9. Foliensatz Betriebssysteme

Prof. Dr. Christian Baun

Frankfurt University of Applied Sciences
(1971–2014: Fachhochschule Frankfurt am Main)
Fachbereich Informatik und Ingenieurwissenschaften
christianbaun@fb2.fra-uas.de

Lernziele dieses Foliensatzes

- Am Ende dieses Foliensatzes kennen/verstehen Sie...
  - was **kritische Abschnitte** sind
  - wie **Wettlaufsituationen** (Race Conditions) entstehen
    - welche **Konsequenzen** Wettlaufsituationen haben
  - den Unterschied zwischen **Kommunikation** und **Kooperation**
  - was **Synchronisation** ist
    - verschiedene Möglichkeiten mit **Signalisierung** eine Ausführungsreihenfolge der Prozesse festzulegen
    - wie mit **Blockieren** kritische Abschnitte gesichert werden können
  - welche **Probleme** beim Blockieren entstehen können
    - den Unterschied zwischen **Verhungern** und **Deadlocks**
  - die **Bedingungen**, die für die Entstehung von Deadlocks nötig sind
  - wie **Betriebsmittel-Graphen** die Beziehungen von Prozessen und Ressourcen darstellen
  - wie **Deadlock-Erkennung mit Matrizen** funktioniert

Übungsblatt 9 wiederholt die für die Lernziele relevanten Inhalte dieses Foliensatzes

Interprozesskommunikation (IPC)

- Prozesse müssen nicht nur Operationen auf Daten ausführen, sondern auch:
  - sich gegenseitig aufrufen
  - aufeinander warten
  - sich abstimmen
  - kurz gesagt: Sie müssen miteinander **interagieren**
- Bei **Interprozesskommunikation** (IPC) ist zu klären:
  - Wie kann ein Prozess Informationen an andere weiterreichen?
  - Wie können mehrere Prozesse auf gemeinsame Ressourcen zugreifen?

Frage: Wie verhält es sich hier mit Threads?

- Bei Threads gelten die gleichen Herausforderungen und Lösungen wie bei Interprozesskommunikation mit Prozessen
- Nur die Kommunikation zwischen den Threads eines Prozesses ist problemlos möglich, weil sie im gleichen Adressraum agieren

Kritische Abschnitte

- Laufen mehrere parallel ausgeführte Prozesse, unterscheidet man:
  - **Unkritische Abschnitte**: Die Prozesse greifen gar nicht oder nur lesend auf gemeinsame Daten zu
  - **Kritische Abschnitte**: Die Prozesse greifen lesend und schreibend auf gemeinsame Daten zu
    - Kritische Abschnitte dürfen nicht von mehreren Prozessen gleichzeitig durchlaufen werden
- Damit Prozesse auf gemeinsam genutzten Speicher ($\Longrightarrow$ Daten) zugreifen können, ist **wechselseitiger Ausschluss** (*Mutual Exclusion*) nötig

Kritische Abschnitte – Beispiel: Drucker-Spooler

Prozess X

```
next_free_slot = in; (16)
```

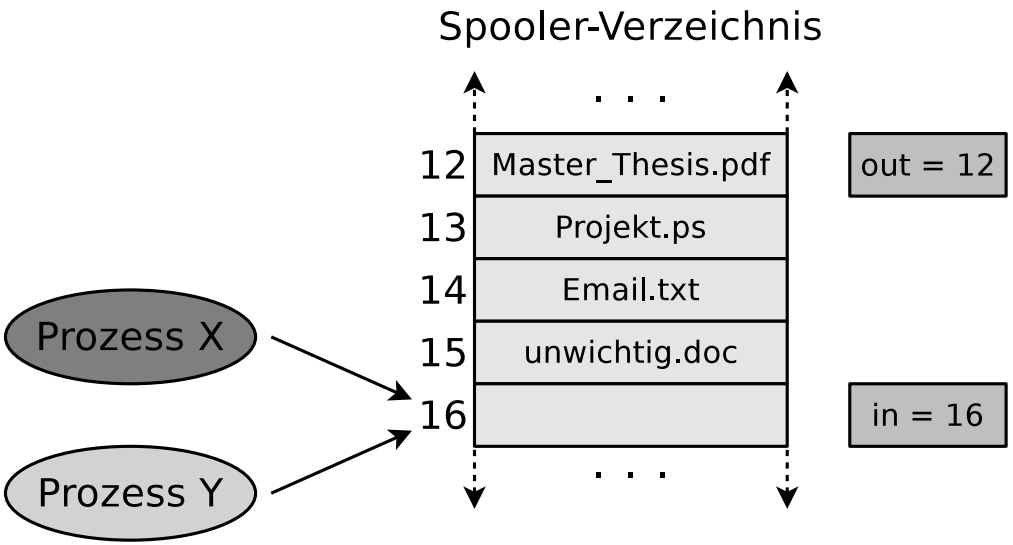
Prozesswechsel

Prozess Y

```
next_free_slot = in; (16)
Speichere Eintrag in next_free_slot; (16)
in = next_free_slot + 1; (17)
```

Prozesswechsel

```
Speichere Eintrag in next_free_slot; (16)
in = next_free_slot + 1; (17)
```



- Das Spooler-Verzeichnis ist konsistent
  - Aber der Eintrag von **Prozess Y** wurde von **Prozess X** überschrieben und ging verloren
- Eine solche Situation heißt **Race Condition**

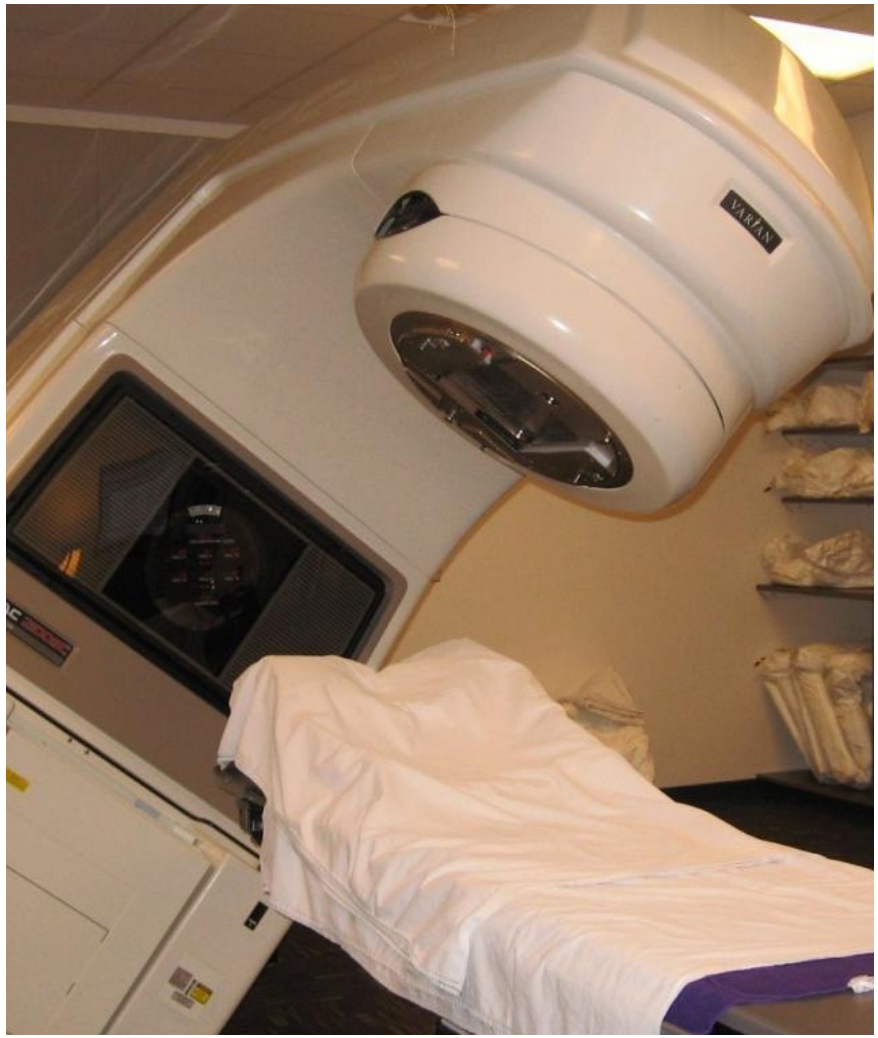
Race Condition (*Wettlaufsituation*)

