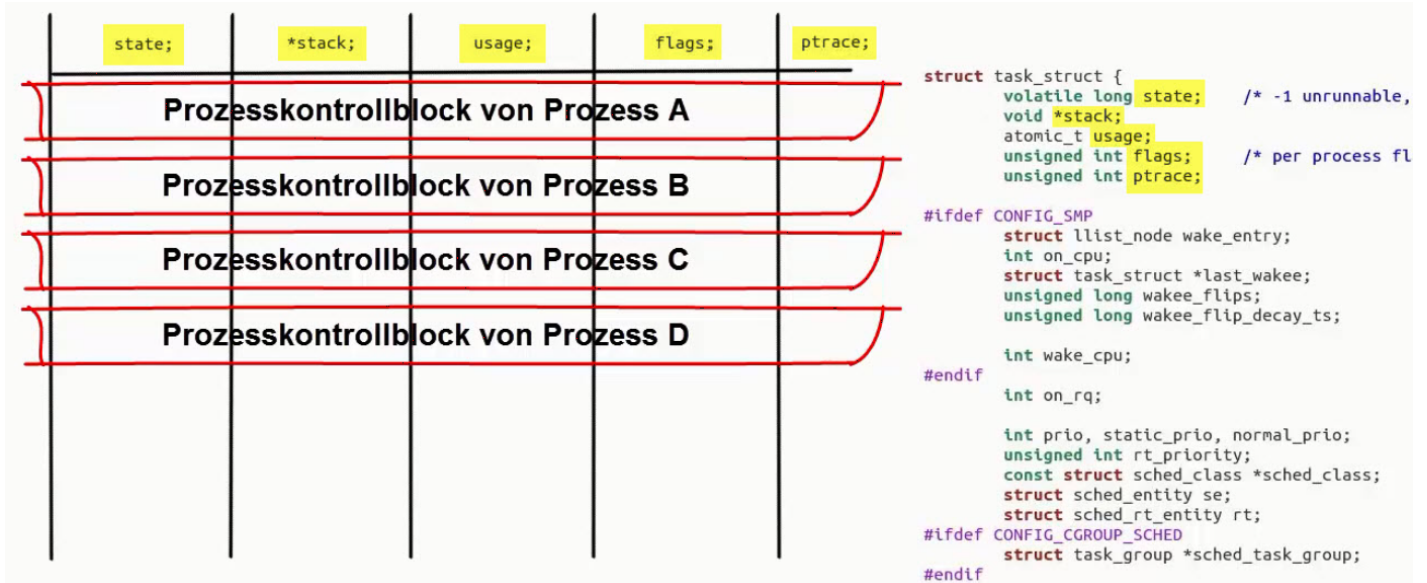
In **Prozesskontrollblock** fasst das Betriebssystem alle zu einem einzelnen Prozess gehörenden Informationen zusammen.

- Sobald ein neuer Prozess erzeugt wird, legt das Betriebssystem einen Prozesskontrollblock als Verwaltungsstruktur an.
  - **Für jeden Prozess existiert somit ein eigener PCB.**

<b>Windows</b>	jeder unter Windows erzeugter Prozess wird durch eine Instanz der Datenstruktur EPROCESS repräsentiert
<b>Linux</b>	die Deklaration läuft über die Datenstruktur <i>task_struct</i>

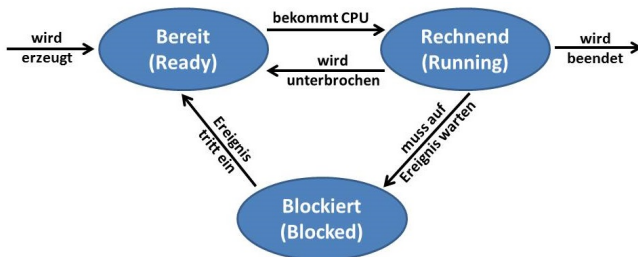
# Prozesstabelle

In der Prozesstabelle fasst das Betriebssystem alle Informationen aller erzeugter Prozesse zusammen. In der Praxis kann die Prozesstabelle ganz einfach als Liste aller Prozesskontrollblöcke realisiert werden.



