

3. Prozesse und Prozessverwaltung

[3.1. Parallele und nebenläufige Prozesse](#)

[3.2. Prozesse aus Sicht des Betriebssystems](#)

[3.3. Erzeugung von Threads in Java](#)

[3.4. Inter-Prozess-Kommunikation und Synchronisation](#)

[3.5. Deadlocks](#)

[3.6. Scheduling-Strategien](#)

[3.7. Besonderheiten bei Echtzeitbetrieb](#)

[3.8. Zusammenfassung \(Kapitel 3\)](#)

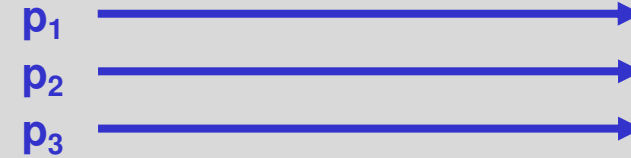
3.1. Parallele und nebenläufige Prozesse

Ein **Prozess** ist

- ein **in Ausführung befindliches Programm**
(= ausführbares Programm + Daten + Stack + Programmzähler + ...)
- die **kleinste Einheit**, der **Betriebsmittel** (Speicherplatz, Prozessorzeit, ...) **zugeordnet** werden.

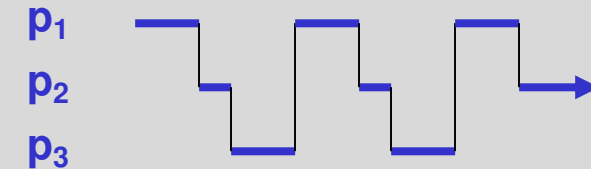
Parallele Prozesse sind

- mehrere Prozesse, die **gleichzeitig auf mehreren Prozessoren** ausgeführt werden.



Verzahnte Prozesse sind

- mehrere Prozesse, die **stückweise abwechselnd auf einem Prozessor** ausgeführt werden. Die Umschaltung erfolgt entweder durch die Prozessverwaltung oder durch die Prozesse selbst.



Mit verzahnter Ausführung kann parallele Ausführung simuliert werden. Häufige Wechsel vermitteln den Eindruck, dass alle Prozesse gleichmäßig fortschreiten.

Nebenläufige Prozesse sind Prozesse, die **parallel oder verzahnt** ausgeführt werden.

Threads (Fäden der Programmausführung)

- sind „**leichte**“ **Prozesse**, die parallel oder verzahnt **in gemeinsamem Speicher** ablaufen. Die **Prozessumschaltung** ist hier **besonders einfach und schnell**.

Prozess = Task ?

Das „IEEE Standard Computer Dictionary“ definiert die häufig synonym verwendeten Begriffe „Prozess“ und „Task“ wie folgt:

process

- (1) A sequence of steps performed for a given purpose; for example, the software development process.
- (2) **An executable unit managed by an operating system scheduler.**
- (3) To perform operations on data.

task

- (1) **A sequence of instructions treated as a basic unit of work by the supervisory program of an operating system.**
- (2) In software design, a software component that can operate in parallel with other software components.

Der Begriff „Prozess“ hat für den Bereich der Betriebssysteme fundamentale Bedeutung. Er wurde sogar zur Definition des Begriffs „Betriebssystem“ verwendet:

**„Das Betriebssystem ist die System-Software,
welche die Hardware bei der Prozessverwaltung unterstützt“.**

(P.J. Denning, 1983, in Encyclopedia of Computer Science & Engineering)

Programme sind statisch, Prozesse sind dynamisch

Ein ausführbares Programm

- liegt (hoffentlich gut geschützt) als **Datei** vor (statisch),
- hat die gleichen **Attribute wie andere Dateien**, d.h. wie „Daten“ (Sichtbarkeit, Zugriffsrecht, ...),
- geht bei Vernichtung verloren, **löscht sich aber nicht selbst**.

Ein Prozess

- **entsteht durch Laden eines Programms** (genauer: einer Kopie) in den Speicher und Kreieren des Prozesses (= „Start“ des Programms),
- existiert nach dem Kreieren **unabhängig von der Existenz des zugrundeliegenden Programms**,
- hat spezielle Attribute (Status, Priorität, Zugriffsrechte, ...),
- entsteht ggf. als **Konkurrent anderer Prozesse** aus dem Programm (z.B. mehrfacher Aufruf eines Editors),
- kann ggf. **andere Prozesse kreieren**,
- **beendet sich i.d.R. selbst** (ohne Einfluss auf das Programm!).

„Multiprocessing“, „Multitasking“, „Multiprogramming“

Die nachfolgenden Begriffe werden häufig zur Klassifikation unterschiedlicher Arten der parallelen bzw. nebenläufigen Bearbeitung verwendet:

Multiprocessing:

Mehrere (Hardware-) Prozessoren bearbeiten unterschiedliche Prozesse.

Multitasking:

- **(Quasi-) Gleichzeitige Ausführung mehrerer Programme, ggf. mit nur einem Prozessor.**
- Multitasking-Systeme überlassen die **Verwaltung des task-spezifischen Speichers** häufig vollständig dem **Anwender**.
- Verschiedene Tasks arbeiten häufig auf demselben Speicherbereich, wobei die ggf. erforderliche **Koordination dem Anwender überlassen** wird.

Multiprogramming:

... **ergänzt Multitasking um eine automatisch Speicherverwaltung**, welche die Programmbearbeitung in getrennten Speicherbereichen sicherstellt.

Anwendungen paralleler und nebenläufiger Prozesse

Viele Anwendungen heutiger Computer basieren wesentlich auf dem Einsatz paralleler bzw. nebenläufiger Prozesse. Hier einige Beispiele für Anwendungen, die ohne dieses Konzept nicht realisierbar wären:

Benutzungsschnittstellen:

Aufwändige Berechnungen werden häufig nebenläufig programmiert, damit die Bedienung der Oberfläche nicht blockiert wird.

Simulation realer Abläufe:

z.B. Produktion in einer Fabrik.

Animation:

Veranschaulichung von Abläufen bzw. Algorithmen. Wichtiges Einsatzgebiet: Spiele.

Steuerung von Geräten:

Prozesse im Computer überwachen und steuern externe Geräte, z.B. Montage-Roboter.

Parallelrechner:

Mehrere Prozesse bearbeiten gemeinsam die gleiche Aufgabe.

Beispiel 1: Erzeuger-/Verbraucher-System

Erzeuger:

Prozess, der „**Produkte**“ **generiert** und diese **in einem Puffer** (fester ?) Länge **ablegt**.

Verbraucher:

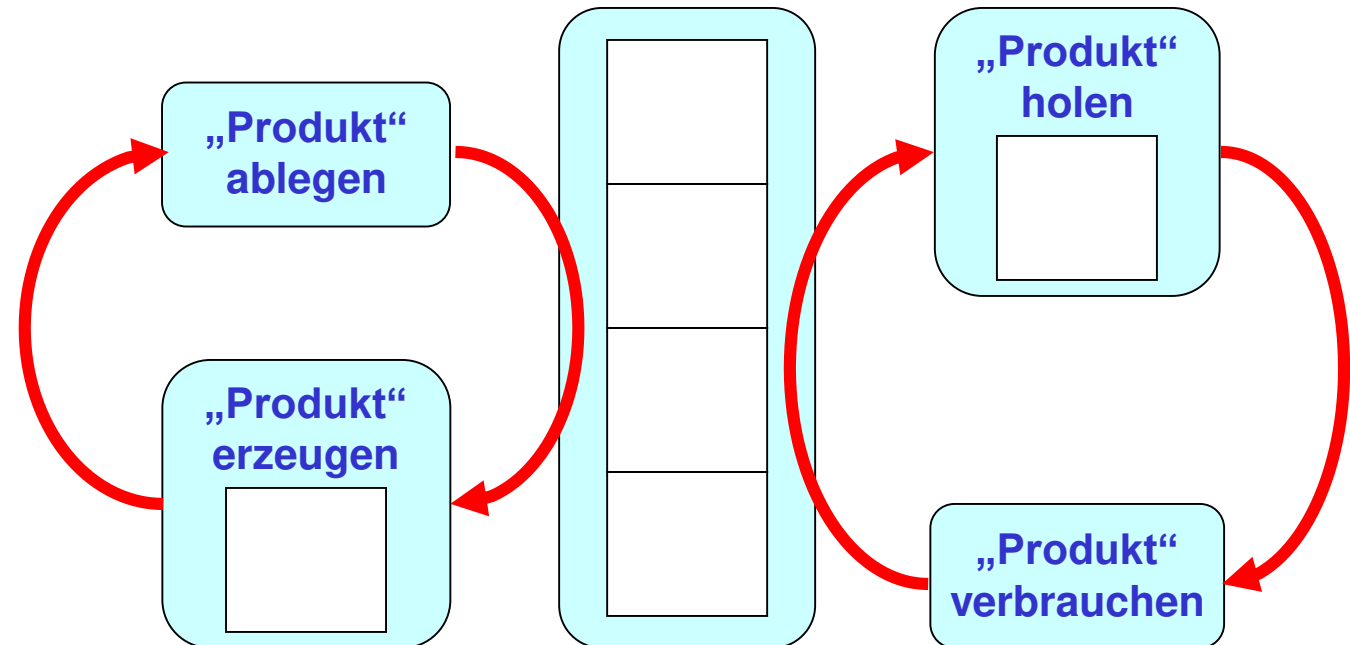
Prozess, der die vom Erzeuger bereitgestellten „**Produkte**“ **aus dem Puffer holt und verbraucht**.

Wartezeiten:

- Bei **gefülltem Lager** muss der Erzeuger warten, bis der **Verbraucher Platz schafft**.
- Bei **leerem Lager** muss der Verbraucher warten, bis der **Erzeuger die Produktion beendet** hat („Lieferzeit“).

Anmerkungen:

- Zusätzlich kann **Erhaltung der Reihenfolge** verlangt werden.
- Für das hier skizzierte System gibt es beim Computer-Einsatz **viel-fältige Anwendungen**.



Beispiel 2: „Dining Philosophers“

Erstmals formuliert von Dijkstra 1968

Die Ausgangssituation:

- **Fünf Philosophen** sitzen um einen runden Tisch und **wollen Spaghetti essen**.
- Diese sind so glitschig, dass ein Verzehr **nur mit 2 Gabeln möglich** ist.
- **Zwischen je 2 Philosophen liegt eine Gabel.**



Das Leben der Philosophen besteht aus:

- a) **Denken** und
- b) **Essen**.

Sobald ein Philosoph hungrig wird:

- **nimmt** er zunächst **eine der beiden Gabeln** neben seinem Teller,
- **nimmt dann** - sofern verfügbar - **die andere Gabel**,
- **isst** eine gewisse Zeit lang und
- **legt** dann **beide Gabeln** wieder **auf den Tisch**.

Aufg.: Man schreibe ein **Simulationsprogramm** für die Aktivitäten der Philosophen.

Beispiel 2: Dining Philosophers (2)

Eine erste Idee führt zu folgender Schleife:

```
while (true) {  
    denken ();  
    nimmGabel (i);           // nimm rechte Gabel, falls verfügbar. Sonst: Warte, bis sie verfügbar wird.  
    nimmGabel ((i+1)%n);     // nimm linke Gabel, falls verfügbar. Sonst: Warte, bis sie verfügbar wird.  
    essen ();  
    legeGabelHin (i);        // lege die rechte Gabel wieder hin  
    legeGabelHin ((i+1)%n); // lege die linke Gabel wieder hin  
}
```

Nehmen alle Philosophen gleichzeitig die jeweils rechte Gabel, dann entsteht eine **Verklemmung (Deadlock)**:

- jeder hält die jeweils rechte Gabel,
- keiner gibt das wieder her, was er schon hat,
- **niemand kann etwas essen.**



Helfen Sie den Philosophen:

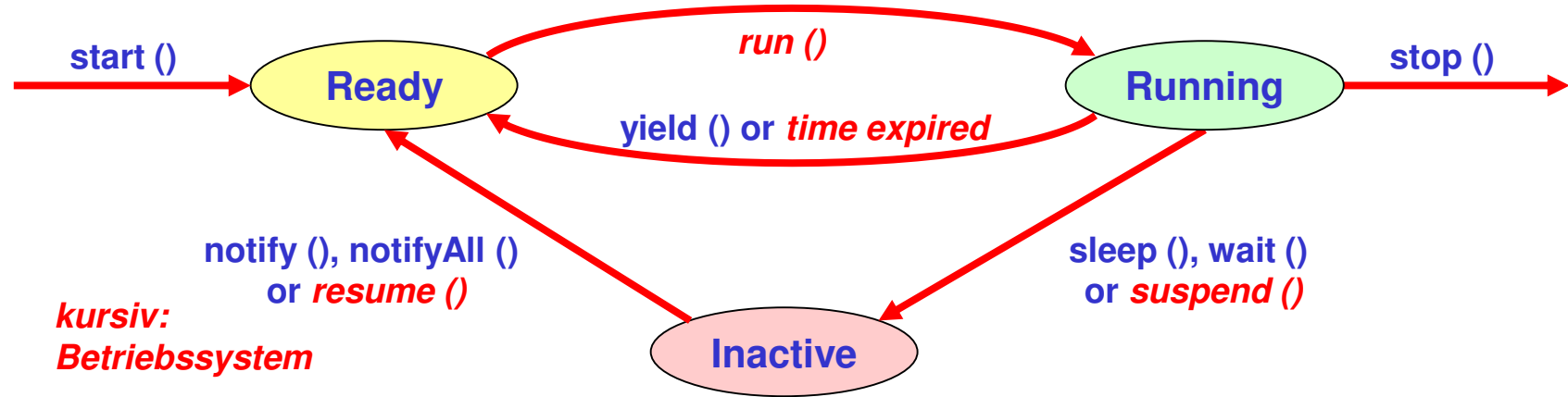
Schreiben Sie ein **Java-Programm**, bei dem

- **jeder Philosoph durch einen Thread repräsentiert** wird,
- **kein Deadlock** eintreten kann.