- **Unbeabsichtigte Wettlaufsituation** zweier Prozesse, die den Wert der gleichen Speicherstelle ändern wollen
  - Das Ergebnis eines Prozesses hängt von der Reihenfolge oder dem zeitlichen Ablauf anderer Ereignisse ab
  - Häufiger Grund für schwer auffindbare Programmfehler
- Problem: Das Auftreten und die Symptome hängen von unterschiedlichen Ereignissen ab
  - Bei jedem Testdurchlauf können die Symptome unterschiedlich sein oder verschwinden
- Vermeidung ist u.a durch das Konzept der **Semaphore** ($\Longrightarrow$ Foliensatz 10) möglich

Therac-25: Race Condition mit tragischem Ausgang (1/2)

- Therac-25 ist ein Elektronen-Linearbeschleuniger zur Strahlentherapie von Krebstumoren
- Verursachte Mitte der 80er Jahre in den USA Unfälle durch mangelhafte Programmierung und Qualitätssicherung
  - Einige Patienten erhielten eine bis zu hundertfach erhöhte Strahlendosis

Bildquelle: Google Bildersuche



Therac-25: Race Condition mit tragischem Ausgang (2/2)

An Investigation of the Therac-25 Accidents. Nancy Leveson, Clark S. Turner
IEEE Computer, Vol. 26, No. 7, July 1993, S.18-41
`http://courses.cs.vt.edu/~cs3604/lib/Therac_25/Therac_1.html`

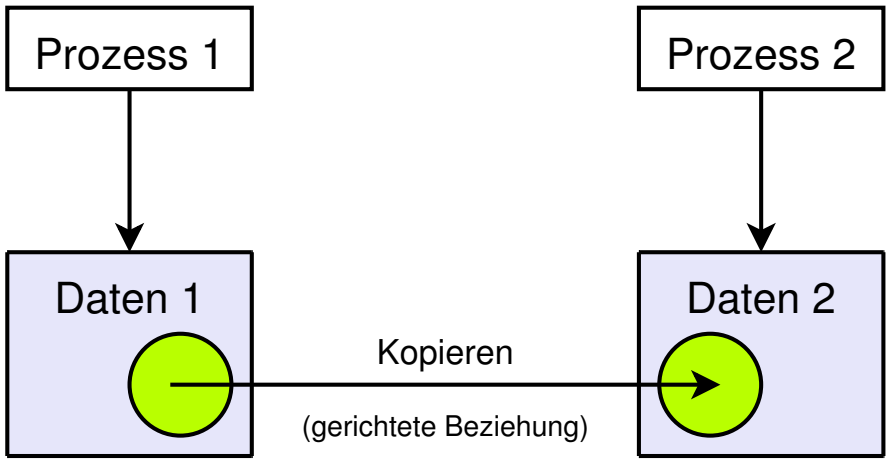
- 3 Patienten starben wegen Programmfehlern
- 2 Patienten starben durch eine Race Condition, die zu inkonsistenten Einstellungen des Gerätes und damit zu erhöhter Strahlendosis führte
  - Der Kontroll-Prozess synchronisierte nicht korrekt mit dem Prozess der Eingabeaufforderung
  - Der Fehler trat nur dann auf, wenn die Bedienung zu schnell erfolgte
  - Bei Tests trat der Fehler nicht auf, weil es Erfahrung (Routine) erforderte, um das Gerät so schnell zu bedienen

Bildquelle: `http://www.ircrisk.com/blognet/`

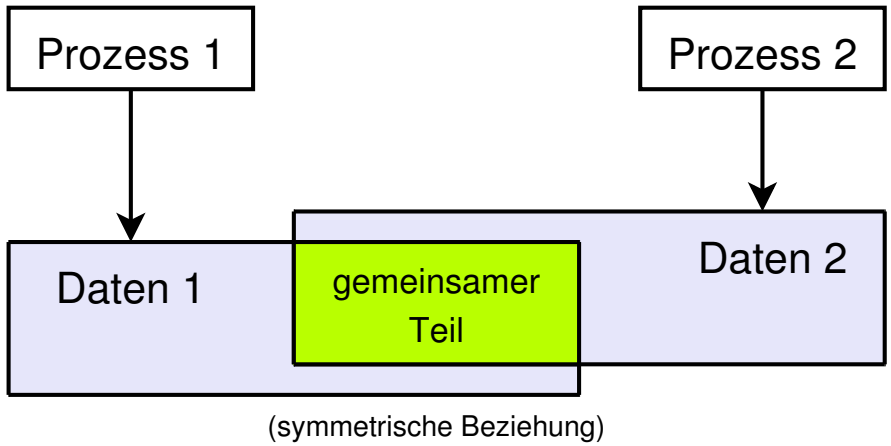
Die Prozessinteraktion besitzt 2 Aspekte...

1. Funktionaler Aspekt: **Kommunikation** und **Kooperation**

Kommunikation
(= expliziter Datentransport)



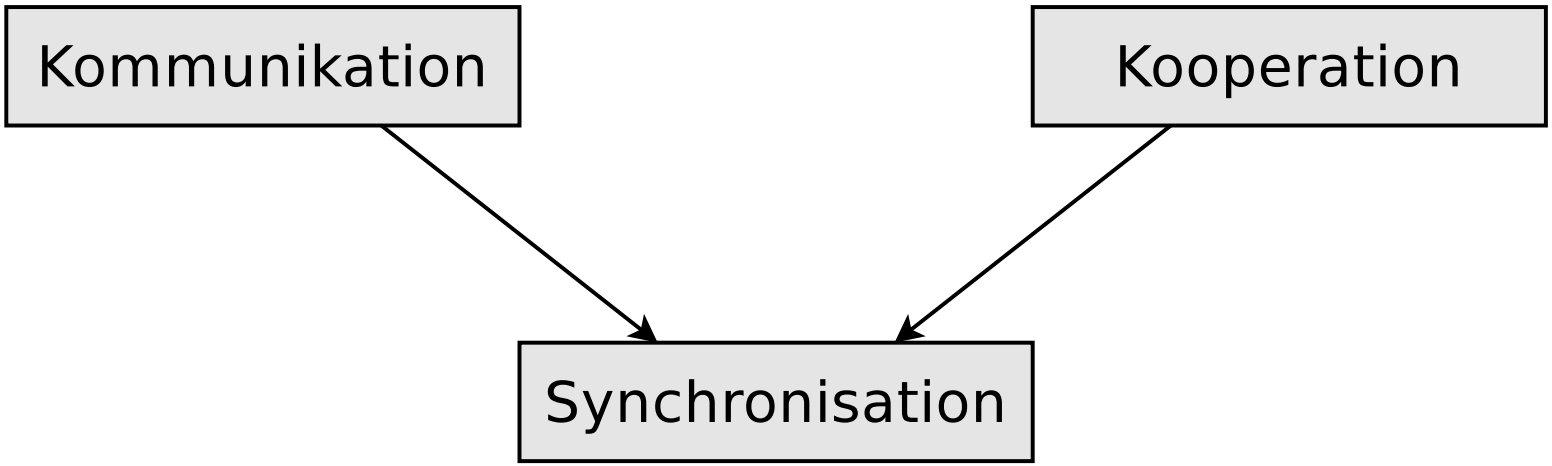
Kooperation
(= Zugriff auf gemeinsame Daten)