# Prozesszustände

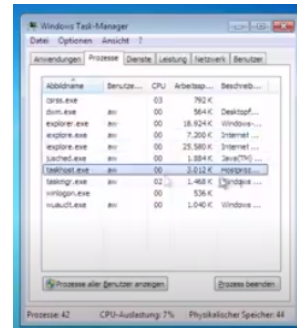
Wird ein Prozess innerhalb eines Betriebssystems erzeugt, so bedeutet dies nicht, dass er auch sofort auf der CPU ausgeführt wird.



- Direkt nach Erzeugung befindet sich Prozess im Zustand **Bereit**
  - wartet auf die Zuteilung der **CPU**
- Sobald er die CPU bekommt, wechselt er in den Zustand **Rechnend**.
  - **CPU** kann wieder entzogen werden und Prozess zurück in Bereit
- Prozess wechselt vom Zustand **Rechnend** in den Zustand **Blockiert**
  - hat Befehl ausgeführt, der auf sich *warten* lässt (z.B: E/A-Geräte)
  - **CPU** ist für einen anderen Prozess bereit
- Tritt das gewünschte Ereignis ein, meldet sich beispielsweise das E/A-Gerät mit dem *Ergebnis*
  - **ISR** merkt, dass Festplatt was fertig gestellt hat für blockierten Prozess
- Prozess wechselt durch **ISR** vom Zustand **Blockiert** in den Zustand **Bereit**. Hier wartet er wieder auf die Zuteilung der **CPU**.

# Prozessverwaltung

- Eine der **Aufgaben eines Betriebssystems** ist ja die Verwaltung der erzeugten Prozesse
- Dabei kommen **Prozesskontrollblock** und **Prozestabelle** zum Einsatz.



## Prozessverwaltung aus Admin-Sicht unter Windows

- Auch viele Informationen aus dem **Prozesskontrollblock** eines Prozesses werden hier zur Laufzeit des Prozesses sichtbar.

## Prozessverwaltung aus Admin-Sicht unter Linux

- auf **Kommandozeile** gibt es Programme, womit **Administrator** Aufschluss über die aktuell gestarteten Prozesse bekommt.
- ps | pstree | top | htop



# Threads

**Thread** oder **Leichtgewichtigen Prozess** ist Teil eines Prozesses, der einen unabhängigen Kontrollfluss repräsentiert.

- Ein **Prozess** kann aus **mehreren Threads** bestehen, somit mehrere voneinander unabhängige, **nebenläufige Kontrollflüsse**

## BEISPIEL

- Ein Textverarbeitungsprogramm wird gestartet. Somit existiert auf dem Computersystem ein Textverarbeitungs-Prozess.
  - Dieser Prozess startet intern mehrere Threads, die jeweils bestimmte Aufgaben übernehmen:
    - Thread 1: realisiert den Texteditor. Er reagiert also auf Eingaben des Users mit Tastatur oder Maus.
    - Thread 2: realisiert eine "Alle 10 Minuten automatisch im Hintergrund speichern"-Funktion.
    - Thread 3: realisiert die automatische Rechtschreibprüfung in kleinen zeitlichen Abständen

- Das Thread-Konzept wird von Betriebssystemen unterstützt und können deshalb auf **Kernel-Ebene** realisiert sein
- Unterstützt Betriebssystem keine Threads, so kann auf der **User-Ebene** ein spezielles Programm Bereitstellung übernehmen
- Wird die Java-Laufzeitumgebung auf einem Betriebssystem installiert, laufen Threads auf beiden Ebenen
- Bei Systemen ohne jegliche Thread-Unterstützung ist ein Prozess praktisch gleichzusetzen mit einem Thread.

**Wenn ein Prozess nur einen Thread besitzt**, ist dieser betrachtete Prozess praktisch gleichzusetzen mit seinem Thread.

Bei **Thread-Kontextwechsel** auf der CPU wird aktiver Thread A durch einen anderen Thread B ersetzt.