3.2. Prozesse aus Sicht des Betriebssystems

Zu jedem Prozess verwaltet das Betriebssystem umfassende Informationen:

a) Zustand des Prozesses



b) Programm-Zähler (Angabe des nächsten Befehls)

c) CPU Register (Akkumulator, Index-Register, Stack Pointer, ...)

d) Scheduling Information (Priorität, Position in der Schlange, ...)

e) Speicherverwaltung (Aktuelle Speicherzuordnung)

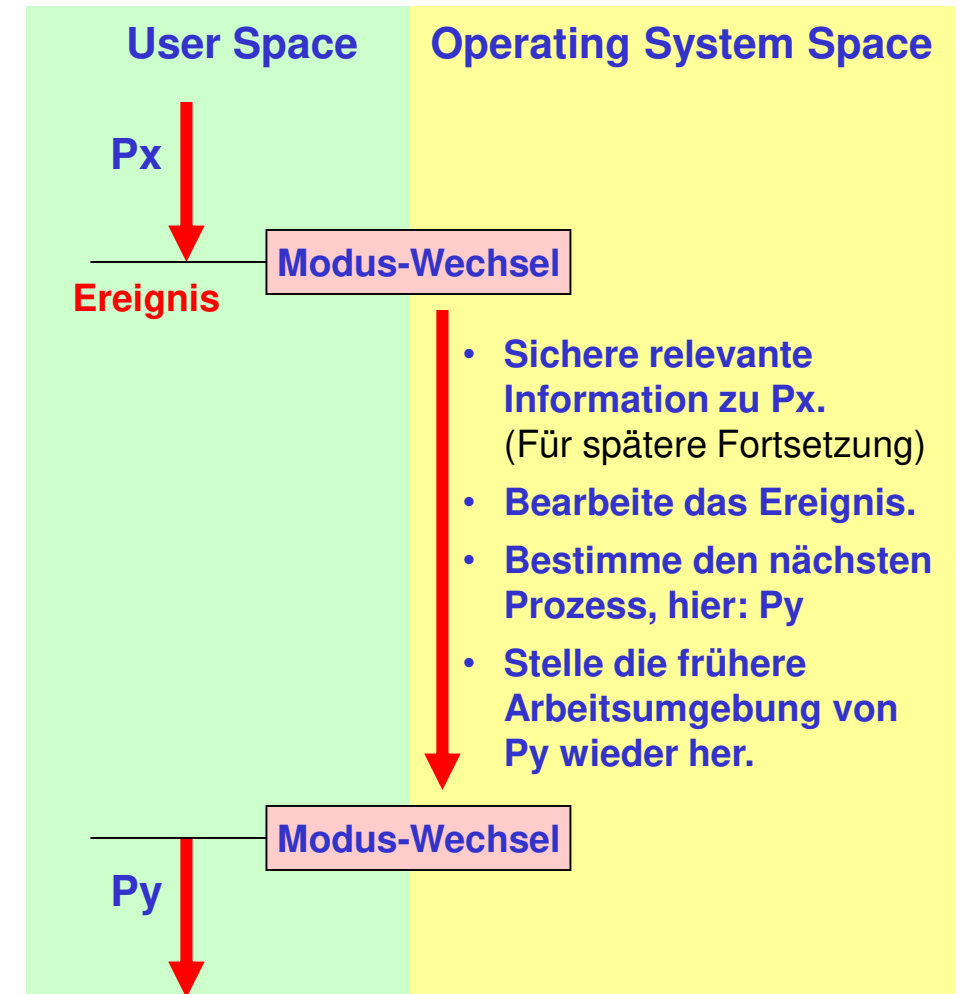
f) I/O-Status (Reservierte Geräte, angeforderte Geräte, offene Files, ...)

Prozess-Wechsel

Bei Prozess-Wechseln müssen alle **relevanten Status-Informationen gesichert** werden, damit die Bearbeitung später korrekt fortgesetzt werden kann.

Anmerkungen:

- **Prozess-Wechsel** erfolgen i.A. **ereignis-gesteuert**: Das Betriebssystem wird über ein Ereignis informiert und bearbeitet dieses Ereignis in eigener Verantwortung.
- Aus Sicherheitsgründen arbeitet das **Betriebssystem** häufig nicht nur in eigenen Adress-Bereichen, sondern auch **in einem speziellen Modus**.
- Bei **Wechsel zwischen Anwender-Prozessen** ist zunächst der **Wechsel auf einen System-Prozess** erforderlich.
- Prozess-Wechsel können durch **Spezial-Hardware** deutlich beschleunigt werden.

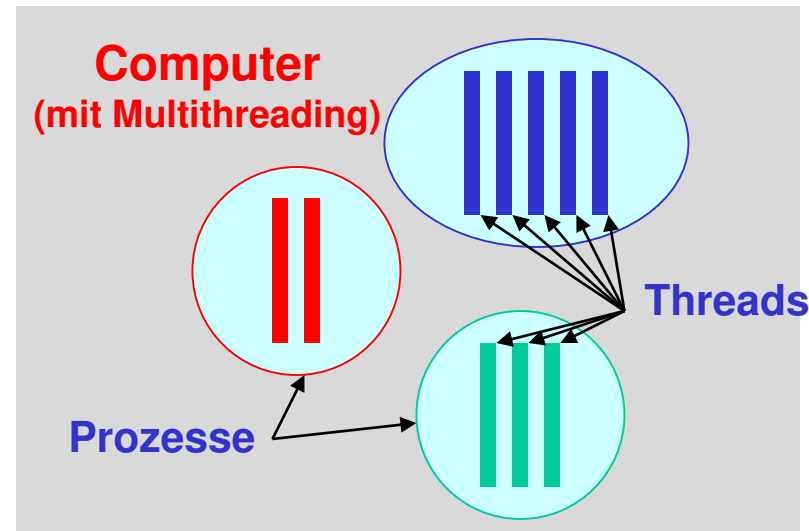
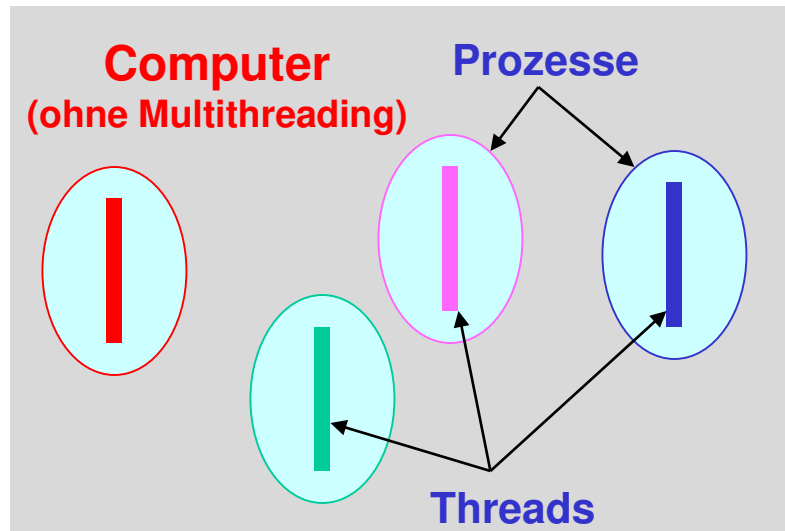


Threads: „Lightweight processes“

Prozess-Wechsel können beschleunigt werden, wenn die Prozesse **gemeinsame Betriebsmittel** nutzen, z.B.

- **gemeinsamen Speicherplatz**
- **gemeinsamen Programm-Code**
- **gemeinsame Dateien.**

Moderne Betriebssysteme unterstützen schnelle Wechsel des Kontrollflusses durch das Konzept der Threads: **Kontrollflusswechsel ohne Prozesswechsel.**



Threads stellen einen **Kompromiss** dar **zwischen**

- **umfassenden Schutzmechanismen** bei „schweren“ (klassischen) Prozessen **und**
- **Verzicht auf Schutzmechanismen** (z.B. bei vielen Echtzeit-Betriebssystemen).

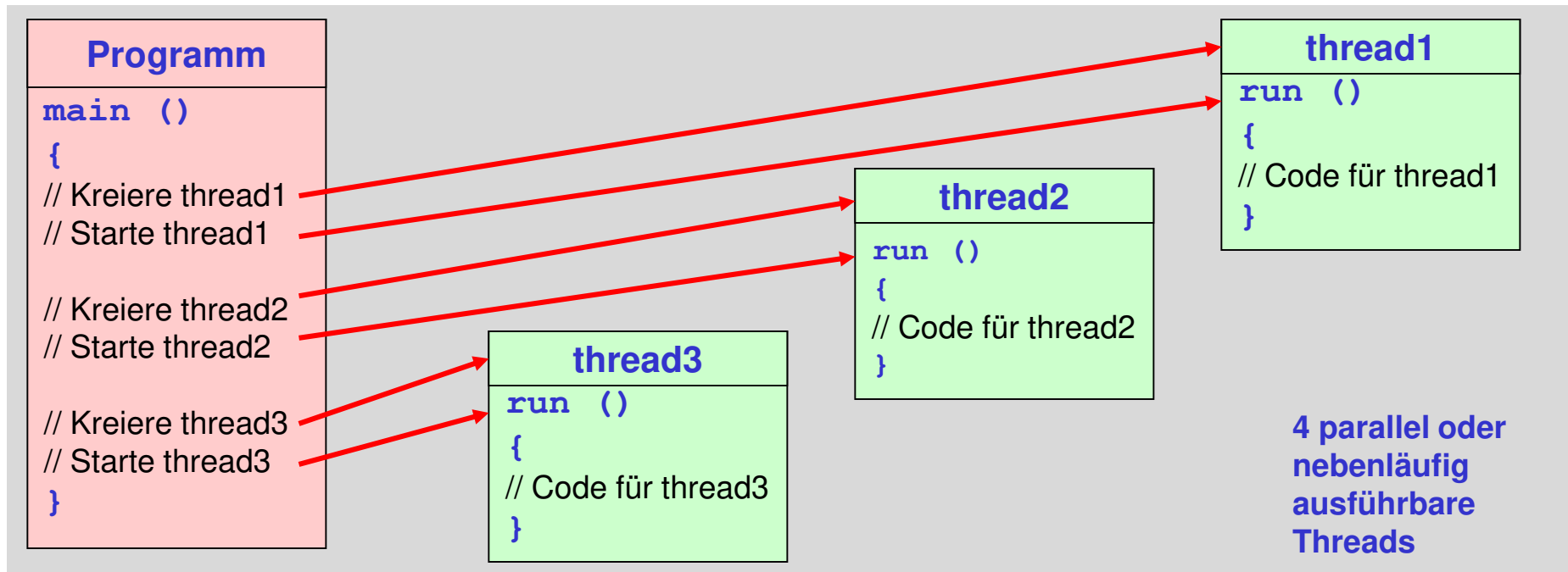
3.3. Wiederholung: Erzeugung von Threads in Java

Ein laufendes Java-Programm hat **immer mindestens einen Thread**:

- **Programm**: Der Kontrollfluss beginnt mit dem **Anfang von `main()`**
- **Applet**: Der Kontrollfluss beginnt mit dem **Browser**.

Weitere Threads können* **als Objekte** der Klasse `java.lang.Thread` erzeugt werden. Die Codeausführung beginnt dort stets mit der Methode `run ()`.

Die Methode `run ()` ist **public**; sie akzeptiert keine Parameter und liefert keine Werte.



* Alternativ: Implementierung für das Interface „Runnable“:

```
public class MeineRunnableClass extends IrgendeineKlasse implements Runnable { ... }
```

Wiederholung: Beispiel „MeineThreads“

Hier ein Beispiel, bei dem **zwei Threads der neu definierten Klasse MeineThreads** (Unterklasse von Thread) **erzeugt und gestartet** werden.

```
public class MeineThreads extends Thread {
    int i;

    public static void main (String args []) {
        MeineThreads meinErsterThread = new MeineThreads ();
        meinErsterThread.start ();
        MeineThreads meinZweiterThread = new MeineThreads ();
        meinZweiterThread.start ();
    }

    public void run () {
        while (true) {
            System.out.println (Thread.currentThread().getName());
            i = 0;
            while (i<1000000000)
                i += 1;
        }
    }
}
```

„meinErsterThread“
als neuen Thread
deklarieren und
erzeugen.