2. Zeitlicher Aspekt: **Synchronisation**

Interaktionsformen

- Kommunikation und Kooperation basieren auf Synchronisation
  - Synchronisation ist die elementarste Form der Interaktion
    - Grund: Kommunikation und Kooperation benötigen eine zeitliche Abstimmung zwischen den Intaraktionspartnern, um korrekte Ergebnisse zu erhalten
  - Darum behandeln wir zuerst die **Synchronisation**

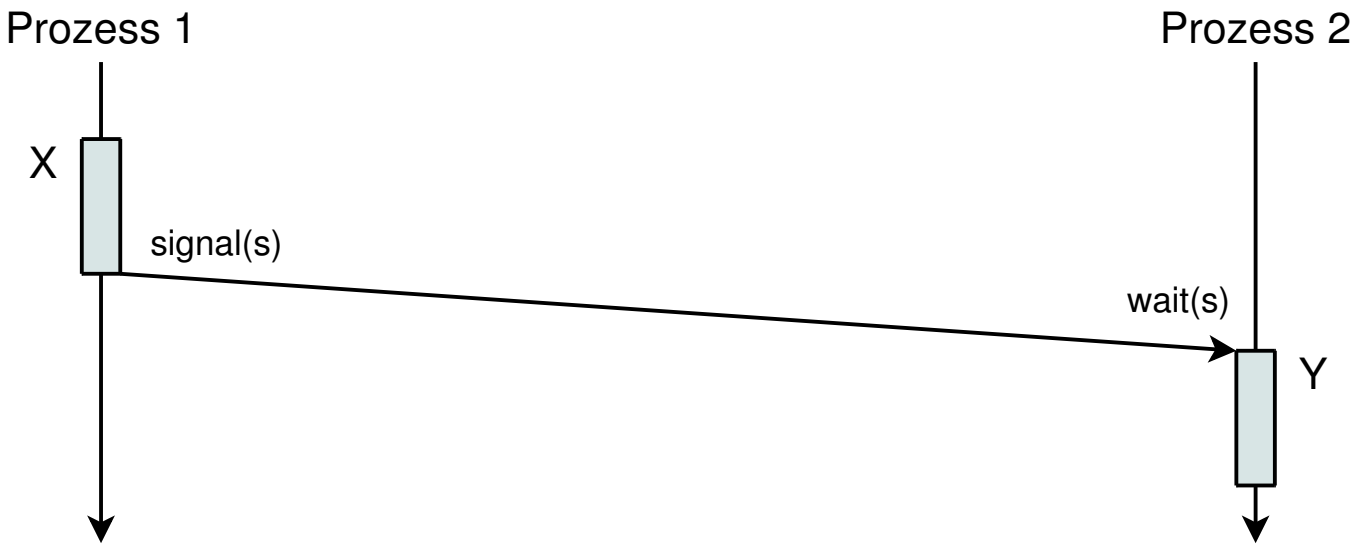


Synchronisation

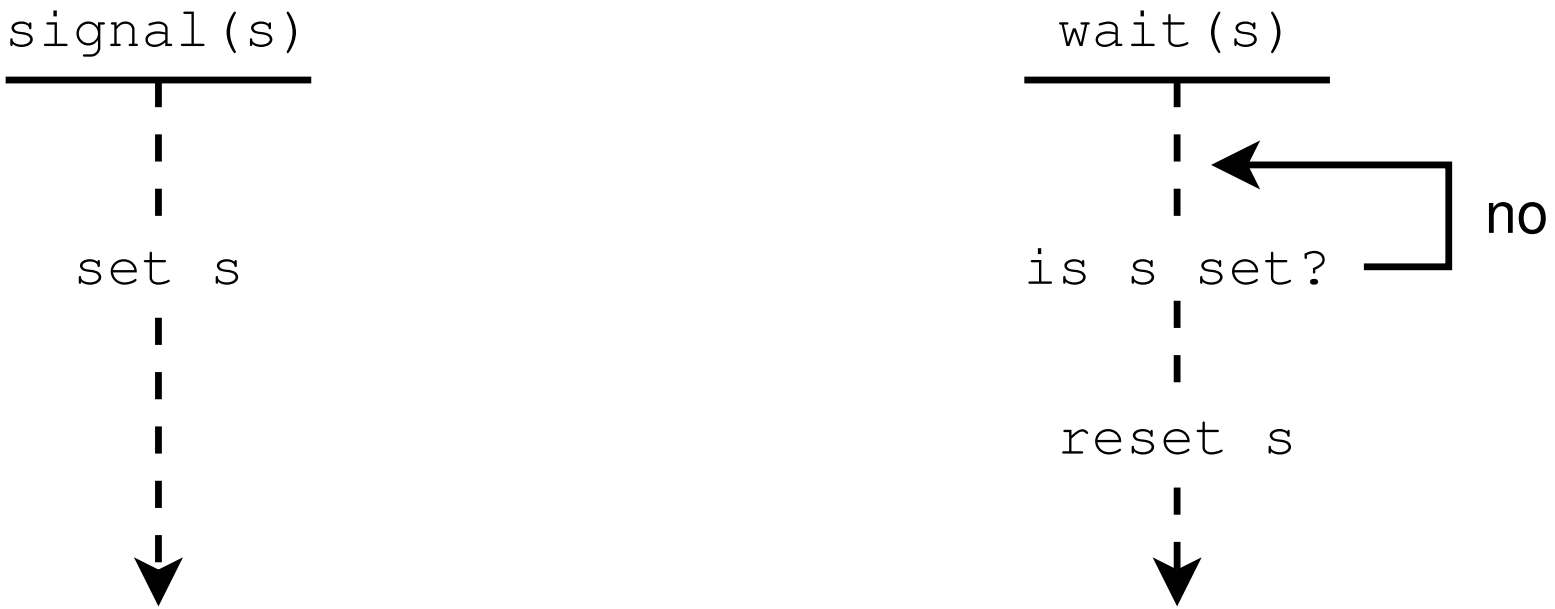
- Signalisieren
  - Aktives Warten (*Busy Waiting*)
  - Signalisieren und Warten
  - Rendezvous
  - Gruppensignalisierung
  - Gruppensynchronisierung mit Barrieren
- Blockieren
  - Sperren und Freigeben

Signalisierung

- Spezielle Art der Synchronisation
- Mit Signalisierung wird eine **Ausführungsreihenfolge** festgelegt
- Beispiel: Abschnitt **X** von Prozess 1 soll **vor** Abschnitt **Y** von Prozess 2 ausgeführt werden
  - Die Operation `signal` signalisiert, wenn Prozess 1 den Abschnitt **X** abgearbeitet hat
  - Prozess 2 muss eventuell auf das Signal von Prozess 1 warten