Vorteile	Nachteil
----------	----------

- Thread-Kontextwechsel leichter als **Prozess-Kontextwechsel**.
- Threads eines Prozesses haben Zugriff auf alle Betriebsmittel, welche Prozess zugeordnet sind.
- Der Anwendungsprogrammierer kann die Funktionalität der Gesamtanwendung in unterschiedliche Threads aufteilen, welche jeder für sich einfacher zu implementieren ist.

- Threads eines Prozesses haben Zugriff auf Betriebsmittel, welche Prozess zugeordnet sind.
- Anwendungsprogrammierer müssen über Kenntnisse bei der Programmierung von Threads verfügen.

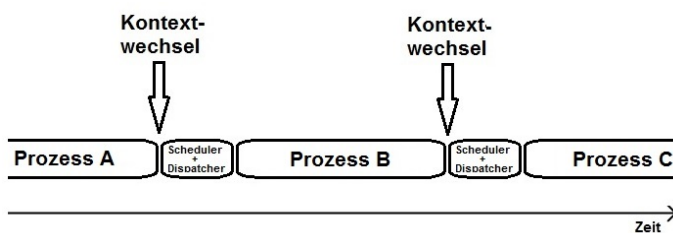
# Scheduling

Beim Scheduling geht es um die Zuteilung des Betriebsmittels *CPU* zu den einzelnen Prozessen.

Unter **Scheduling** versteht man die Tätigkeit des Aufteilens der verfügbaren Prozessorzeit auf alle Prozesse.

Unter **Scheduler** versteht man den Teil des Betriebssystems, welcher die Scheduling-Tätigkeit durchführt. Er ist zuständig zu entscheiden, welcher Prozess als nächstes die CPU bekommt. Umsetzung dieser Entscheidung obliegt dem Dispatcher.

**Dispatcher** entzieht bei Kontextwechsel dem aktiven Prozess die CPU, um sie dem nächsten Prozess zuzuteilen.

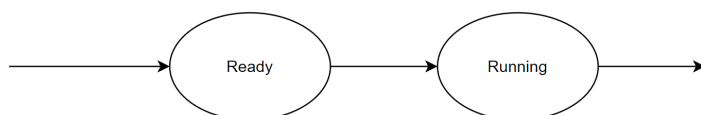


**Ziele:** Möglichst viele Prozesse werden in kurzer Zeit abgearbeitet.

**Effizienz:** zur Verfügung stehenden Ressourcen werden ausgelastet.

**Fairness:** Die Ressourcen werden den Prozessen gerecht zugeteilt, das heißt, kein Prozess wird dauerhaft vernachlässigt.

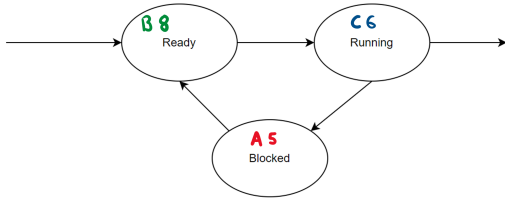
## Scheduling-Verfahren



### First Come First Serve (FCFS)

arbeitet Prozesse in Reihenfolge ihres Starts ab. Zuerst erzeugter Prozess darf auch als erstes in den Zustand *Rechnend* wechseln. Während Rechenzeit kann Prozess durch Interrupt unterbrochen werden. Direkt nach Interrupt setzt er Arbeit auf der CPU fort.

**Shortest Job First (kurz: SJF):** Von allen Prozessen auf System bekommt derjenige als erstes die CPU mit der kürzesten Laufzeit



### Shortest Remaining Time Next (kurz: SRTN)

ist eine Abwandlung des SJF-Verfahrens. Hierbei bekommt immer derjenige Prozess die CPU, welcher die kürzeste Restlaufzeit besitzt. Wenn Prozess A(5) aus Blocked hinaus kommt, kommt er vor Prozess B(8) wieder in Running, wenn Prozess C(6) von Running in Blocked wechselt.

### Round Robin (RR)

teilt die zur Verfügung stehende CPU-Zeit in kleine *Zeitscheiben*. Prozesse werden in eine initiale Reihenfolge gebracht und der erste Prozess bekommt die CPU, er darf diese genau bis zum Ablauf seines *Zeit-Quantums* nutzen, danach in der Reihenfolge.