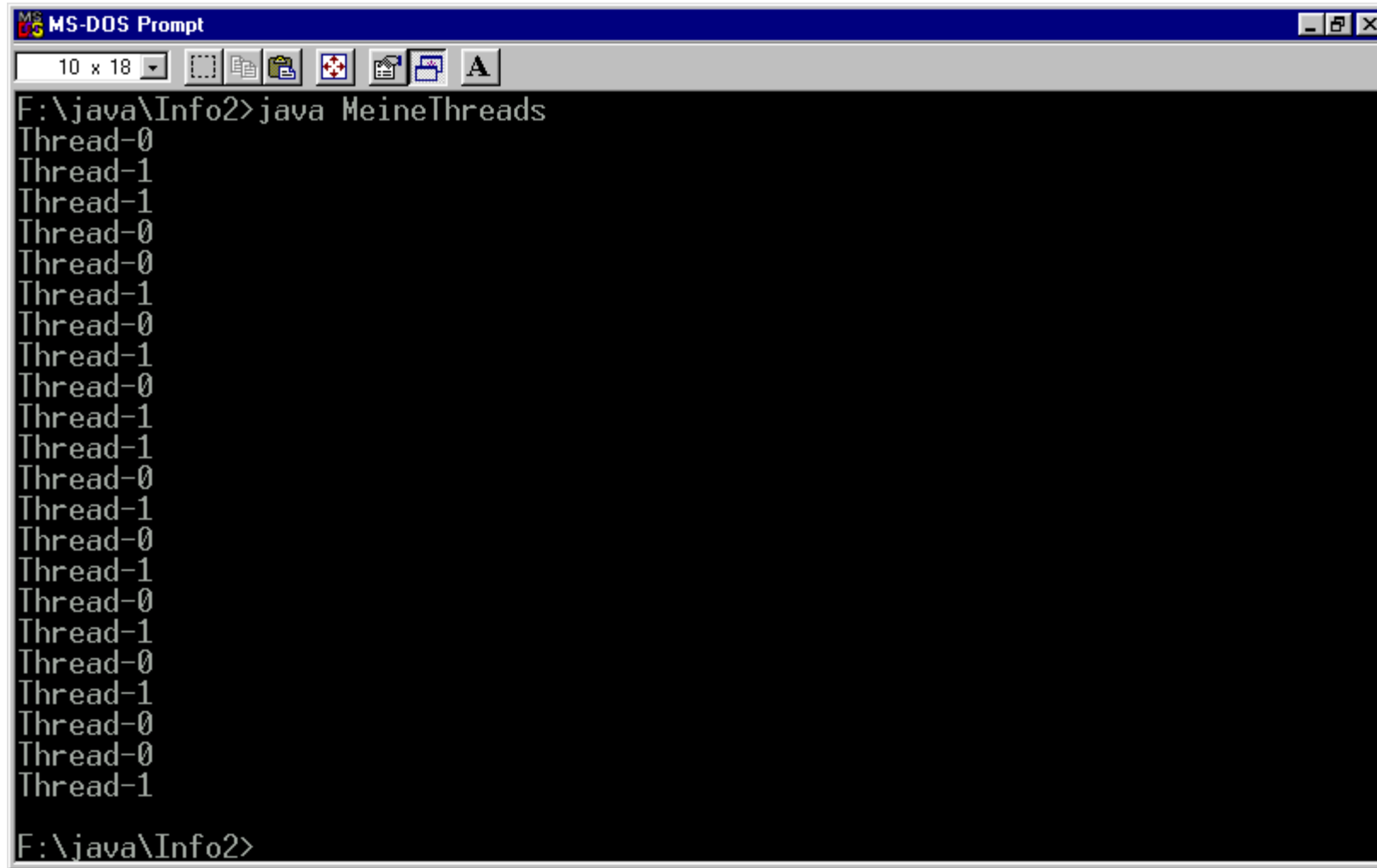
Erzeugten Thread
starten.

Weiteren Thread
deklarieren,
erzeugen und
starten.

Nach dem Starten
bearbeiten beide
Threads die
Methode run.

Beispiel: MeineThreads (2)

Die nebenläufige Bearbeitung der Threads brachte auf dem in der Vorlesung eingesetzten Laptop folgenden Output.



```
MS-DOS Prompt
10 x 18
F:\java\Info2>java MeineThreads
Thread-0
Thread-1
Thread-1
Thread-0
Thread-0
Thread-1
Thread-0
Thread-1
Thread-0
Thread-1
Thread-1
Thread-0
Thread-1
Thread-1
Thread-0
Thread-1
Thread-0
Thread-1
Thread-0
Thread-1
Thread-0
Thread-1
F:\java\Info2>
```

Man erkennt deutlich den Einfluss der Einplanung durch das Betriebssystem.

3.4. Inter-Prozess-Kommunikation und Synchronisation

Kooperierende Prozesse müssen in der Lage sein, Informationen auszutauschen.

3.4.1. Grundlegende Betrachtung

3.4.2. Wechselseitiger Ausschluss (Mutual Exclusion)

3.4.3. „Monitore“

3.4.4. Monitore mit Bedingungssynchronisation

3.4.5. Monitore zur Vergabe gleichartiger Ressourcen

3.4.1. Grundlegende Betrachtung

Klassische „Inter-Prozess-Kommunikation“ arbeitet mit gemeinsamem Speicher.

Beispiel: Print Spooler

1. Der druckwillige Prozess legt den Namen einer druckfähigen Datei im sog. „**Spooler-Directory**“ ab.
2. Ein anderer Prozess (der sog. **Printer Dämon**)
 - überprüft regelmäßig das „Spooler Directory“,
 - veranlasst das Ausdrucken dort vorhandener Dateien und
 - entfernt die Einträge im Spooler Directory.

In verteilten Systemen (z.B. in „Lokalen Netzen“, „Local Area Networks“, LANs) wird zusätzlich ein **Konzept zur Übermittlung von Nachrichten** benötigt:

- **send (destination, message)**
 - **receive (source, message)**
- } Das Betriebssystem muss Mechanismen bereitstellen, welche die gesicherte Übertragung von Nachrichten ermöglichen.

Auch bei verteilten Systemen gibt es jeweils **lokal eine Kommunikation über Speicher**, die vom Kommunikations-Subsystem und den „Anwendern“ gemeinsam genutzt werden.

3.4.2. Wechselseitiger Ausschluss (Mutual Exclusion)

Verändern mehrere Prozesse die Inhalte gemeinsamer Variablen, dann kann eine ungünstige Verzahnung (oder echte Parallelausführung) zu **inkonsistenten Daten** führen.

Beispiel: Gemeinsame Nutzung einer Variablen

Seien

- **p und q zwei Prozesse** mit
- **lokalen Variablen temp** und
- **gemeinsam genutzter Variable konto.**

p	<u>konto = 50;</u>	<u>temp = konto;</u>	<u>konto = temp + 10 ;</u>
q	<u>temp = 5;</u>	<u>konto = temp + 10 ;</u>	

Kritische Abschnitte sind Bereiche mit Anweisungen, deren Ausführung den Ausschluss anderer Prozesse erforderlich macht:

Wechselseitiger Ausschluss (Mutual Exclusion)

- In jedem kritischen Abschnitt darf **maximal ein Prozess** sein.
- Ist ein Prozess im kritischen Abschnitt, dann müssen alle anderen Prozesse mit Wunsch nach Betreten desselben kritischen Abschnitts **warten**.

3.4.3. „Monitore“

Manche Programmiersprachen - so z.B. Java - unterstützen den Programmierer bei der Behandlung von kritischen Abschnitten durch sog. „Monitore“.

Ein Monitor ist ein Software-Modul, der Daten und Operationen darauf kapselt.

Idee:

- **Kritische Abschnitte** auf Daten werden **als Monitor-Operationen** formuliert.
- Auf die Daten kann **nur durch Aufruf von Monitor-Operationen** zugegriffen werden.
- Die **Monitor-Operationen** werden **unter wechselseitigem Ausschluss** ausgeführt (Sicherstellung durch das Laufzeitsystem).



Monitore in Java

In Java können Methoden als „**synchronized**“ gekennzeichnet werden:

- Zu jeder Zeit kann **pro Objekt** nur **höchstens eine** der als „**synchronized**“ gekennzeichneten **Instanzenmethoden** ausgeführt werden.
- Zu jeder Zeit kann **pro Klasse** nur **höchstens eine** der als „**synchronized**“ gekennzeichneten **Klassenmethoden** ausgeführt werden.
- **Sperren für Klassen- und Objektmethoden sind voneinander unabhängig.**

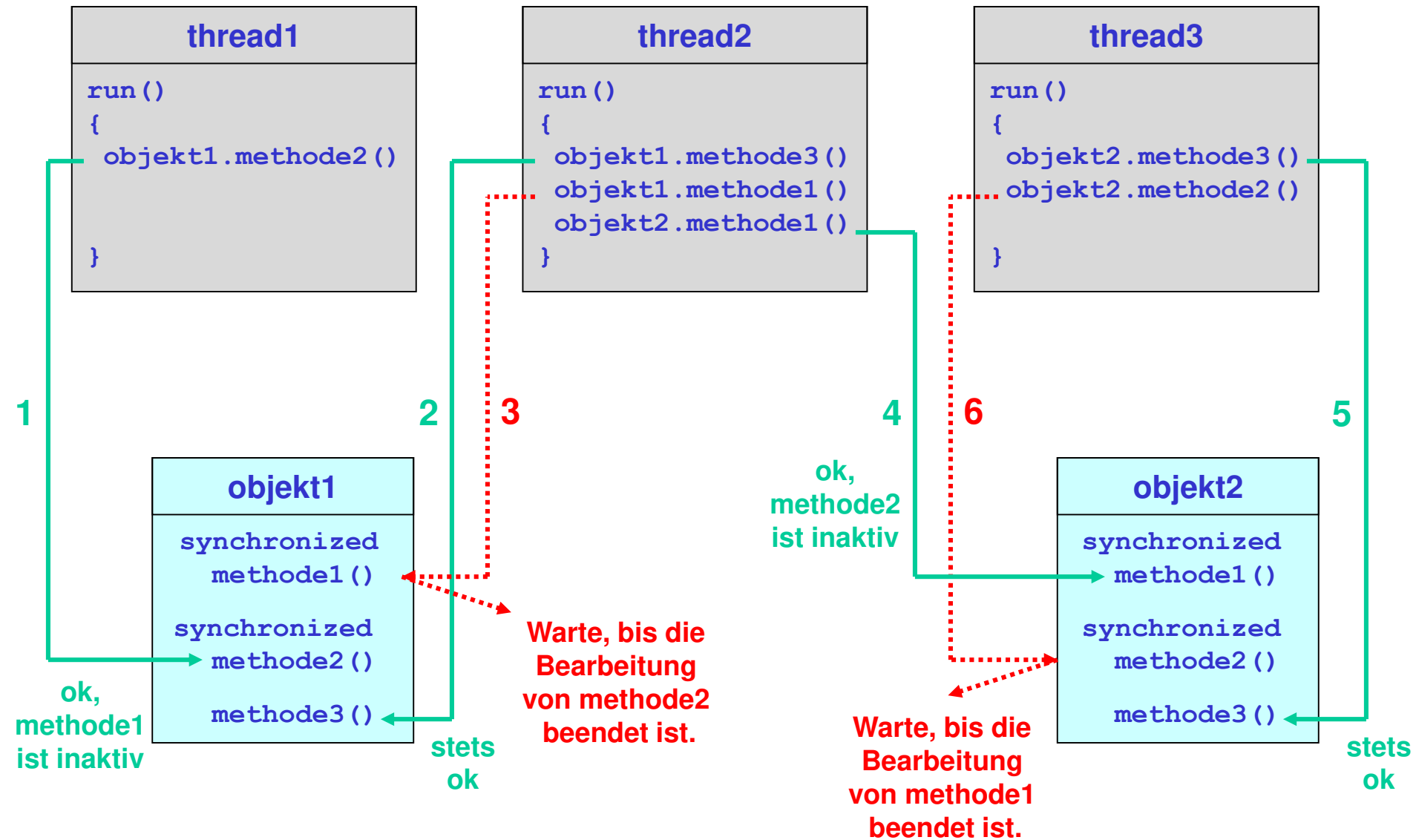
Beispiel:

```
class MeineKlasse
{
    synchronized public void methode1()
    {
        //Code für diese Methode ...
    }

    synchronized public void methode2()
    {
        //Code für diese Methode ...
    }

    public void methode3()
    {
        //Code für diese Methode ...
    }
}
```

Beispiel: Wechselseitiger Ausschluss in Java (Fortsetzung)



3.4.4. Monitore mit Bedingungssynchronisation

Bei einer „Bedingungssynchronisation“ müssen Prozesse bzw. Threads **warten, bis eine bestimmte Bedingung erfüllt** ist. Erst dann kann die Bearbeitung fortgesetzt werden.

Beispiel: Ein Prozess kann erst **in einen Puffer schreiben**, **wenn** dieser **nicht mehr voll** ist.

Bei **Java** gibt es **vordefinierte Methoden** der Klasse `Object`, die

- zur Bedingungssynchronisation eingesetzt werden und
- nur aus **synchronized**-Methoden aufgerufen werden können:

wait ()	blockiert den aufrufenden Prozess (bzw. Thread): Der aufrufende Thread wartet auf das aktuelle Objekt und gibt alle etwaigen Sperren mittels synchronized-Methoden auf diesem Objekt auf.
notify ()	weckt einen (Wahl: zufällig) der beim jeweiligen Objekt wartenden Threads.
notifyAll ()	weckt alle Threads, die am jeweiligen Objekt warten (ggf. „Wettrennen“)

Nachdem ein blockierter Prozess geweckt wurde, muss er die Bedingung, auf die er wartet, erneut prüfen: Vielleicht hat ein anderer Prozess schon zugegriffen!

```
while (avail < n)
try {wait();} catch (InterruptedException e) {}
```

3.4.5. Monitore zur Vergabe gleichartiger Ressourcen

Monitore können auch eingesetzt werden, um eine **begrenzte Anzahl $k \geq 1$ von gleichartigen Ressourcen** zu verwalten:

Prozesse fordern n Ressourcen an und geben sie später zurück.

Die **Wartebedingung** lautet hier: **Ist die geforderte Anzahl Ressourcen aktuell verfügbar ?**

Beispiele:

- Museum mit k Walkmen zur Vergabe an Besuchergruppen (an Besucher).
- Taxi-Unternehmen mit k Fahrzeugen.

Anmerkungen:

- Es kann sich auch um **abstrakte Ressourcen** handeln.
 - Das Recht, eine Brücke begrenzter Kapazität (max. Gewicht) zu befahren.
 - Das Recht, an einem Praktikum mit max. 16 Teilnehmern teilzunehmen.
- Es sind auch **Ressourcen mit Identität** möglich. Dann muss der Monitor die jeweilige Identität in einer entsprechenden Datenstruktur verwalten.
 - Nummern von Taxis eines Taxi-Unternehmens.
 - Adressen von Speicherblöcken, die eine Speicherverwaltung vergibt.