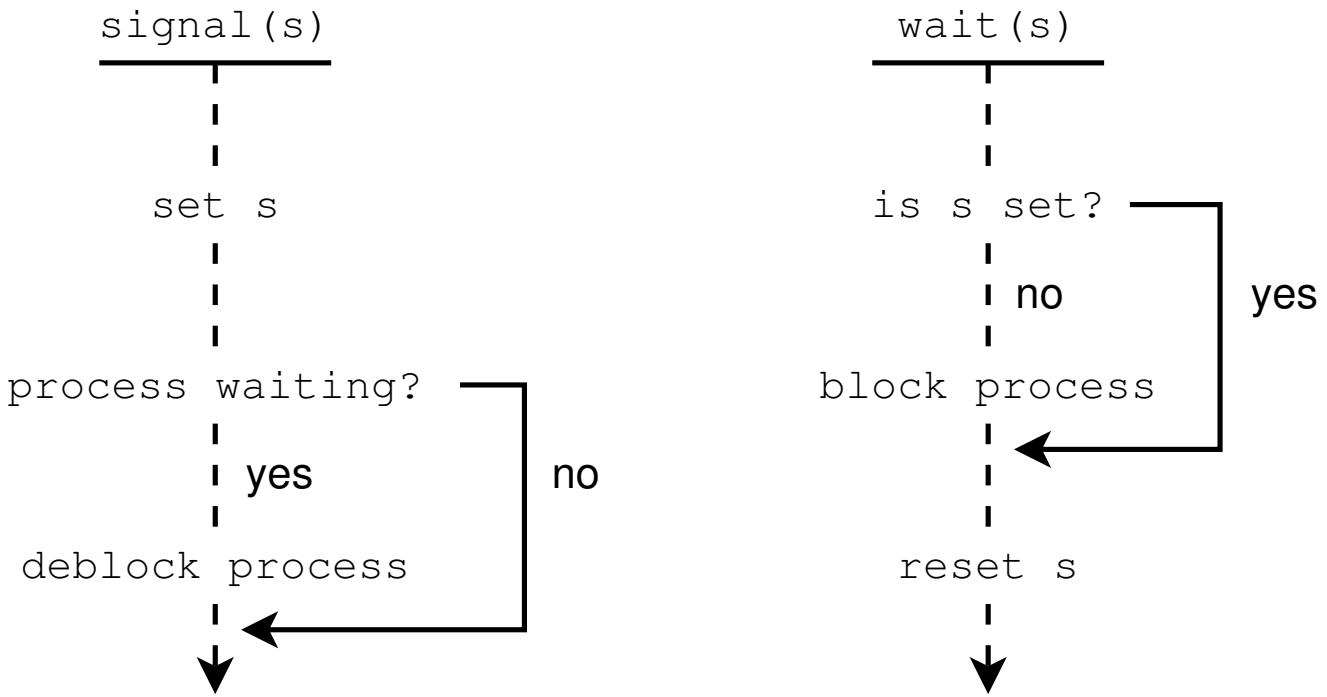
Einfachste Form der Signalisierung (aktives Warten)



- Vorgehensweise: Aktives Warten an der Signalvariable `s`
  - Rechenzeit der CPU wird verschwendet, weil diese immer wieder von dem Prozess belegt wird
- Diese Technik heißt auch **Warteschleife**

Signalisieren und Warten

- Besseres Konzept: Prozess 2 blockieren, bis Prozess 1 den Abschnitt **X** abgearbeitet hat
  - Vorteil: Die CPU wird entlastet
  - Nachteil: Es kann nur ein Prozess warten



Signalisieren und Warten unter JAVA

- Unter JAVA heißt die `signal()`-Operation `notify()`
  - `notify()` deblockiert einen wartenden Thread
- `wait()` blockiert die Ausführung eines Threads
- Das Schlüsselwort `synchronized` sorgt unter JAVA für gegenseitigen Ausschluss aller damit gekennzeichneten Methoden eines Objekts

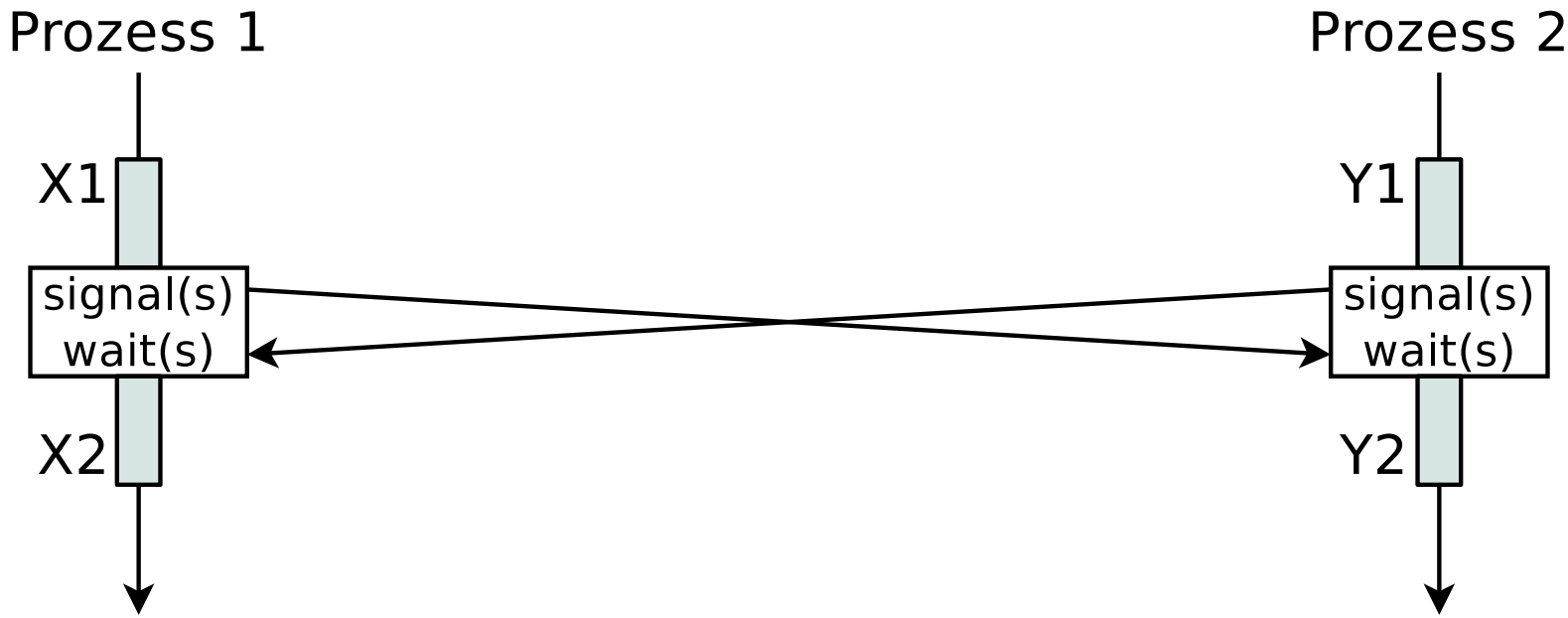
```
1  class Signal {
2      private boolean set = false;        // Signalvariable deklarieren
3
4      public synchronized void signal() {
5          set = true;                     // Signalvariable ändern
6          notify();                       // deblockiert den wartenden Thread (sofern vorhanden)
7      }
8
9      public synchronized void wait() {
10         if (!set) {                     // Wenn die Signalvariable nicht auf true gesetzt ist...
11             wait();                     // blockiert den Thread => wartet auf das Signal
12             set = false;                // Signalvariable ändern
13         }
14     }
15 }
```