### Priority Scheduling (kurz: PS)

Jedem Prozess wird eine *Priorität* zugewiesen. Der Prozess mit der höchsten Priorität bekommt als erstes die CPU.

# 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.

**Nebenläufigkeit** ist quasi-parallele Ausführung von Befehlen mehrerer Prozesse oder Threads mit Kontextwechsel auf CPU.

Beim **Race Conditions** nutzen zwei oder mehr Prozesse ein oder mehrere Betriebsmittel gemeinsam, wobei das Ergebnis der Ausführung von der zeitlichen Reihenfolge der Zugriffe der Prozesse oder Threads auf das Betriebsmittel abhängt.

kritischer Abschnitt sind Programmteile, die während Ausführung auf der CPU **nicht** durch kritische Abschnitte anderer Prozesse oder Threads unterbrochen werden dürfen, sofern Prozesse **auf gemeinsam genutzte Betriebsmittel zugreifen**.

## Aktives Warten - Prozesse und Threads synchronisieren

Unter **aktivem Warten** versteht man ständige Abfragen eines Sperrkennzeichens am Eingang eines kritischen Abschnitts.

- dafür wird das **Polling** genutzt, um ständig *Abfragen* durchzuführen
- um einen Prozess oder einen Thread zu sperren wird das genannte **Sperrkennzeichen**, was sich als *Sperrzeichen* äußert

```
public class Beispiel_Aktives_Warten {  
  
    static int counter = 0;        // gemeinsam genutztes Betriebsmittel  
    static int lock = 0;           // Sperrvariable  
  
    public static class Thread_A extends Thread {  
        public void run() {  
            do_something();        // unkritisch  
        }  
    }  
}
```

```

    count_from_10();          // kritisch !!!
    do_something_else();      // unkritisch
}

private void do_something() {
    // unkritischer Abschnitt
    System.out.println("Thread_A: unkritisch");
}

private void count_from_10() {
    // Vorsicht: kritischer Abschnitt!
    while (lock == 1);        // wenn lock auf 1 ist, dann ist das Thread in der While-Schleife
    // gefangen
    lock = 1; // wenn lock 0 war wird er hier auf 1 gesetzt und die Methode durchlaufen
    counter = 10;
    counter++;
    counter++;
    System.out.println("A-Counter: " + counter);
    lock = 0; // die methode ist durchlaufen, auch wenn zwischen durch ein anderer
    // Thread dran war, ab hier kann ein anderer Thread durchlaufen werden
    private void do_something_else() {
        // unkritischer Abschnitt
        System.out.println("Thread_A: wieder unkritisch");
    }
}

public static class Thread_B extends Thread {
    public void run() {
        System.out.println("Thread_B ist gestartet.");
        while (lock == 1);        // Semikolon beachten!
        lock = 1;
        counter = 20;
        counter++;
        counter++;
        System.out.println("B-Counter: " + counter);
        lock = 0;
    }
}

public static void main(String[] args) {
    Thread a = new Thread_A();
    Thread b = new Thread_B();
}

```

```

    p.start();
    p.start();
}

}

```

- Kontextwechsel im **ungünstigsten Moment** kann dazu führen, dass zwei oder mehr Prozesse oder Threads sich gleichzeitig in ihrem **kritischen Abschnitt befinden**.
  - es gibt Mechanismus, welcher eine Sperrvariable innerhalb einer **atomaren Aktion** abfragen und setzen kann
  - eine solche Aktion ist *atomar*, ausgeführt werden müssen und (logisch) *nicht unterbrochen* werden dürfen
- Eine Möglichkeit bietet dafür der **TSL-Assemblerbefehl**:
  - CPU sorgt bei der Ausführung des TSL-Befehls dafür, dass zwei Dinge innerhalb einer Aktion (**atomar!**) passieren, weder ein Kontextwechsel, noch ein Interrupt können den TSL-Befehl unterbrechen
    - **Kopiere** den aktuellen **Wert der lock-Variablen** aus *Speicherzelle 25* in das *Register R1*
    - **Schreibe** in *Speicherzelle 25* den Wert 1, setze also **lock = 1**.
  - **EQUAL-Befehl** vergleicht **Wert aus Register 1** mit der **Zahl 0**.
    - *Abhängig vom Ergebnis muss Prozess entweder in Schleife warten oder darf kritischen Bereich ausführen.*

```

10: TSL 25          ; Hier passieren zwei Dinge als atomare
Aktion:
    R1              ; --> Wert aus Speicherzelle 25 in Akkumulator
kopieren
                    ; --> Zahl 1 in Speicherzelle 25 schreiben (setze lock=1)
11: EQUAL #0       ; Prüfe: Ist ACC == 0? (Eigentlich: ist lock == 0?)
12: JUMP 10         ; Prüfung ergab FALSE
13: ...            ; Prüfung ergab TRUE