Beispiel: Monitor zur Ressourcenvergabe in Java

In Java kann die Vergabe einer begrenzten Zahl gleichartiger Ressourcen wie im folgenden Beispiel realisiert werden:

```
class Monitor
{ private int avail;                                // Anzahl verfügbarer Ressourcen
  Monitor (int a) { avail = a; }

  synchronized void getElem (int n, int who)        // Monitor gibt n Elemente ab
  { System.out.println("Client"+who+" needs "+n+", available "+avail);

    while (avail < n) // Bedingung in Warteschleife prüfen
    { try { wait(); } catch (InterruptedException e) {}
      }                                                    // try ... catch nötig wegen wait()
    avail -= n;
    System.out.println("Client"+who+" got "+n+", available "+avail);
  }

  synchronized void putElem (int n, int who)        // Monitor nimmt n Elemente zurück
  { avail += n;
    System.out.println("Client"+who+" put "+n+", available "+avail);
    notifyAll();                                           // Alle Wartenden können Bedingung erneut prüfen
  }
}
```


Beispiel: Monitor zur Ressourcenvergabe in Java (2)

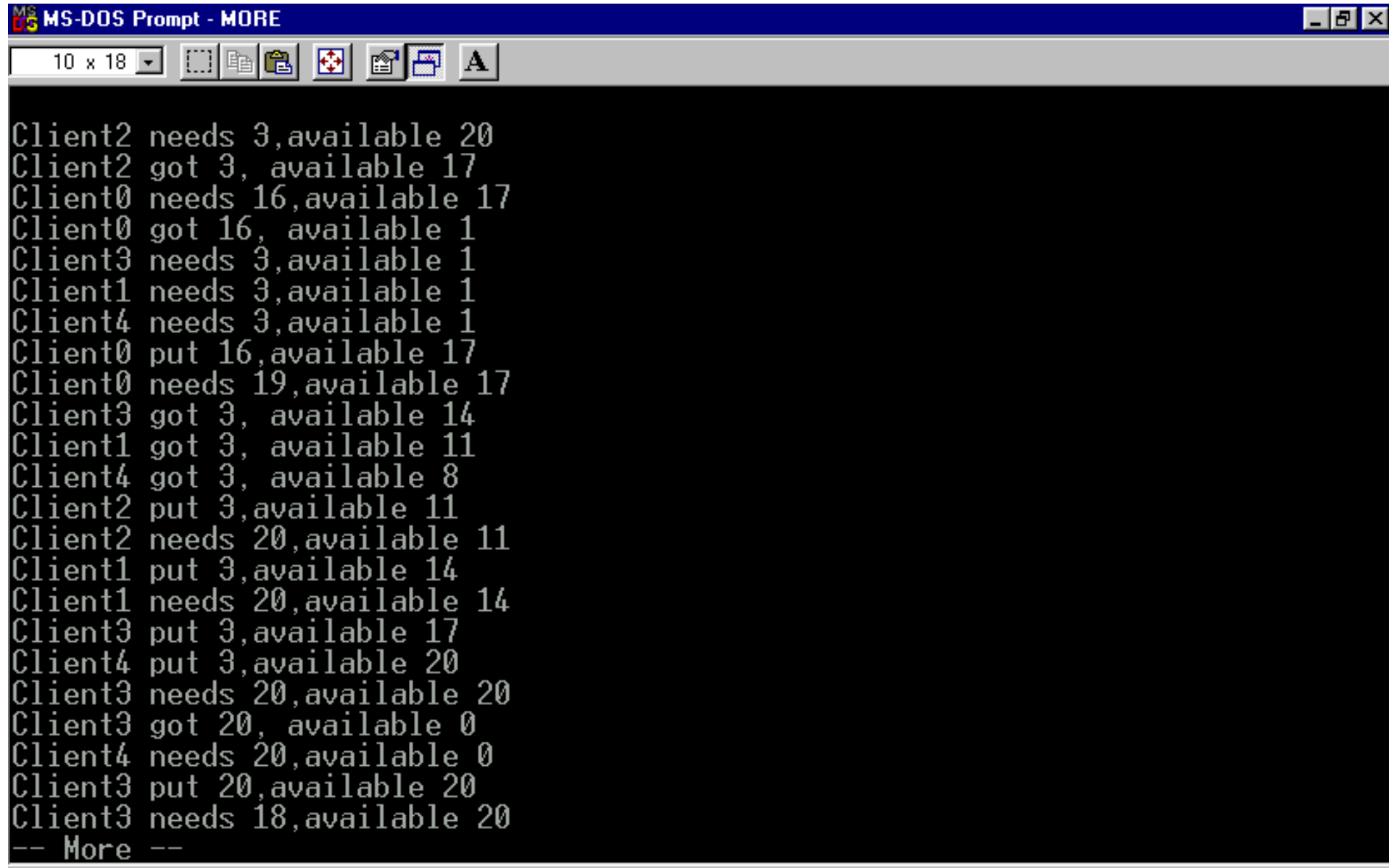
```
import java.util.Random;

class Client extends Thread
{ private Monitor mon; private Random rand;
  private int ident, rounds, max;
  Client (Monitor m, int id, int rd, int avail)
  { ident = id; rounds = rd; mon = m; max = avail;
    rand = new Random();           // Neuer Zufallszahlengenerator
  }

  public void run ()
  { while (rounds > 0)
    { int m = Math.abs(rand.nextInt()) % max + 1;
      mon.getElem (m, ident);       // m Elemente anfordern
      try { sleep (Math.abs(rand.nextInt()) % 1000 + 1); }
          catch (InterruptedException e) {}
      mon.putElem (m, ident);       // m Elemente zurückgeben
      rounds--;
    }
  }
}
```

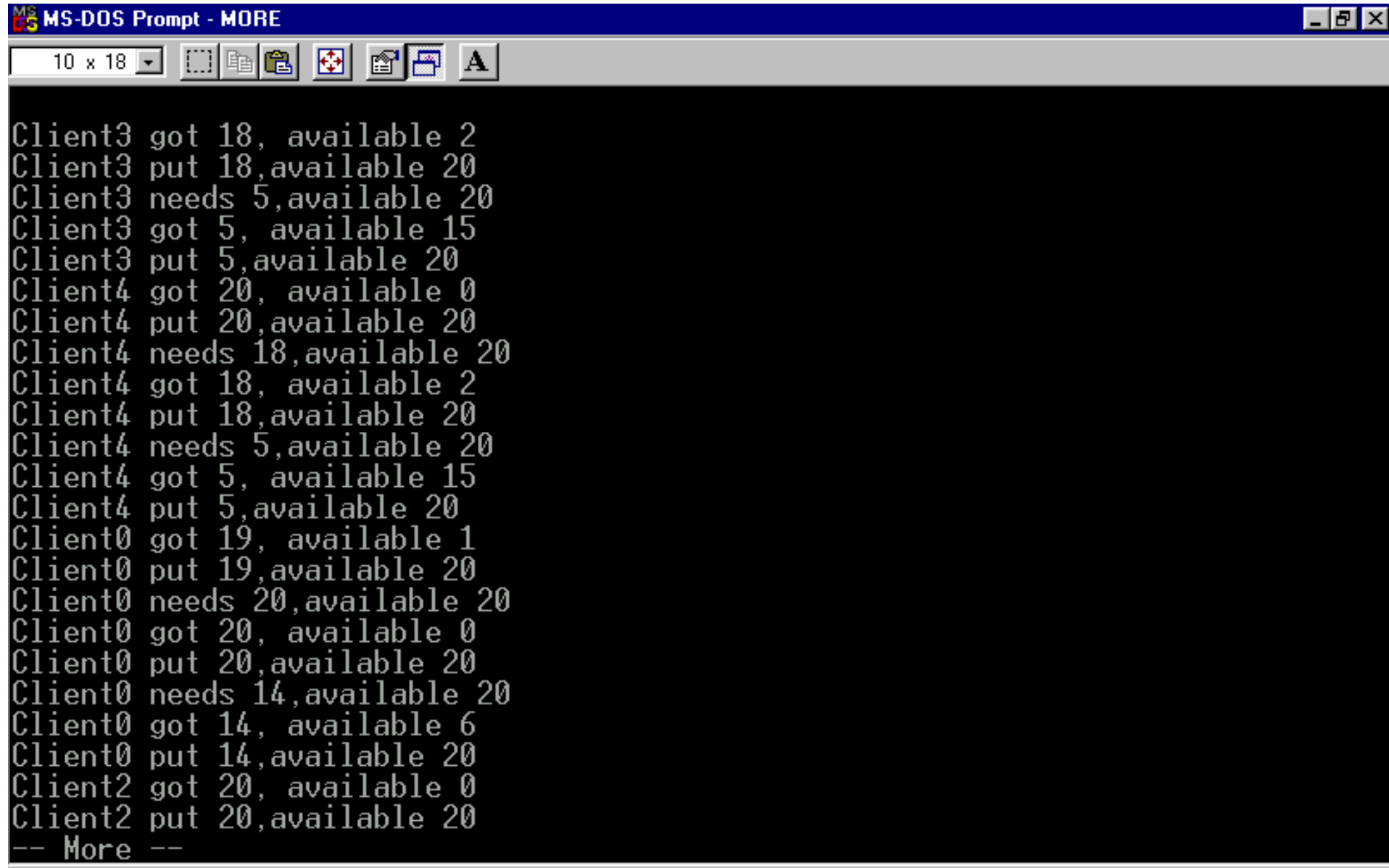
```
class TestMonitor
{ public static void main (String[] args)
  { int avail = 20;
    Monitor mon = new Monitor (avail);
    for (int i = 0; i < 5; i++)
      new Client(mon, i, 4, avail).start();
  }
}
```

Beispiel: Monitor zur Ressourcenvergabe in Java (3)



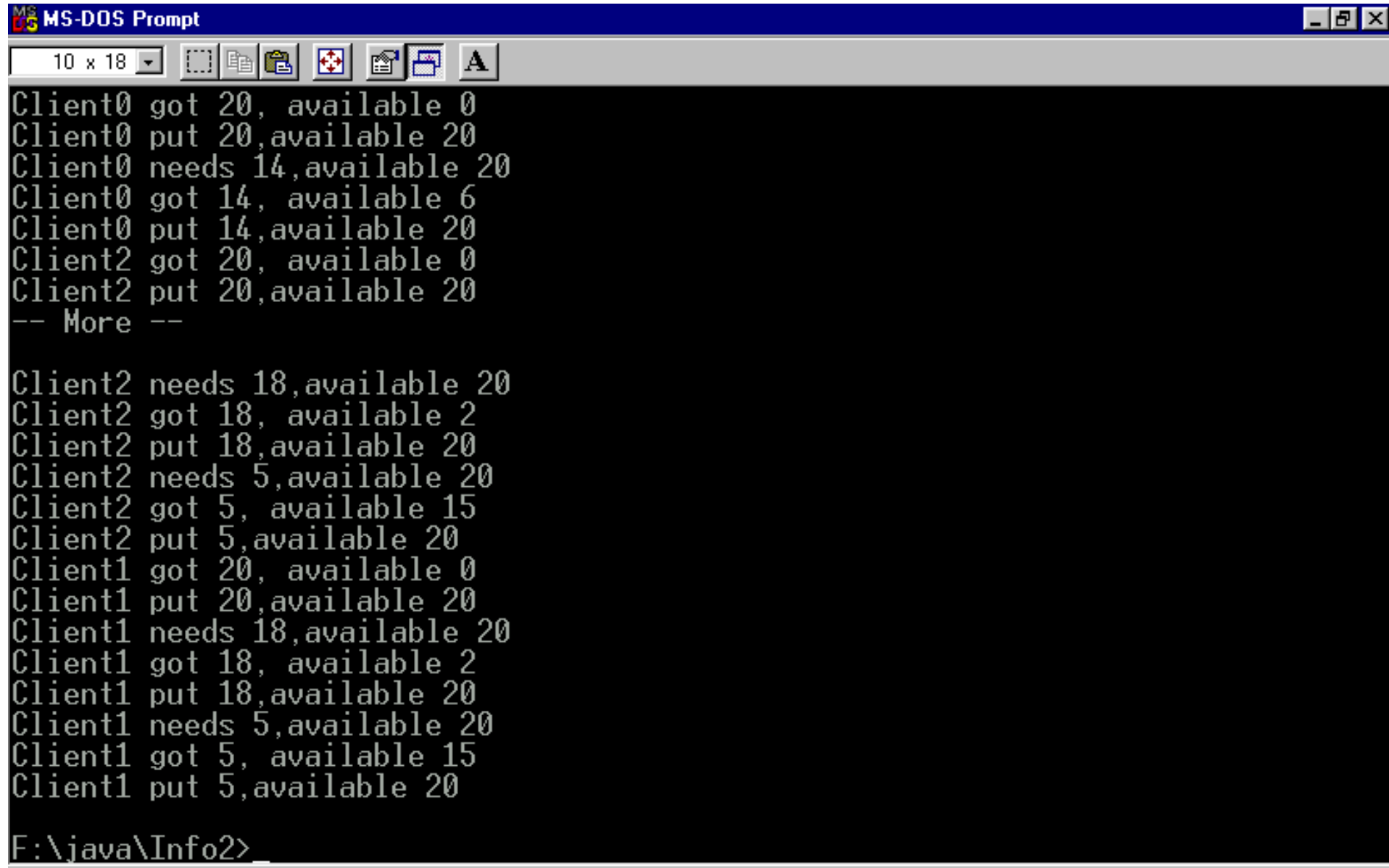
```
Client2 needs 3,available 20
Client2 got 3, available 17
Client0 needs 16,available 17
Client0 got 16, available 1
Client3 needs 3,available 1
Client1 needs 3,available 1
Client4 needs 3,available 1
Client0 put 16,available 17
Client0 needs 19,available 17
Client3 got 3, available 14
Client1 got 3, available 11
Client4 got 3, available 8
Client2 put 3,available 11
Client2 needs 20,available 11
Client1 put 3,available 14
Client1 needs 20,available 14
Client3 put 3,available 17
Client4 put 3,available 20
Client3 needs 20,available 20
Client3 got 20, available 0
Client4 needs 20,available 0
Client3 put 20,available 20
Client3 needs 18,available 20
-- More --
```

Beispiel: Monitor zur Ressourcenvergabe in Java (4)



```
MS-DOS Prompt - MORE
10 x 18
Client3 got 18, available 2
Client3 put 18,available 20
Client3 needs 5,available 20
Client3 got 5, available 15
Client3 put 5,available 20
Client4 got 20, available 0
Client4 put 20,available 20
Client4 needs 18,available 20
Client4 got 18, available 2
Client4 put 18,available 20
Client4 needs 5,available 20
Client4 got 5, available 15
Client4 put 5,available 20
Client0 got 19, available 1
Client0 put 19,available 20
Client0 needs 20,available 20
Client0 got 20, available 0
Client0 put 20,available 20
Client0 needs 14,available 20
Client0 got 14, available 6
Client0 put 14,available 20
Client2 got 20, available 0
Client2 put 20,available 20
-- More --
```

Beispiel: Monitor zur Ressourcenvergabe in Java (5)



```
MS-DOS Prompt
10 x 18
Client0 got 20, available 0
Client0 put 20,available 20
Client0 needs 14,available 20
Client0 got 14, available 6
Client0 put 14,available 20
Client2 got 20, available 0
Client2 put 20,available 20
-- More --

Client2 needs 18,available 20
Client2 got 18, available 2
Client2 put 18,available 20
Client2 needs 5,available 20
Client2 got 5, available 15
Client2 put 5,available 20
Client1 got 20, available 0
Client1 put 20,available 20
Client1 needs 18,available 20
Client1 got 18, available 2
Client1 put 18,available 20
Client1 needs 5,available 20
Client1 got 5, available 15
Client1 put 5,available 20
F:\java\Info2>
```

3.5. Deadlocks

In vielen Fällen ist die **exklusive Zuordnung eines Betriebsmittels** (z.B. E/A-Gerät, Speicherplatz, ...) für längere Zeit erforderlich.