Bei diesem JAVA-Beispiel und den folgenden JAVA-Beispielen fehlt u.a....

main-Methode, Objektinstanziierung mit Threads...
Mehr Informationen: `http://docs.oracle.com/javase/7/docs/api/?java/lang/Thread.html`

Rendezvous

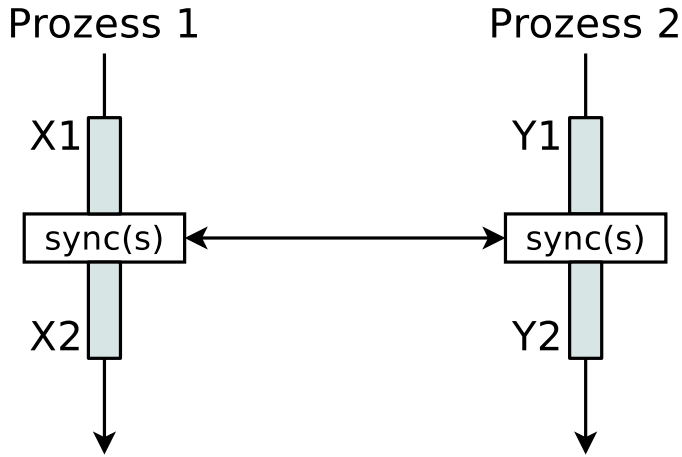
- Werden `wait()` und `signal()` symmetrisch ausgeführt, werden die Abschnitte `X1` und `Y1` vor den Abschnitten `X2` und `Y2` ausgeführt



- In einem solchen Fall **synchronisieren** die Prozesse 1 und 2

Rendezvous – sync()

- Implementierung einer Synchronisierung (Rendezvous) mit JAVA
- Die Methode `sync()` fasst die Operationen `wait()` und `signal()` zusammen



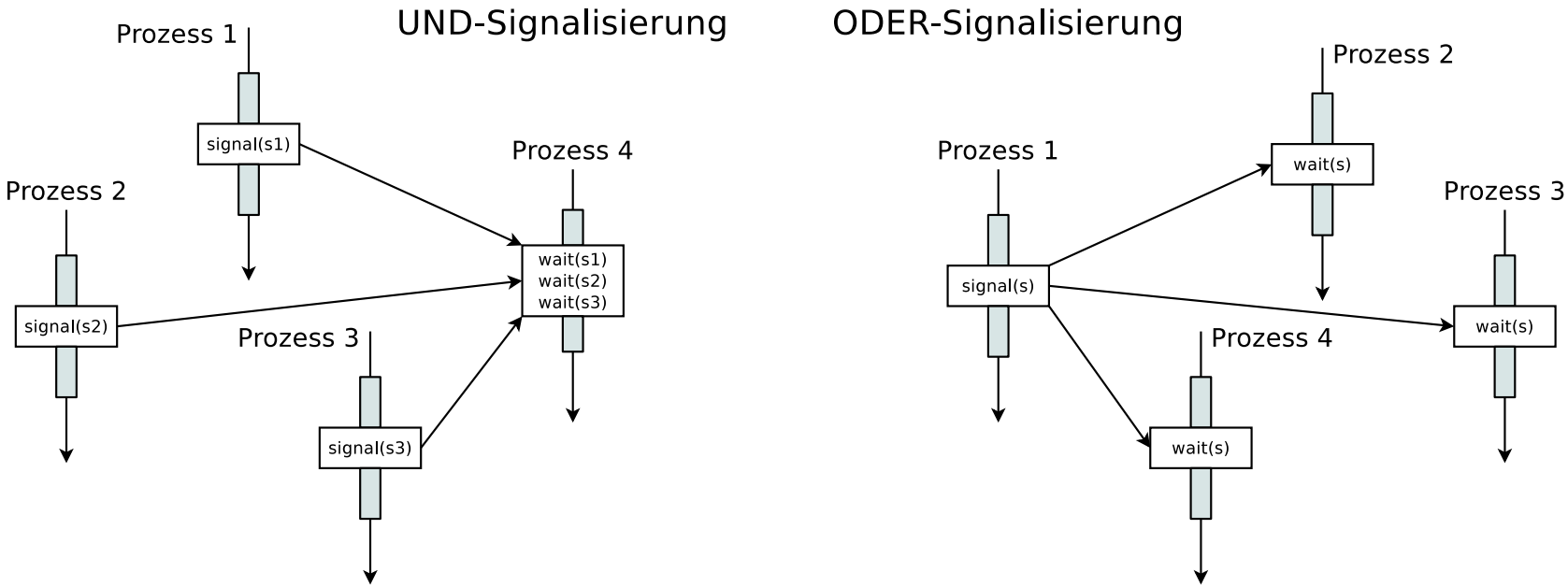

```
1  class Synchronisierung {
2      private boolean set = false;       // Signalvariable deklarieren
3
4      public synchronized void sync() {
5          if (set == false) {            // ich bin der erste Thread
6              set = true;                // Signalvariable ändern
7              wait();                    // blockiert den Thread => wartet auf das Signal
8          } else {                       // ich bin der zweite Thread
9              set = false;               // Signalvariable ändern
10             notify();                  // deblockiert den wartenden Thread (sofern vorhanden)
11         }
12     }
13 }
```

Hilfreiche Quellen zum Thema...

- Carsten Vogt, **Nebenläufige Programmierung – Ein Arbeitsbuch mit UNIX/LINUX und JAVA**, Hanser (2012) S.137-141
- David Flanagan, **JAVA in a Nutshell**, deutsche Übersetzung der 3. Auflage, O'Reilly Verlag (2000), S.166

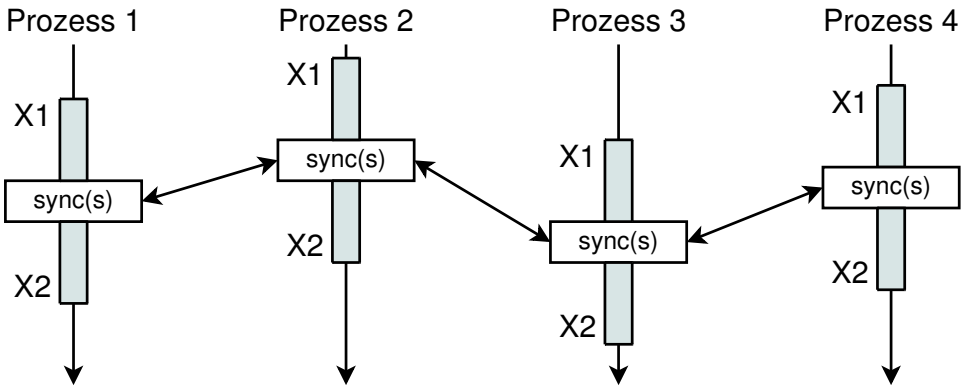
Gruppensignalisierung

- Signalisierung mit > 2 Prozessen
- Beispiele:
  - **UND-Signalisierung**:
    - Ein Prozess läuft erst weiter, wenn mehrere Prozesse ein Signal aufrufen
  - **ODER-Signalisierung**:
    - Mehrere Prozesse warten auf ein Signal
    - Erfolgt das Signal, wird einer der wartenden Prozesse deblockiert