```

# Semaphore

- **Vorteil:** es wird aufs **Aktive Warten** verzichtet und somit wird weniger **CPU-Zeit** verschwendet, indem der wartende Prozess durch einen **Kontextwechsel** in den **Zustand Blockiert** gebracht wird

Datenstruktur, welche **ganzzahligen Zähler**, **Warteschlange** bereitstellt und zwei **atomare Operationen P() und V()** definiert.

- der **ganzzahliger Zähler** (int) kann keine negativen Werte annehmen.
- **Warteschlange** ist Datenstruktur, die nach dem FIFO-Prinzip arbeitet und als zur geordneten Aufnahme von Prozessen dient.
- Die **Operation P()** wird in einigen Quellen auch als **down()**-Operation betitelt
  - Hat die Zählervariable einen positiven Wert ( $>0$ ), dann verringere den Wert der Zählvariablen um 1.
  - Wenn Zählvariable den Wert 0 hat, dann blockiere aufrufenden Prozess und fügt ihn an das Ende ihrer Warteschlange
- analog **up()** anstatt **V()**.
  - wenn die Warteschlange nicht leer ist, dann entblockiere den ersten Prozess der Warteschlange.
  - ist die Warteschlange leer, dann erhöhe den Wert der Zählervariable um 1.
  - Mit "entblockieren" wird der erste Prozess aus der Warteschlange in den **Zustand "Bereit"** geändert.

ein **binären Semaphore** ist ein Semaphore, dessen ganzzahliger Zähler nur die Werte 0 oder 1 annehmen kann.

1. **Mutex** ist ein binären Semaphore, der mit einem **wechselseitigen Ausschluss** arbeitet und das **aktive Warten** ersetzt
  - Wenn ein Thread in **kritischen Abschnitt** befindet, darf andere Thread nicht in seinen **kritischen Abschnitt** eintreten
  - **Reihenfolgedurchsetzung** ist dabei sehr wichtig: Erst wenn erste Prozess Zugriff erledigt hat, darf der zweite
2. **Zählsemaphor** ist ein Semaphore, der den Zugriff auf eine begrenzte Anzahl **eines** Betriebsmittels regelt.
  - Der Zugriff auf das Betriebsmittel stellt deshalb einen **kritischen Abschnitt** dar und muss geschützt werden.



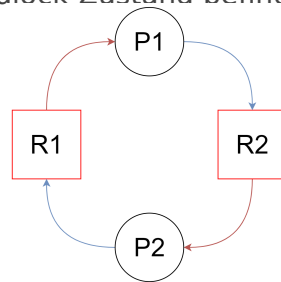
- **Erzeuger- / Verbraucherproblem:** Erzeuger erzeugt etwas, was Verbraucher verbraucht. Natürlich kann Verbraucher erst etwas verbrauchen, wenn zuvor Erzeuger etwas erzeugt hat. Erzeuger kann nur soviel erzeugen, bis alle Lagerplätze für Erzeugnis **belegt** sind. Dann muss Verbraucher wieder verbrauchen, bis Erzeuger wieder etwas erzeugen kann.

Ein Semaphore hat keinen ungünstigen Moment, weil die P() Methode atomar ist!