Blockieren sich mehrere Prozesse gegenseitig, indem alle auf ein Ereignis warten, das nur von einem anderen Prozess der betrachteten Menge ausgelöst werden kann, dann spricht man (vgl. „Dining Philosophers“) von einem

Deadlock (Verklemmung).

Es gibt **vier notwendige Bedingungen** für Deadlocks:

1. Mutual exclusion

Die Prozesse benötigen exklusiven Zugriff.

2. Hold and wait

Die Prozesse verfügen über Betriebsmittel, halten diese und warten auf mindestens je ein weiteres.

3. No preemption

Betriebsmittel können erst nach Abschluss der erfolgreichen Nutzung entzogen werden.

4. Circular wait condition

Es gibt eine geschlossene Kette von Prozessen, die Betriebsmittel anfordern, die der jeweilige Nachfolger in der Kette hält.

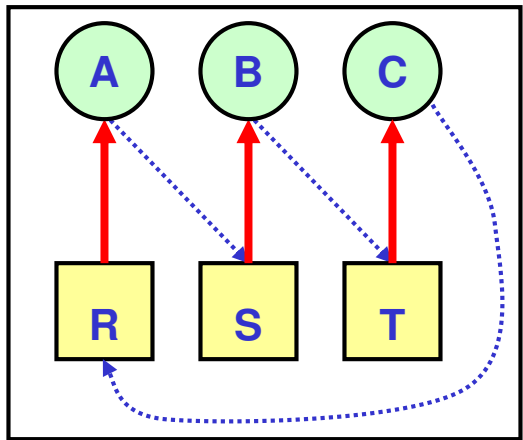
Deadlocks können also **verhindert** werden, **indem** sichergestellt wird, dass **mindestens eine dieser Bedingungen nicht erfüllt** wird.

Beispiel: Entstehung eines Deadlocks

Das nachfolgende Beispiel zeigt, dass Deadlocks nicht allein aus einer Betrachtung der Forderungen erkennbar sind.

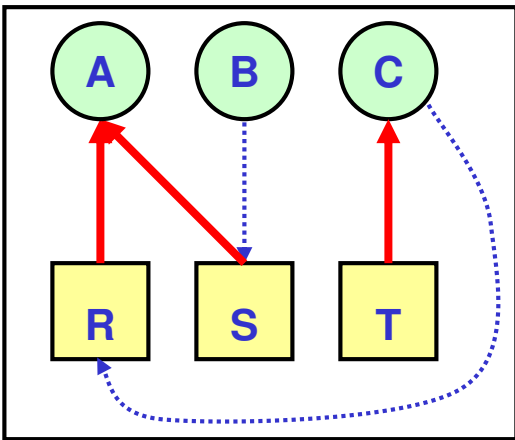
Prozess A:	Request R,	Request S,	Release R,	Release S
Prozess B:	Request S,	Request T,	Release S,	Release T
Prozess C:	Request T,	Request R,	Release T,	Release R

Führt dies zum Deadlock ?

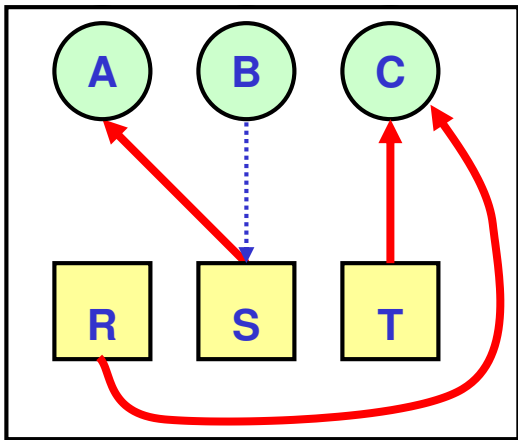


Deadlock !

Ungünstige Reihenfolge führte zum Deadlock.



A aktiv, C wartet



A gibt R frei, C erhält R

Günstige Reihenfolge führt nicht zum Deadlock.

→ Betriebsmittel ist zugeordnet → Betriebsmittel wird angefordert

3.6. Scheduling-Strategien

Sind mehrere Prozesse im Zustand „Ready“, dann muss das Betriebssystem entscheiden, welcher Prozess zuerst laufen darf.

Diese Aufgabe nimmt der Scheduler (= Planer) wahr.

[3.6.1. Kriterien zur Beurteilung von Scheduling-Algorithmen](#)

[3.6.2. Das allgemeine Prioritätenverfahren](#)

[3.6.3. Die FIFO-Strategie](#)

[3.6.4. Round Robin Scheduling](#)

[3.6.5. E/A-orientiertes Prioritätenverfahren](#)

[3.6.6. SPT: Shortest Processing Time First](#)

3.6.1. Kriterien zur Beurteilung von Scheduling-Strategien

Wichtige Kriterien zur Beurteilung von Scheduling-Algorithmen sind:

- 1. Fairness:** Bekommen alle Prozesse einen **fairen Anteil**?
- 2. Effizienz:** Ist das Betriebsmittel **gut ausgelastet** ?
Wieviel **CPU-Zeit** verbraucht der **Scheduler** ?
- 3. Antwortzeit:** Ist die **Antwortzeit für interaktive Nutzer** akzeptabel ?
(Ausgleich zwischen „Hintergrundlast“ und „Vordergrundlast“)
- 4. Verweilzeit:** Ist die **Laufzeit von Hintergrundprozessen** akzeptabel ?
- 5. Durchsatz:** Werden hinreichend viele **Aufträge pro Zeiteinheit** bearbeitet ?

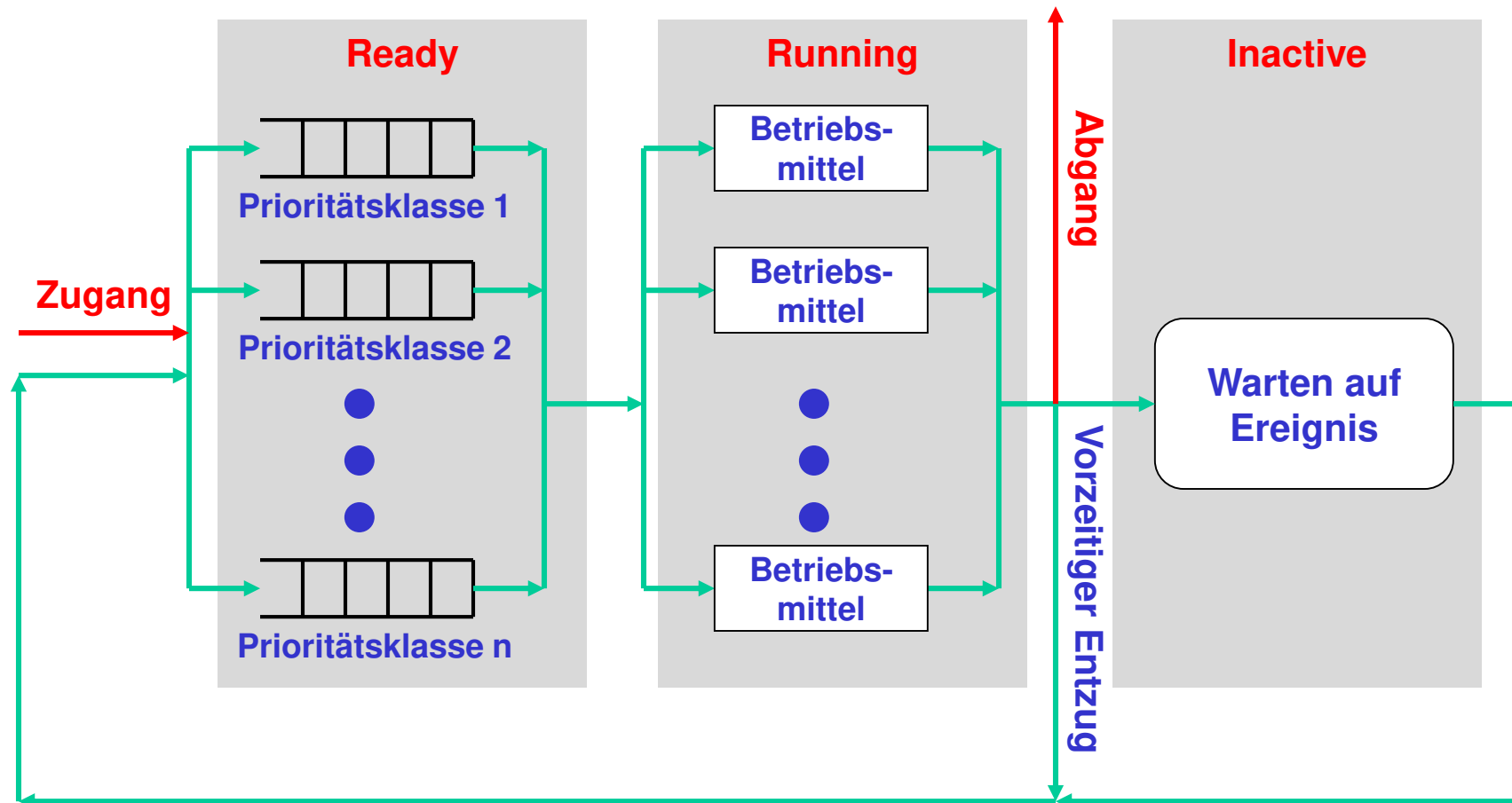
Scheduling-Algorithmen stellen **Kompromisse** zwischen den genannten Kriterien dar.

Wir beschränken die nachfolgenden Betrachtungen i.W. auf das **CPU-Scheduling**.
Auch bei anderen Betriebsmitteln ist aber ein geeignetes Scheduling erforderlich.

3.6.2. Das allgemeine Prioritätenverfahren

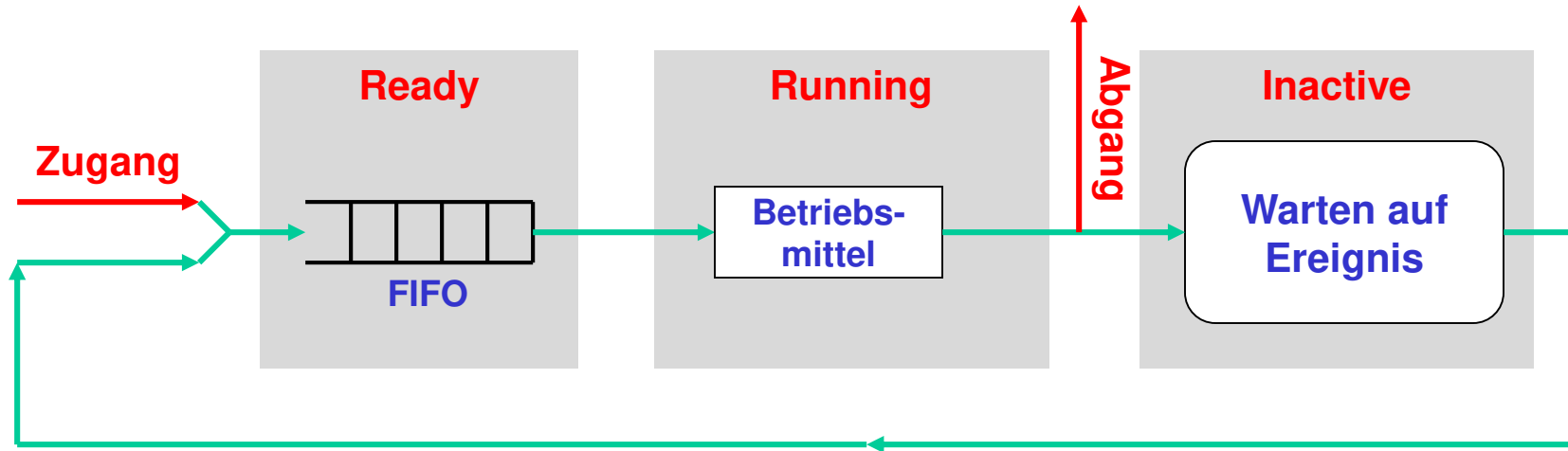
Fast alle Scheduling-Algorithmen sind Sonderfälle des allgemeinen Prioritätenverfahrens:

- a) Auswahl einer Prioritätsklasse
- b) Einordnung innerhalb der ausgewählten Klasse
- c) Auswahl der Prioritätsklasse für die Betriebsmittelzuteilung.



3.6.3. Die FIFO-Strategie

Die FIFO-Strategie ergibt sich als Sonderfall mit nur einer Prioritätsklasse:



Achtung:

Extreme Langläufer blockieren hier das gesamte System, falls - wie hier - kein vorzeitiger Entzug von Betriebsmitteln erfolgt.