Gruppenrendezvous mit Barriere

- Eine Barriere synchronisiert die beteiligten Prozesse an einer Stelle

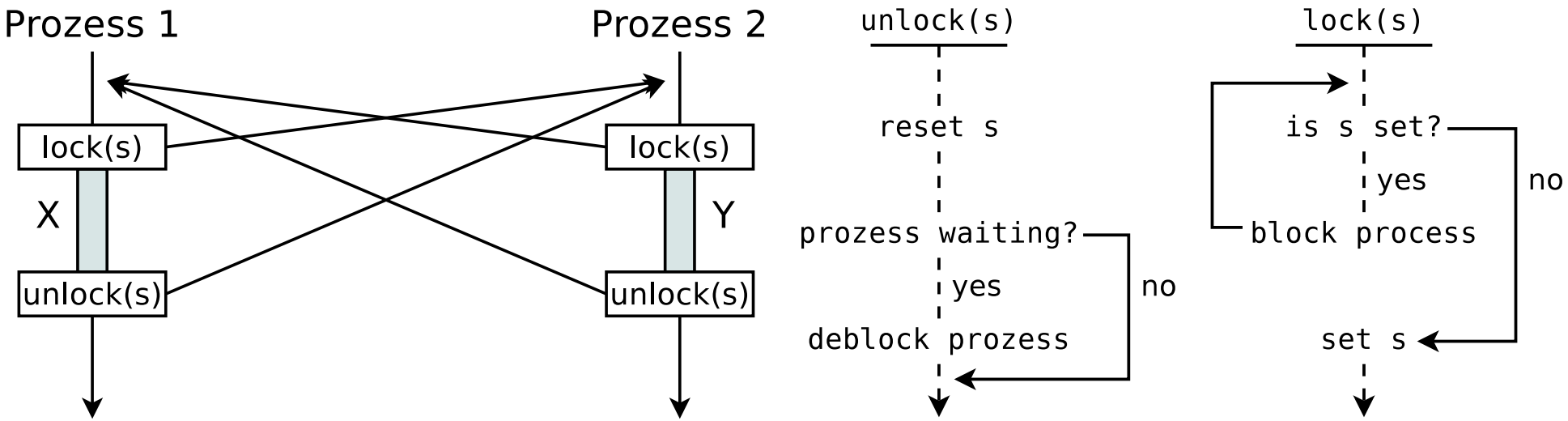


- Erst wenn alle Prozesse die Synchronisationsstelle erreicht haben, dürfen sie weiterlaufen

```
1  class BarrierenSynchronisation {
2      private int summe = p;              // Anzahl Prozesse
3      private int zaehler = 0;            // Anzahl wartende Prozesse
4
5      public synchronized void sync() {
6          zaehler = zaehler + 1;
7          if (zaehler < summe) {          // es fehlen noch Prozesse
8              wait();                     // auf die fehlenden Prozesse warten
9          } else {                        // es sind alle Prozesse eingetroffen
10             notifyAll();                // alle wartenden Prozesse deblockieren
11             zaehler = 0;
12         }
13     }
14 }
```

Blockieren (Sperren und Freigeben)

- Mit **Sperren** sichert man **kritische Abschnitte**



- Sperren garantieren, dass es bei der Abarbeitung von 2 kritischen Abschnitten keine Überlappung gibt
  - Beispiel: Kritische Abschnitte **X** von Prozess 1 und **Y** von Prozess 2

Unterschied zwischen Signalisieren und Blockieren

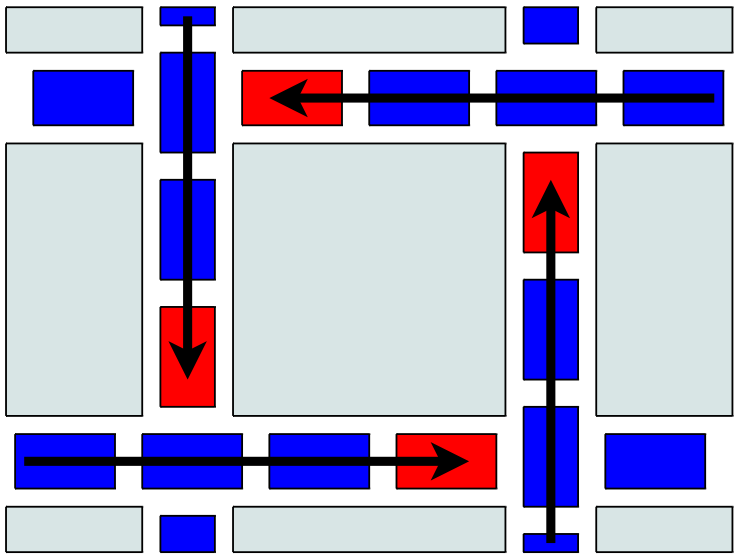
- **Signalisieren** legt die Ausführungsreihenfolge fest
  - Beispiel: Abschnitt A von Prozess 1 vor Abschnitt B von 2 ausführen
- **Blockieren** sichert kritische Abschnitte
  - Die Reihenfolge, in der die Prozesse ihre kritische Abschnitte abarbeiten, ist nicht festgelegt!
  - Es wird nur sichergestellt, dass es keine Überlappung in der Ausführung der kritischen Abschnitte gibt

```
1  class Sperre {
2      private boolean gesperrt = false;     // Signalvariable
3
4      public synchronized void sperre() {
5          while(gesperrt)
6              wait();                       // auf die fehlenden Prozesse warten
7              gesperrt = true;              // den Prozess sprerren
8      }
9
10     public synchronized void entsperren() {
11         gesperrt = false;                 // Sperre des Prozesses aufheben
12         notify();                         // alle wartenden Prozesse benachrichtigen
13     }
14 }
```

Probleme, die durch Blockieren entstehen

Bildquelle: Google Bildersuche

- **Verhungern** (Starvation)
  - Hebt ein Prozess eine Sperre nicht wieder auf, müssen die anderen Prozesse unendlich lange auf die Freigabe warten
- **Verklemmung** (Deadlock)
  - Es warten mehrere Prozesse gegenseitig auf die von ihnen gesperrten Ressourcen, sperren sie sich gegenseitig
  - Da alle am Deadlock beteiligten Prozesse (ewig) warten, kann keiner ein Ereignis auslösen, dass die Situation auflöst