# 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 aus der Menge auslösen kann.

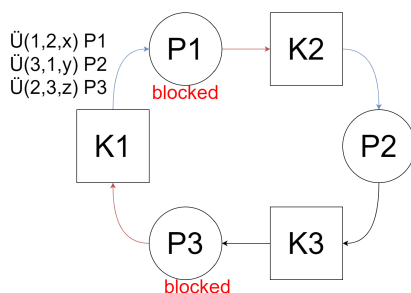
Wenn sich mehrere Prozesse in Deadlock-Zustand befinden, so sagt man auch vereinfachend:



**Es ist ein Deadlock aufgetreten.**

Vier Bedingungen damit ein Deadlock auftritt:

1. **Mutual exclusion condition:** Eine Ressource steht einem Prozess nur exklusiv zur Verfügung
2. **Wait for condition:** Prozesse warten und behalten dabei Kontrolle über bereits Ressourcen
3. **No preemption condition:** Ressourcen können Prozess nicht entrissen werden.
4. **Circular wait condition:** Es gibt eine zyklische Kette von Prozessen



**Prinzip des ersten Kontos**

- P1: Schnappt sich **K1** **Kontextwechsel**
- P2: Schnappt sich **K2** **Kontextwechsel**
- P3: Möchte **K1** haben ist aber belegt: *blockiert*; **Kontextwechsel**
- P1: Möchte **K2** haben ist aber belegt: *blockiert*; **Kontextwechsel**
- P2: schnappt sich **K3**: **Fetch, Decode, Execute**

---

### **Deadlocks ignorieren**

*"Es gibt keine Deadlocks, weil ich daran glaube, dass es keine Deadlocks gibt!",* sagt das Betriebssystem.

Zum Ignorieren von Deadlocks kommt in Betriebssystemen üblicherweise der **Vogel-Strauß-Algorithmus** zum Einsatz.

### **Deadlocks vermeiden**

Ein Betriebssystem könnte von vorneherein so konstruiert werden, dass Deadlocks gar nicht möglich sind.

### **Verhinderung von Deadlocks**

Prozess nur dann weitere Ressource zuzusprechen, wenn sichergestellt ist, dass **in der Zukunft** kein Deadlock entstehen kann

Allgemein ist davon auszugehen, dass die Erkennung, Vermeidung oder Verhinderung von Deadlocks sehr aufwendig ist. Eine ausnahmslose Abdeckung aller denkbaren Fälle scheint nahezu unmöglich.

# Interprozesskommunikation

Bei der **Interprozesskommunikation** geht es um den Austausch von Informationen zwischen zwei (oder mehr) Prozessen bzw. Threads. Damit alle Beteiligten die ausgetauschten Informationen in gleicher Weise verstehen können, sind bestimmte Regeln der Kommunikation einzuhalten, das sogenannte *Protokoll*.

## **Zwei Threads kommunizieren über gemeinsame Variablen**

→ Synchronisation Threads beim Zugriff auf gemeinsamen Datenbereiche wird erforderlich, da es zu kritischen Abläufen kommt

## **Zwei Prozesse kommunizieren über gemeinsame Speicherobjekte (Dateien)**

→ Prozess A erzeugt während Laufzeit Daten und speichert diese in einer Datei ab.

→ zweiter Prozess B liest Datei zu einem **späteren** Zeitpunkt ein und kann so die enthaltenen Informationen weiterverarbeiten.

→ sorgt Betriebssystem nur für eine Synchronisation beim *gleichzeitigen Zugriff* der beiden Prozesse auf die (z.B. über Semaphore).

→ Falls Prozess B Dateiinhalt ausliest, **bevor** Prozess A die Informationen hineingeschrieben hat, so ist nicht zu erwarten, dass Prozess B das erwartete Ergebnis berechnen kann

## **Zwei Prozesse kommunizieren über Shared Memory**

→ Man spricht von **Shared Memory**, wenn Teil des Hauptspeichers gemeinsam mehreren Prozessen zur Verfügung gestellt wird

## **Zwei Prozesse kommunizieren über Pipes**

→ *Einweg-Kommunikationskanäle*, die Prozess ermöglichen, Daten über das BS als Datenstrom an anderen Prozess zu übertragen

## **Zwei Prozesse kommunizieren über Sockets**

→ Sockets sind eine vom Betriebssystem bereitgestellte Kommunikationsmöglichkeit über TCP