3.6.4. Round Robin Scheduling

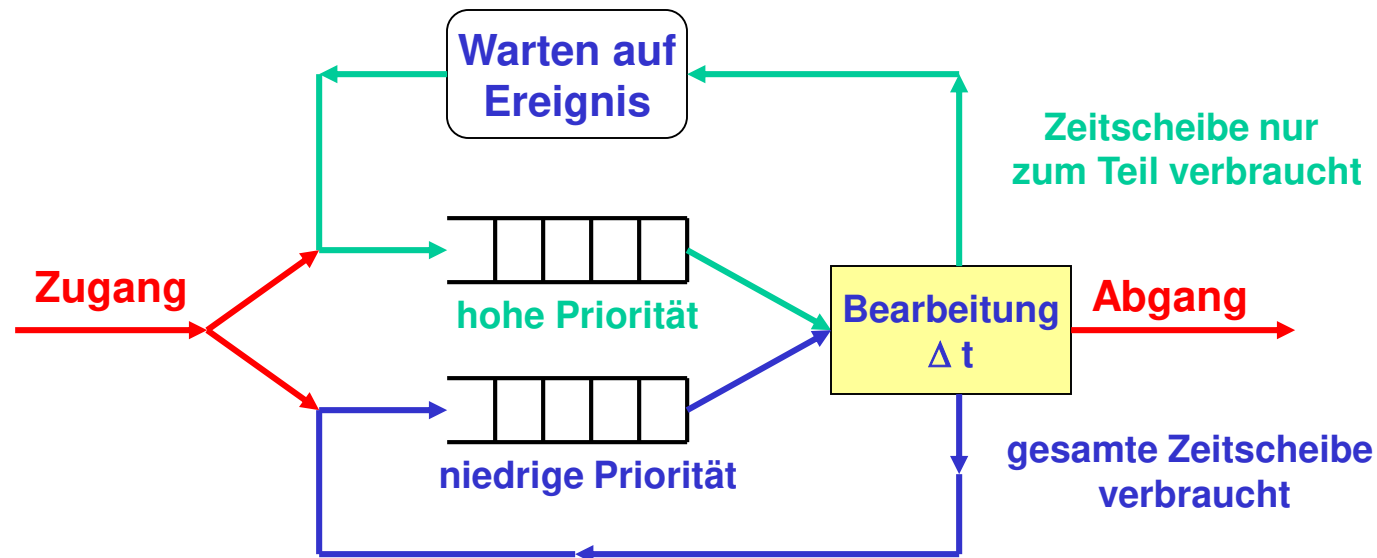
„Round Robin“ ist die älteste, einfachste und am häufigsten eingesetzte Strategie.

Bei Round Robin erfolgt die CPU-Zuteilung mit zeitlicher Beschränkung („**preemptive scheduling**“).

Varianten des Verfahrens:

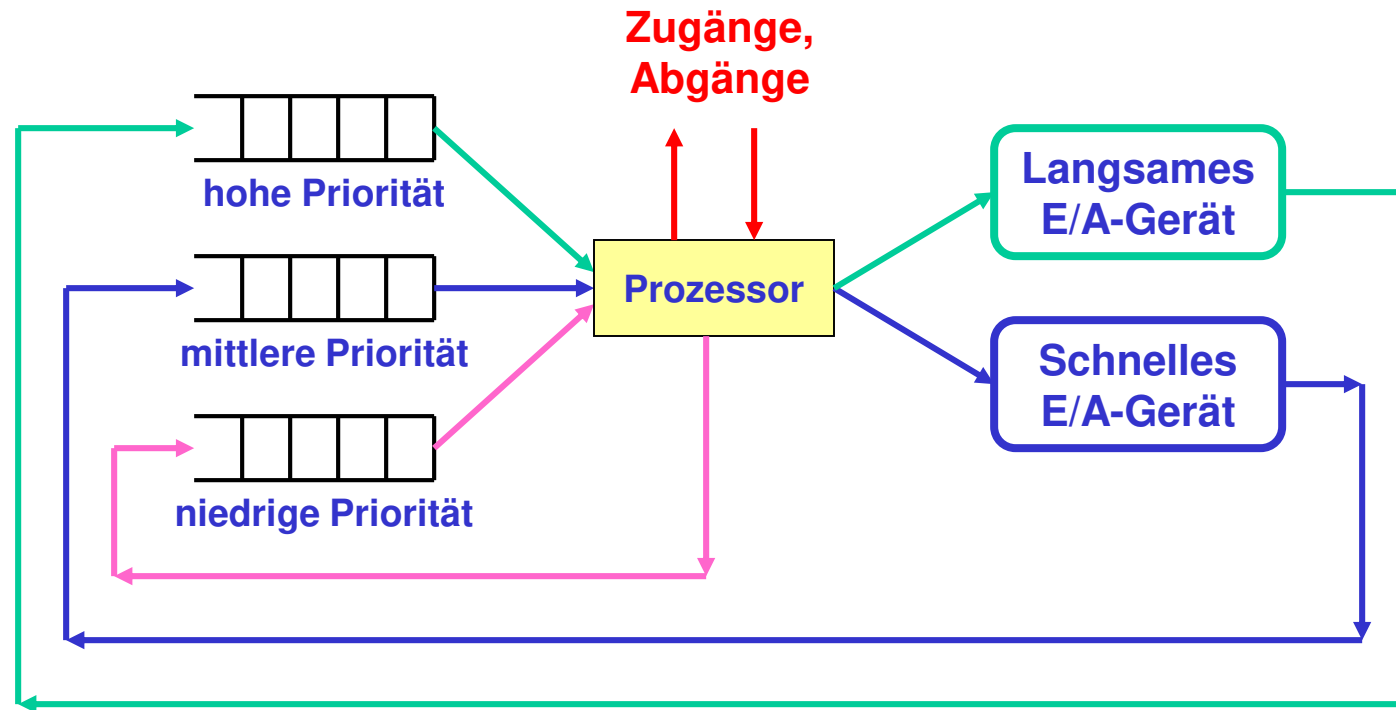
- gleich große / unterschiedlich große Zeitscheiben,
- eine / mehrere Prioritätsklassen,
- Bevorzugung von Prozessen mit kurzer Nutzungszeit.

Beispiel:



3.6.5. E/A-orientiertes Prioritätenverfahren

Häufig ergibt sich eine besonders gute Auslastung des Gesamtsystems, wenn beim CPU-Scheduling die Prozesse bevorzugt werden, die ein langsames E/A-Gerät benötigen.



3.6.6. SPT: Shortest Processing Time First

Warten mehrere Prozesse mit gleicher Priorität, dann wird die kürzeste mittlere Bearbeitungszeit erzielt, wenn die „Kurzläufer“ zuerst bearbeitet werden.

Beispiel:

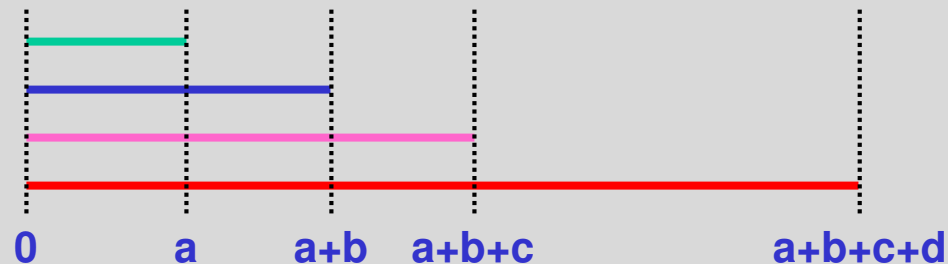
Es warten 4 Prozesse mit reinen Laufzeiten a , b , c und d , die hintereinander bearbeitet werden sollen: Zuerst a , dann b , dann c , dann d .

Dann werden Prozesse beendet zu den Zeiten:

Ende des 1. Prozesses: a
Ende des 2. Prozesses: $a + b$
Ende des 3. Prozesses: $a + b + c$
Ende des 4. Prozesses: $a + b + c + d$

Die mittlere Bearbeitungszeit ergibt sich dann als

$$\frac{4a + 3b + 2c + d}{4}$$



Offenbar wird die kürzeste mittlere Bearbeitungszeit erzielt für $a \leq b \leq c \leq d$.

Restlaufzeit und Schätzung der Gesamtlaufzeit

Werden Prozesse unterbrochen, um auch später laufbereit werdende Prozesse mit kürzeren Laufzeiten vorzuziehen, dann wird SPT zu

SRPT = Shortest Remaining Processing Time First

SRPT ist eine „Optimalstrategie“ mit beweisbar kürzesten mittleren Bearbeitungszeiten.

Herausforderung beim praktischen Einsatz:

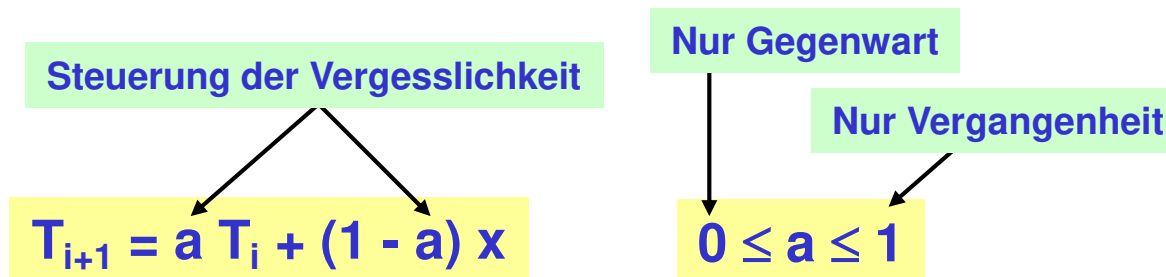
- Woher kennt der Scheduler die Gesamtlaufzeit eines Prozesses ?

Heuristik: Lerne aus der Vergangenheit

Sei

- T_i der **aktuelle Schätzwert**
- x der Bedarf an CPU-Zeit im letzten Inactive-Inactive-Zyklus („**aktueller Messwert**“).

Dann schätze



3.7. Besonderheiten bei Echtzeitbetrieb

Bei Echtzeitbetrieb (real-time processing) muss/soll der Computer die Ergebnisse nicht nur korrekt, sondern auch **innerhalb einer von Außen vorgegebenen Zeitspanne** verfügbar machen: Die Forderung nach Rechtzeitigkeit (Einhaltung einer Deadline) ist hier eine zusätzliche Korrektheitsanforderung.

[3.7.1. Harte und weiche Zeitbedingungen](#)

[3.7.2. Zeitgerechte Einplanung](#)

[3.7.3. CPU-Scheduling für Einprozessorsysteme](#)

[3.7.4. CPU-Scheduling für Mehrprozessorsysteme](#)

[3.7.5. Scheduling bei variabler CPU-Leistung](#)

3.7.1. Harte und weiche Zeitbedingungen

Echtzeitsysteme werden nach der Art der Zeitbedingungen unterschieden:

a) Harte Zeitbedingungen

Die Zeitbedingungen stehen präzise fest, Überschreitung verursacht inakzeptablen Verlust oder Schaden.



Herzschrittmacher
Quelle: www.biotronik.de



GL-Klasse
Quelle: www.mercedes-benz.de

b) Weiche Zeitbedingungen

Die Zeitbedingungen sind nicht präzise festlegbar, Überschreitung verursacht Störungen, aber das Betriebsziel wird dennoch erreicht.



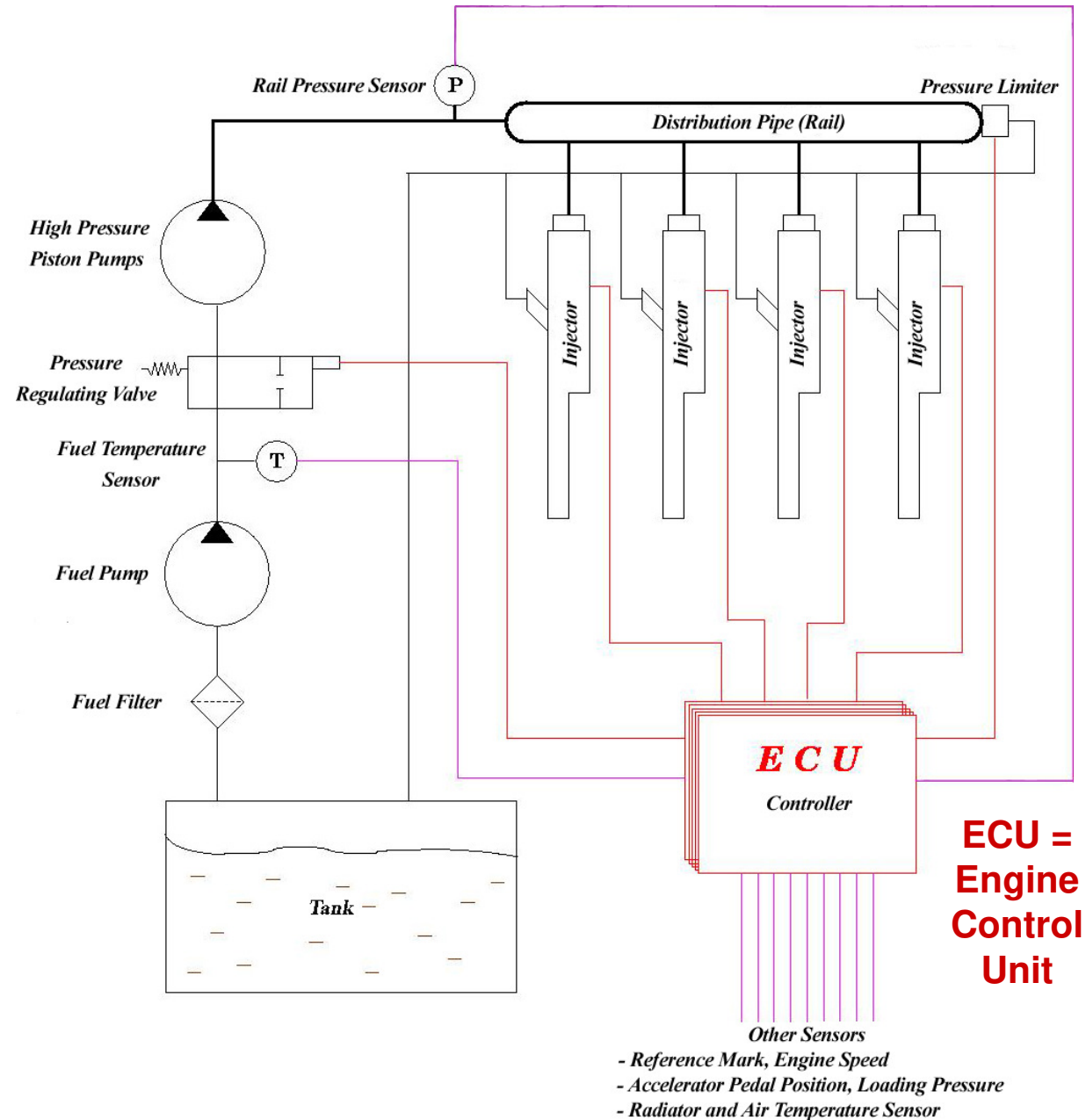
Buchungssystem
Quelle: www.lufthansa.de



Webradio
Quelle: www.einslive.de

Bei hinreichend lockerer Interpretation weicher Zeitbedingungen wird jedes System zum Echtzeitsystem.