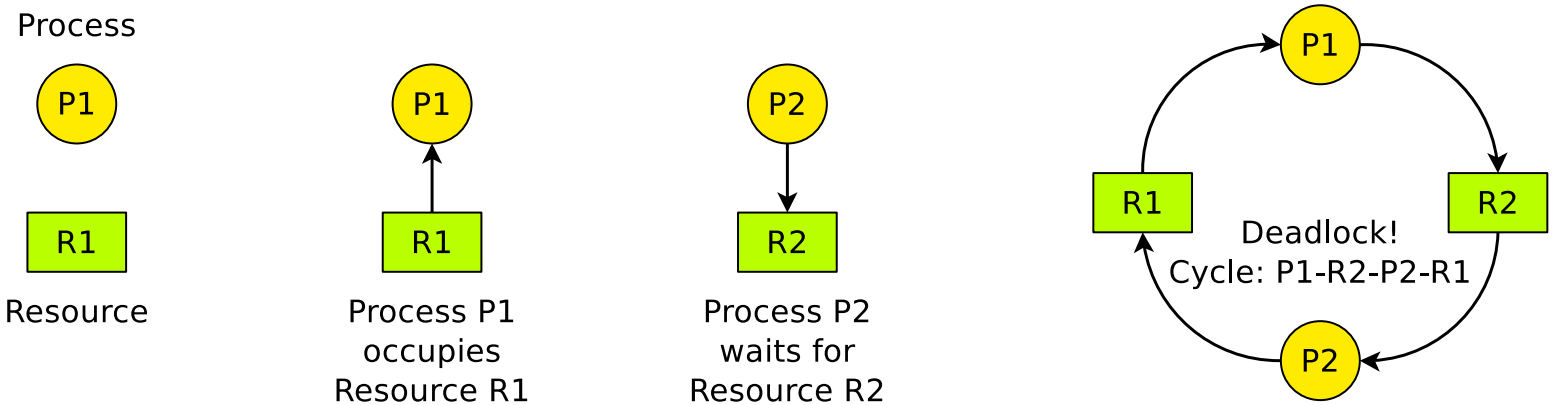
Bedingungen für Deadlocks

System Deadlocks. E. G. Coffman, M. J. Elphick, A. Shoshani. Computing Surveys, Vol. 3, No. 2, June 1971, S.67-78.
`http://people.cs.umass.edu/~mcorner/courses/691J/papers/TS/coffman_deadlocks/coffman_deadlocks.pdf`

- Damit ein Deadlock entstehen kann, müssen folgende Bedingungen gleichzeitig erfüllt sein:
  - **Wechselseitiger Ausschluss** (*mutual exclusion*)
    - Mindestens 1 Ressource wird von genau einem Prozess belegt oder ist verfügbar $\Longrightarrow$ nicht gemeinsam nutzbar (*non-sharable*)
  - **Anforderung weiterer Betriebsmittel** (*hold and wait*)
    - Ein Prozess, der bereits mindestens 1 Ressource belegt, fordert weitere Ressourcen an, die von einem anderen Prozess belegt sind
  - **Ununterbrechbarkeit** (*no preemption*)
    - Die Ressourcen, die ein Prozess besitzt, können nicht vom Betriebssystem entzogen, sondern nur durch ihn selbst freigegeben werden
  - **Zyklische Wartebedingung** (*circular wait*)
    - Es gibt eine zyklische Kette von Prozessen
    - Jeder Prozess fordert eine Ressource an, die der nächste Prozess in der Kette besitzt
- Fehlt eine Bedingung, ist ein Deadlock unmöglich

Betriebsmittel-Graphen

- Mit gerichteten Graphen können die Beziehungen von Prozessen und Ressourcen dargestellt werden
- Mit Betriebsmittel-Graphen kann man Deadlocks modellieren
  - Bei den Knoten eines Betriebsmittel-Graphen handelt es sich um:
    - **Prozesse**: Sind als Kreise dargestellt
    - **Ressourcen**: Sind als Rechtecke dargestellt
  - Eine Kante von einem Prozess zu einer Ressource bedeutet:
    - Der Prozess ist blockiert, weil er auf die Ressource wartet
  - Eine Kante von einer Ressource zu einem Prozess bedeutet:
    - Der Prozess belegt die Ressource



Beispiel zu Betriebsmittel-Graphen

- Es existieren 3 Prozesse:
  - P1, P2 und P3
- Jeder Prozess fordert 2 Ressourcen an und gibt diese dann wieder frei

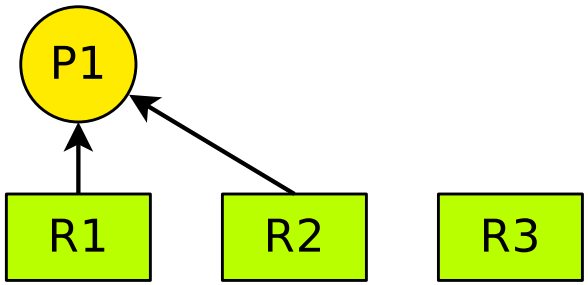
Prozess **P1**	Prozess **P2**	Prozess **P3**
Anforderung R1	Anforderung R2	Anforderung R3
Anforderung R2	Anforderung R3	Anforderung R1
Freigabe R1	Freigabe R2	Freigabe R3
Freigabe R2	Freigabe R3	Freigabe R1

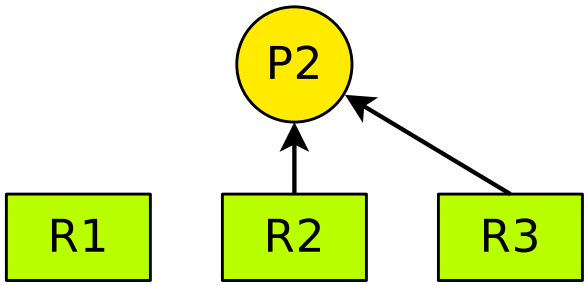
Beispiel aus Tanenbaum. Moderne Betriebssysteme. Pearson Studium. 2003

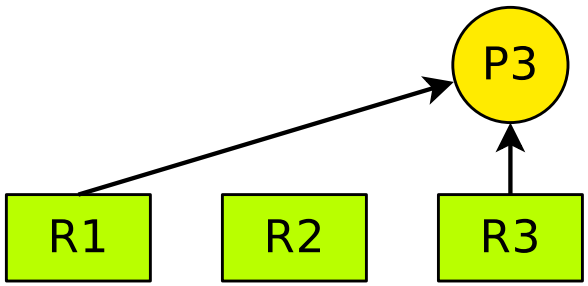
Quellen

Eine umfangreiche Beschreibung zu Betriebsmittel-Graphen enthält das Buch **Betriebssysteme – Eine Einführung**, *Uwe Baumgarten, Hans-Jürgen Siegert*, 6.Auflage, Oldenbourg Verlag (2007), Kapitel 6

Keine Nebenläufigkeit: P1 ⇒ P2 ⇒ P3





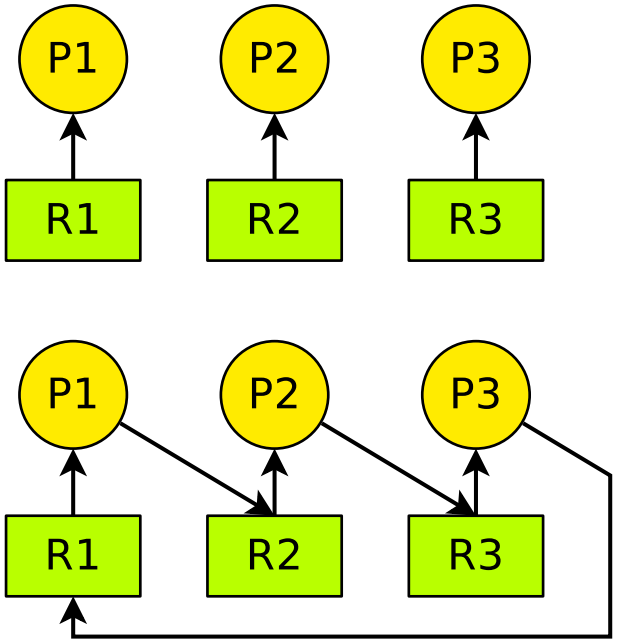


- P1 fordert R1 an
- P1 fordert R2 an
- P1 gibt R1 frei
- P1 gibt R2 frei

- P2 fordert R2 an
- P2 fordert R3 an
- P2 gibt R2 frei
- P2 gibt R3 frei

- P3 fordert R3 an
- P3 fordert R1 an
- P3 gibt R3 frei
- P3 gibt R1 frei
- **Kein Deadlock**