Beispiel: Harte Zeitbedingungen bei „Common Rail“

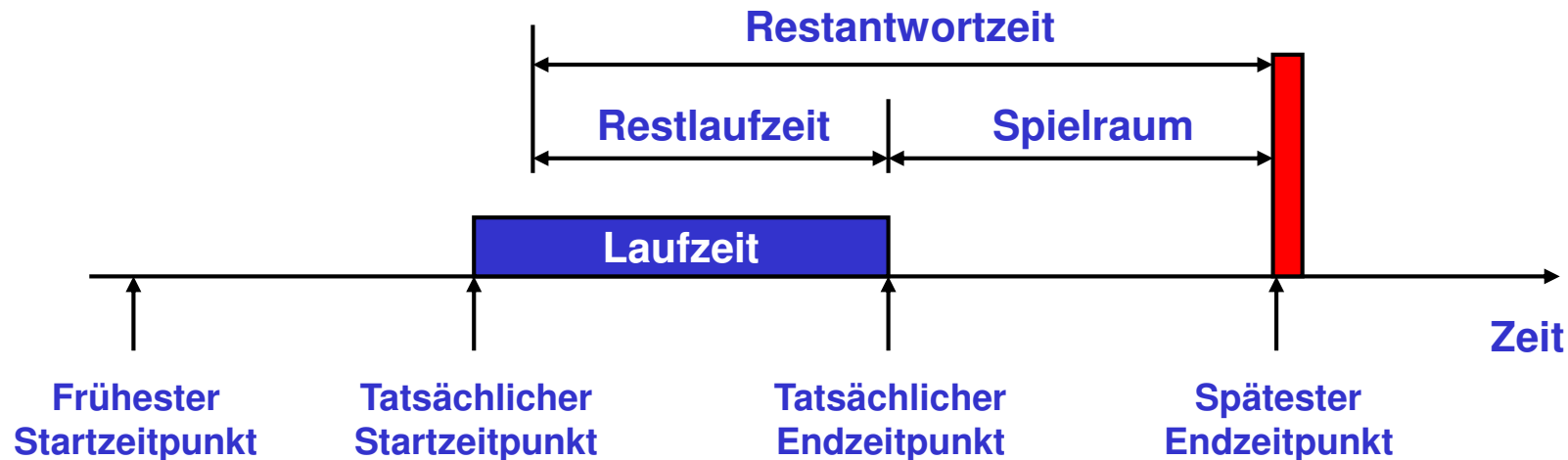


Quelle: Wikipedia

3.7.2. Zeitgerechte Einplanung

Für die zeitgerechte („timely“) Bearbeitung eines Prozesses sind wesentlich:

- **Laufzeit** des Prozesses
- **Frühester Startzeitpunkt** (wann wird er lauffähig?)
- **Tatsächlicher Startzeitpunkt**
- **Spätester Endzeitpunkt** („Deadline“)
- (... die Konkurrenzsituation)



In der Praxis erweist sich häufig die Bestimmung bzw. Vorhersage der Laufzeit als besonders problematisch:

Hier werden meist **Abschätzungen für den ungünstigsten Fall** eingesetzt.

3.7.3. CPU-Scheduling für Einprozessorsysteme

Die bekanntesten Strategien für das Scheduling in Echtzeit-Systemen sind:

3.7.3.1. Einplanung nach statischen Vorab-Prioritäten

3.7.3.2. Einplanung nach Antwortzeit (Earliest-Deadline-First)

3.7.3.3. Einplanung nach Spielraum

Bei allen drei Varianten wird die CPU bei Bedarf vorzeitig entzogen (Preemption).

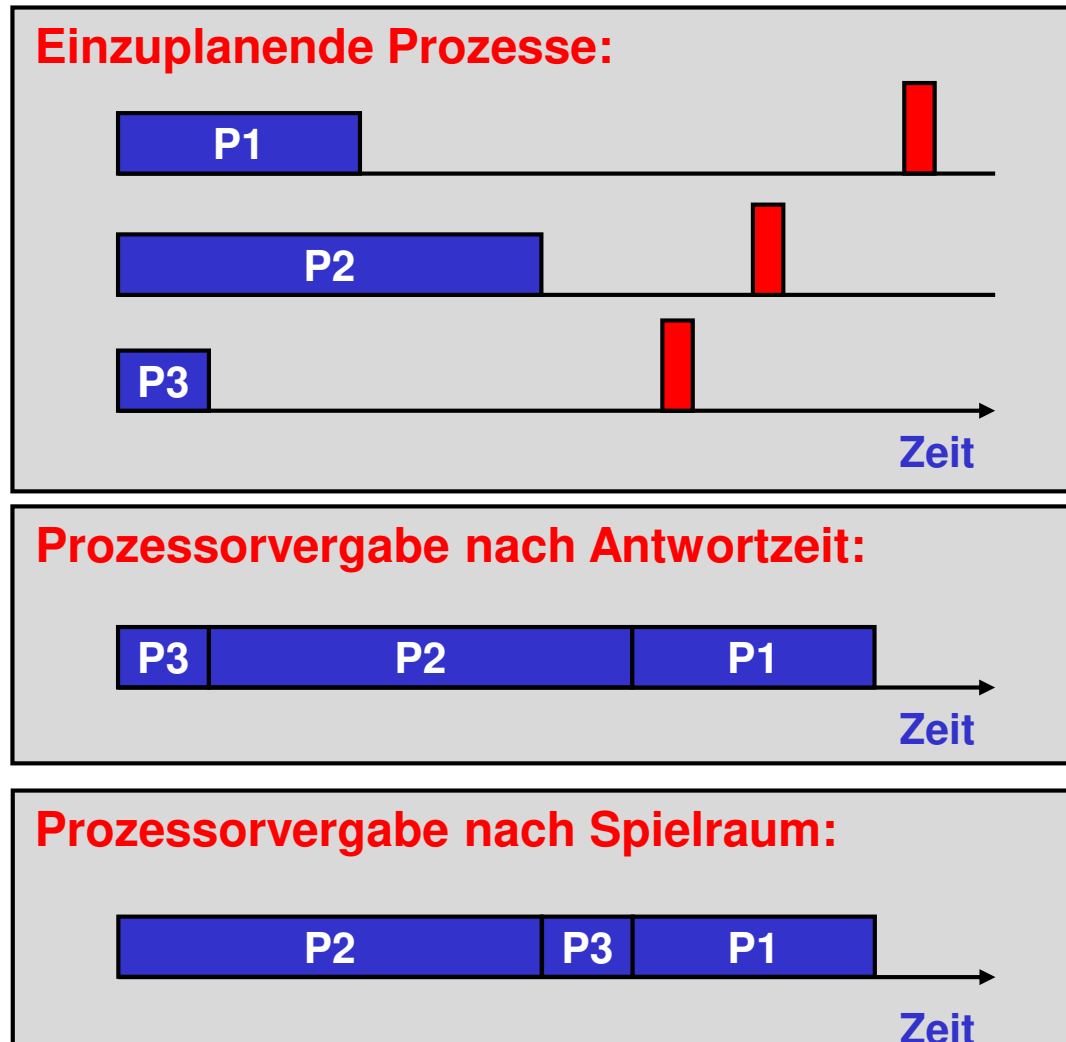
Einplanung nach Antwortzeit und Einplanung nach Spielraum sind **Optimalstrategien**:

Wenn es überhaupt einen zeitgerechten Schedule gibt, dann liefern beide Strategien zulässige Schedules.

Gantt-Diagramme (Gantt charts)

Zur graphischen Darstellung der Abläufe kommen oft Balkendiagramme zum Einsatz, die sog. Gantt-Diagramme (benannt nach dem Unternehmensberater Henry L. Gantt).

Die nachfolgende Graphik zeigt ein Beispiel mit 3 Prozessen und einem Prozessor.

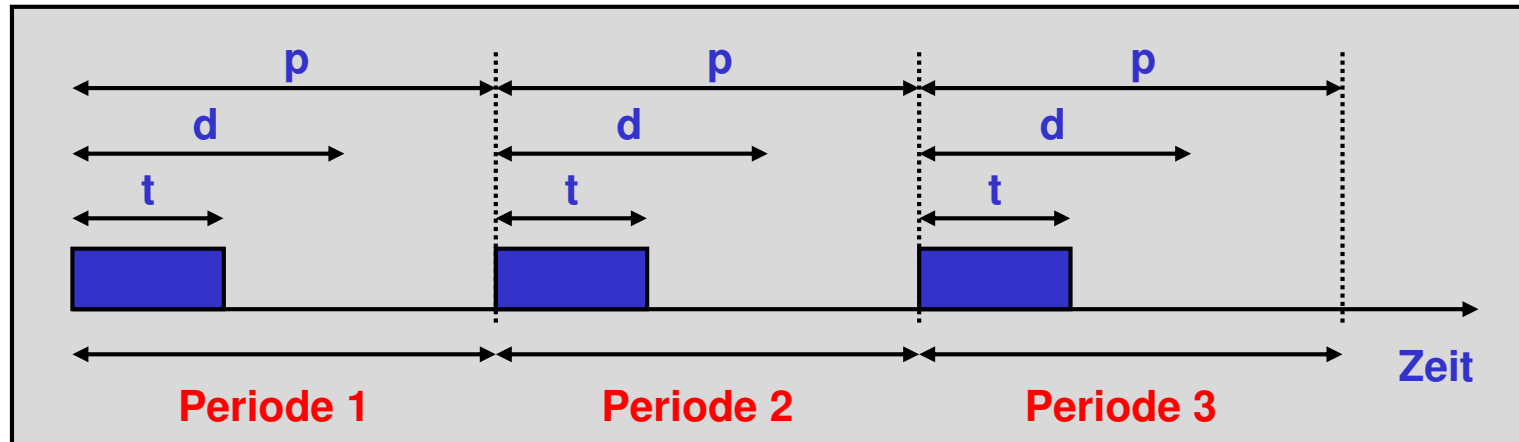


Szenarien mit periodischer Last

Echtzeitbetrieb wird besonders häufig in Szenarien mit periodischer Last eingesetzt:

Periodische Prozesse benötigen die CPU

- in festen zeitlichen Abständen (Perioden der Dauer **p**)
- für eine a priori bekannte Bearbeitungszeit **t**
- unter Einhaltung einer Deadline **d** mit $0 \leq t \leq d \leq p$.

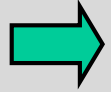


Wird die Last von einem Analog-Digital-Wandler bereit gestellt, dann muss die Abtastfrequenz (= Kehrwert der Periode) mindestens doppelt so hoch sein wie die höchste Frequenz, die im Analogsignal vorkommt („**Abtasttheorem**“).

Sprachsignale in Telefon-Qualität: Grenzfrequenz 3,4 kHz, Abtastperiode $1/8.000 \text{ s} = 125 \mu\text{s}$

Audio CD-Qualität: Grenzfrequenz 20 kHz, Abtastperiode $1/44.100 \text{ s} = \text{ca. } 22,8 \mu\text{s}$

3.7.3.1. Einplanung nach statischen Vorab-Prioritäten



Den Prozessen werden feste Prioritäten zugeordnet.

Zuteilung der CPU an den aktuell lauffähigen Prozess mit höchster Priorität.

Bei periodischen Systemen wird meist eine spezifische Ausprägung eingesetzt als sog.

„Rate-Monotonic Scheduling“: Je größer die Periode, desto kleiner die Priorität.