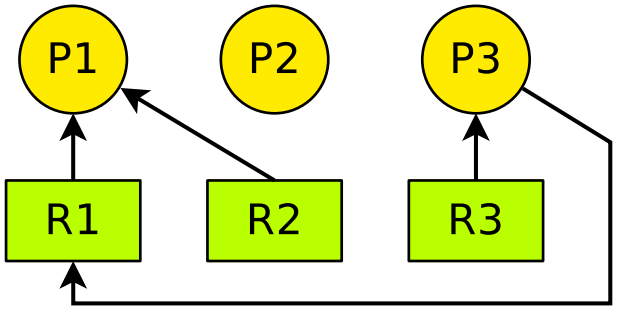
Nebenläufigkeit mit schlechter Reihenfolge



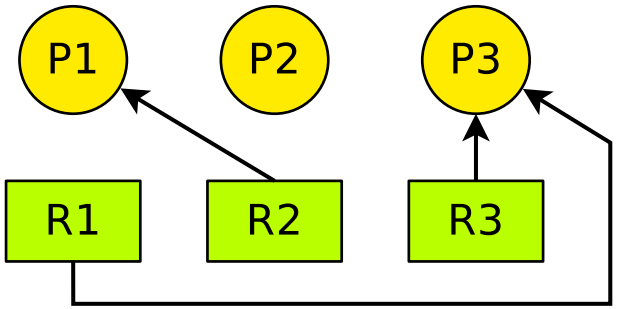
- P1 fordert R1 an
- P2 fordert R2 an
- P3 fordert R3 an

- P1 fordert R2 an
- P2 fordert R3 an
- P3 fordert R1 an
- **Deadlock** wegen Zyklus

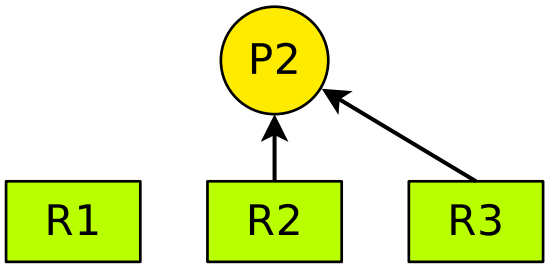
Nebenläufigkeit mit besserer Reihenfolge



- P1 fordert R1 an
- P3 fordert R3 an
- P1 fordert R2 an
- P3 fordert R1 an



- P1 gibt R1 frei
- P1 gibt R2 frei
- P3 gibt R1 frei
- P3 gibt R3 frei



- P2 fordert R2 an
- P2 fordert R3 an
- P2 gibt R2 frei
- P2 gibt R3 frei

Deadlock-Erkennung mit Matrizen

- Gibt es nur eine Ressource pro Ressourcenklasse (Scanner, CD-Brenner, Drucker, usw.), kann man Deadlocks mit Graphen darstellen/erkennen
- Existieren von einer Ressource mehrere Kopien, kann ein matrizenbasierter Algorithmus verwendet werden
- Wir definieren 2 Vektoren
  - **Ressourcenvektor** (*Existing Resource Vektor*)
    - Zeigt an, wie viele Ressourcen von jeder Klasse existieren
  - **Ressourcenrestvektor** (*Available Resource Vektor*)
    - Zeigt an, wie viele Ressourcen von jeder Klasse frei sind
- Zusätzlich sind 2 Matrizen nötig
  - **Belegungsmatrix** (*Current Allocation Matrix*)
    - Zeigt an, welche Ressourcen die Prozesse aktuell belegen
  - **Anforderungsmatrix** (*Request Matrix*)
    - Zeigt an, welche Ressourcen die Prozesse gerne hätten

Deadlock-Erkennung mit Matrizen – Beispiel (1/2)

Quelle des Beispiels: Tanenbaum. Moderne Betriebssysteme. Pearson. 2009

$$\text{Ressourcenvektor} = \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix}$$

- 4 Ressourcen von Klasse 1 existieren
- 2 Ressourcen von Klasse 2 existieren
- 3 Ressourcen von Klasse 3 existieren
- 1 Ressource von Klasse 4 existiert

$$\text{Belegungsmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

- Prozess 1 belegt 1 Ressource von Klasse 3
- Prozess 2 belegt 2 Ressourcen von Klasse 1 und 1 Ressource von Klasse 4
- Prozess 3 belegt 1 Ressource von Klasse 2 und 2 Ressourcen von Klasse 3

$$\text{Ressourcenrestvektor} = \begin{pmatrix} 2 & 1 & 0 & 0 \end{pmatrix}$$

- 2 Ressourcen von Klasse 1 sind frei
- 1 Ressource von Klasse 2 ist frei
- Keine Ressourcen von Klasse 3 sind frei
- Keine Ressourcen von Klasse 4 sind frei

$$\text{Anforderungsmatrix} = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

- Prozess 1 ist blockiert, weil keine Ressource von Klasse 4 frei ist
- Prozess 2 ist blockiert, weil keine Ressource von Klasse 3 frei ist
- **Prozess 3 ist nicht blockiert**

Deadlock-Erkennung mit Matrizen – Beispiel (2/2)

- Wurde Prozess 3 fertig ausgeführt, gibt er seine Ressourcen frei

$$\text{Ressourcenrestvektor} = \begin{pmatrix} 2 & 2 & 2 & 0 \end{pmatrix} \qquad \text{Anforderungsmatrix} = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ - & - & - & - \end{bmatrix}$$

- 2 Ressourcen von Klasse 1 sind frei
- 2 Ressourcen von Klasse 2 sind frei
- 2 Ressourcen von Klasse 3 sind frei
- Keine Ressourcen von Klasse 4 sind frei
- Prozess 1 kann nicht laufen, weil keine Ressource vom Typ 4 frei ist
- **Prozess 2 ist nicht blockiert**
- Wurde Prozess 2 fertig ausgeführt, gibt er seine Ressourcen frei

$$\text{Ressourcenrestvektor} = \begin{pmatrix} 4 & 2 & 2 & 1 \end{pmatrix} \qquad \text{Anforderungsmatrix} = \begin{bmatrix} 2 & 0 & 0 & 1 \\ - & - & - & - \\ - & - & - & - \end{bmatrix}$$

- **Prozess 1 ist nicht blockiert** $\Longrightarrow$ kein Deadlock in diesem Beispiel

Fazit zu Deadlocks

- Manchmal wird die Möglichkeit von Deadlocks akzeptiert
  - Entscheidend ist, wie wichtig ein System ist
    - Ein Deadlock, der statistisch alle 5 Jahre auftritt, ist kein Problem in einem System das wegen Hardwareausfällen oder sonstigen Softwareproblemen jede Woche ein mal abstürzt
- Deadlock-Erkennung ist aufwendig und verursacht Overhead
- In allen Betriebssystemen sind Deadlocks möglich:
  - Prozesstabelle voll
    - Es können keine neuen Prozesse erzeugt werden
  - Maximale Anzahl von Inodes vergeben
    - Es können keine neuen Dateien und Verzeichnisse angelegt werden
- Die Wahrscheinlichkeit, dass so etwas passiert, ist gering, aber $\neq 0$
  - Solche potentiellen Deadlocks werden akzeptiert, weil ein gelegentlicher Deadlock nicht so lästig ist, wie die ansonsten nötigen Einschränkungen (z.B. nur 1 laufender Prozess, nur 1 offene Datei, mehr Overhead)