Wir betrachten ein Beispiel für Rate-Monotonic Scheduling:

Prozess P1:

- **Dauer der Periode:** $p_1 = 50 \text{ ZE}^*$
- **Bearbeitungsdauer:** $t_1 = 20 \text{ ZE}^*$
- **Deadline:** $d_1 = p_1$

*ZE = Zeiteinheit

Prozess P2:

- **Dauer der Periode:** $p_2 = 100 \text{ ZE}^*$
- **Bearbeitungsdauer:** $t_2 = 35 \text{ ZE}^*$
- **Deadline:** $d_2 = p_2$

*ZE = Zeiteinheit

Ist das System überlastet? Zur Beantwortung prüfen wir zunächst die CPU-Auslastung:

CPU-Auslastung durch P1: **Bearbeitungsdauer / Periode** = $20 / 50 = 0,4$

CPU-Auslastung durch P2: **Bearbeitungsdauer / Periode** = $35 / 100 = 0,35$

CPU-Auslastung gesamt: $0,4 + 0,35 = 0,75 = 75 \%$

Beispiel: Rate-Monotonic Scheduling

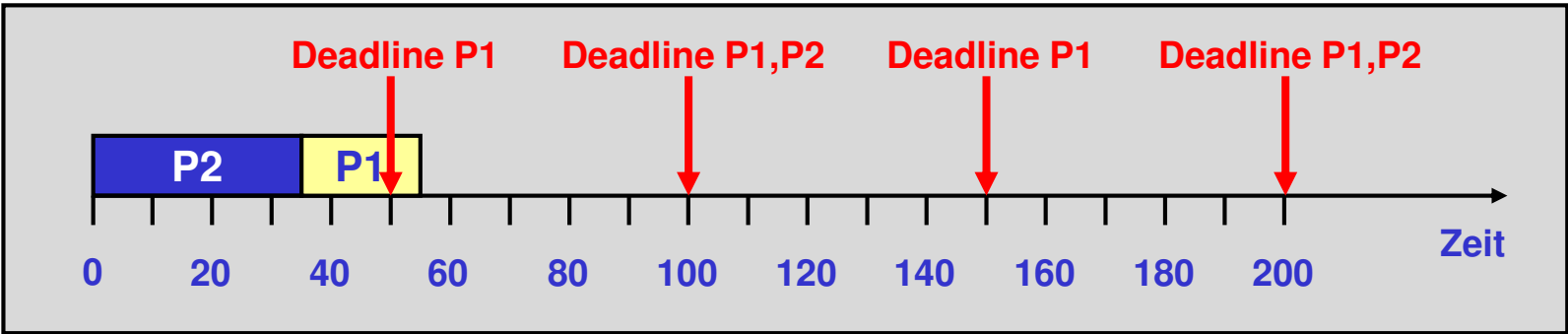
Prozess P1:

- **Dauer der Periode:** $p_1 = 50 \text{ ZE}^*$
- **Bearbeitungsdauer:** $t_1 = 20 \text{ ZE}^*$
- **Deadline:** $d_1 = p_1$

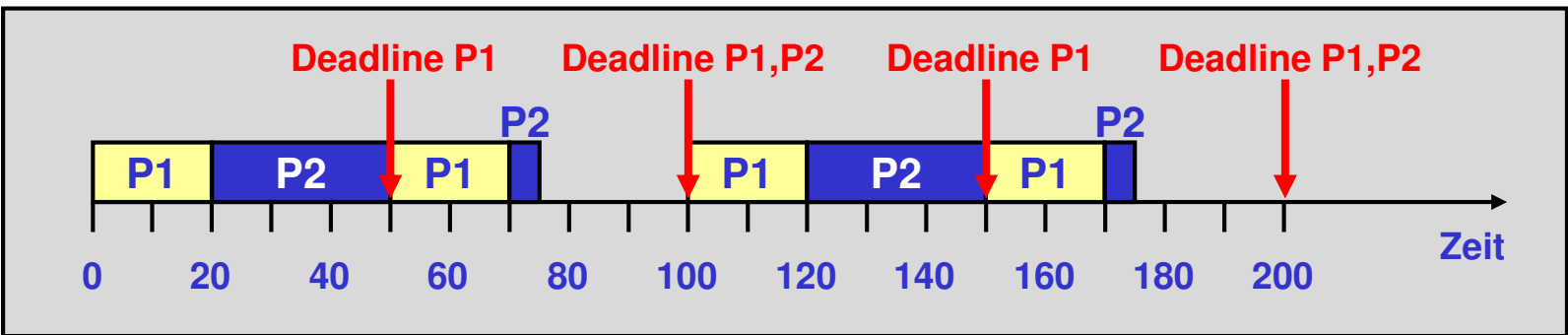
Prozess P2:

- **Dauer der Periode:** $p_2 = 100 \text{ ZE}^*$
- **Bearbeitungsdauer:** $t_2 = 35 \text{ ZE}^*$
- **Deadline:** $d_2 = p_2$

Zunächst prüfen wir, was passieren würde, wenn P2 die höhere Priorität erhielte:



Und nun – wie von Rate Monotonic Scheduling gefordert - höhere Priorität für P1:



Rate-Monotonic Scheduling als Optimal-Strategie

Wenn es überhaupt einen zulässigen Schedule mit statischen Vorab-Prioritäten gibt, dann liefert Rate-Monotonic Scheduling einen solchen.

Allerdings muss bei statischen Vorab-Prioritäten die CPU-Auslastung häufig deutlich unterhalb von 100 % liegen. Wir betrachten wieder ein Beispiel:

Prozess P1:

- **Periode:** $p_1 = 50$ ZE
- **Bearbeitung:** $t_1 = 25$ ZE
- **Deadline:** $d_1 = p_1$

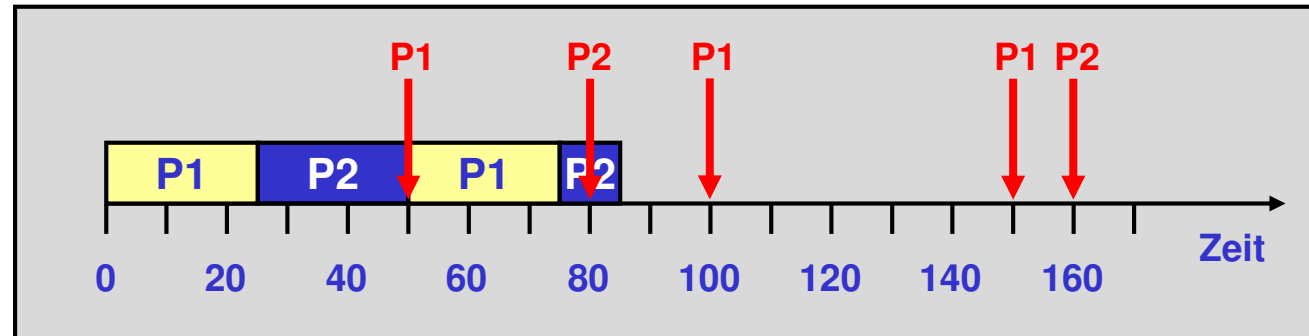
Prozess P2:

- **Periode:** $p_2 = 80$ ZE
- **Bearbeitung:** $t_2 = 35$ ZE
- **Deadline:** $d_2 = p_2$

CPU-Last durch P1: $25 / 50 = 0,5$

CPU-Last durch P2: $35 / 80 \approx 0,44$

CPU-Last gesamt: ca. 94 %



Offenbar wird zur Zeit 80 die Deadline von P2 verletzt.

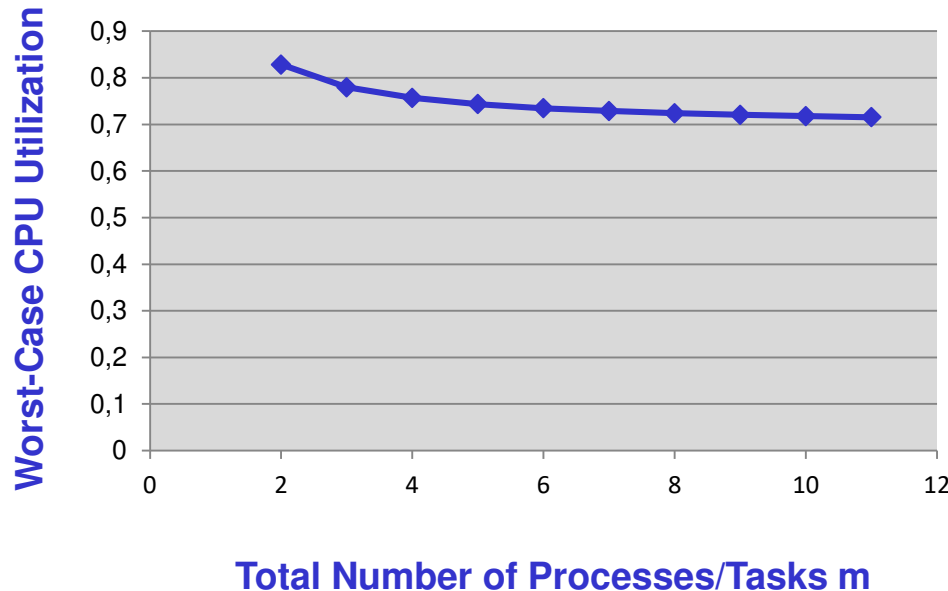
Rate-Monotonic Scheduling als Optimal-Strategie (2)

Bis zu welcher Auslastung stellt Rate-Monotonic Scheduling garantiert* einen gültigen Schedule bereit?

Die Antwort ist abhängig von der Anzahl der insgesamt beteiligten Prozesse:

THEOREM 4. *For a set of m tasks with fixed priority order, and the restriction that the ratio between any two request periods is less than 2, the least upper bound to the processor utilization factor is $U = m(2^{1/m} - 1)$.*

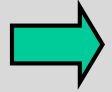
Liu und Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment"
Journal of the ACM, Vol. 20, No. 1, Jan. 1973, [verfügbar in der ACM Digital Library](#)



Für $m = 2$ ergibt sich ca. 83 % als untere Grenze der erzielbaren CPU-Auslastung.

*...sofern keine sonstigen Nebenbedingungen (wie z.B. Abhängigkeiten zwischen Prozessen) zu beachten sind.

3.7.3.2. Einplanung nach Antwortzeit (Earliest-Deadline-First, EDF)



Für jeden lauffähigen Prozess ist seine Deadline bekannt.
Zuteilung der CPU an den Prozess mit frühester Deadline.

EDF ordnet dynamisch Prioritäten zu, die sich an der jeweiligen Deadline orientieren.

Den Prozessen sind somit **keine statischen Prioritäten** zugeordnet.

Durch die zusätzliche Flexibilität lässt sich in manchen Fällen auch dann noch ein gültiger Schedule erzielen, wenn dies mit statischen Vorab-Prioritäten aufgrund hoher Auslastung nicht mehr gelingt.

Prozess P1:

- **Periode:** $p_1 = 50$ ZE
- **Bearbeitung:** $t_1 = 25$ ZE
- **Deadline:** $d_1 = p_1$

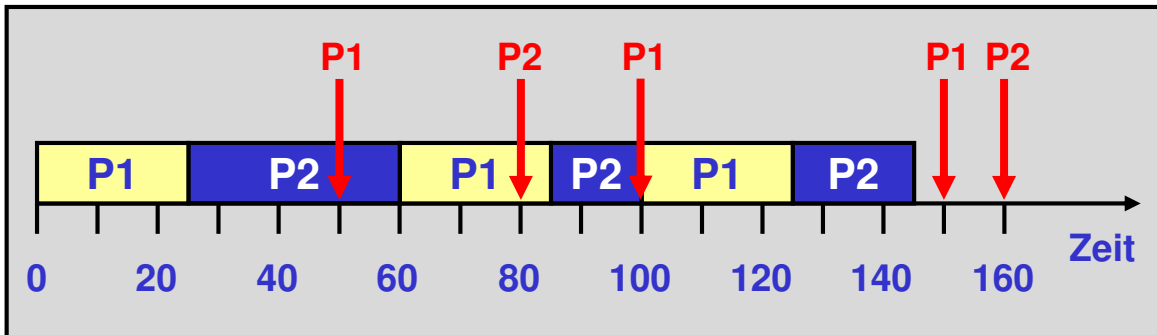
Prozess P2:

- **Periode:** $p_2 = 80$ ZE
- **Bearbeitung:** $t_2 = 35$ ZE
- **Deadline:** $d_2 = p_2$

CPU-Last durch P1: $25 / 50 = 0,5$

CPU-Last durch P2: $35 / 80 \approx 0,44$

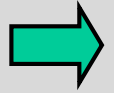
CPU-Last gesamt: ca. 94 %



EDF ist eine Optimal-Strategie:

- Theoretisch gestattet EDF die Einhaltung aller Deadlines bis zu 100 % CPU-Auslastung.
- Praktisch ist dieser Wert aufgrund des Overheads bei Kontext-Wechsel nicht erzielbar.

3.7.3.3. Einplanung nach Spielraum



Für jeden lauffähigen Prozess sind Deadline und (Rest-) Laufzeit bekannt.
Zuteilung der CPU an den Prozess mit minimalem Spielraum, d.h. minimaler Differenz zwischen Restantwortzeit und Restlaufzeit.

Auch „Einplanung nach Spielraum“ ist (wie EDF) eine **Optimalstrategie**, die (theoretisch) **100 % CPU-Auslastung unter Einhaltung aller Deadlines** ermöglicht.

Da die Einplanung nach Spielraum – im Gegensatz zu EDF – auch die Laufzeit berücksichtigt, werden aber **Fehler früher als bei EDF erkannt**:



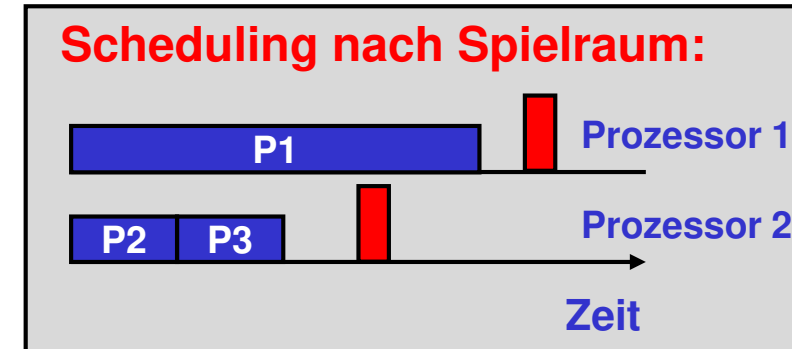
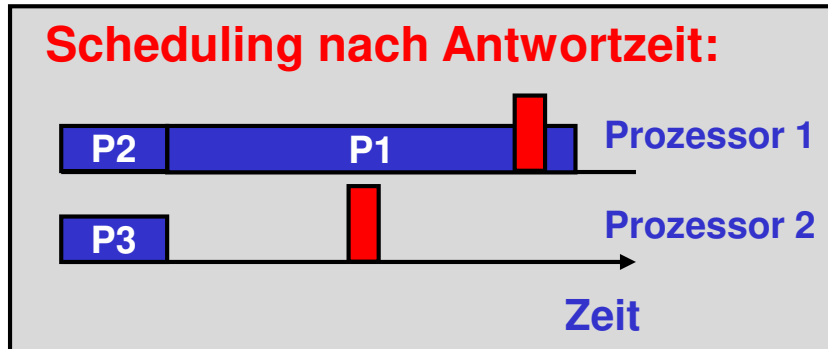
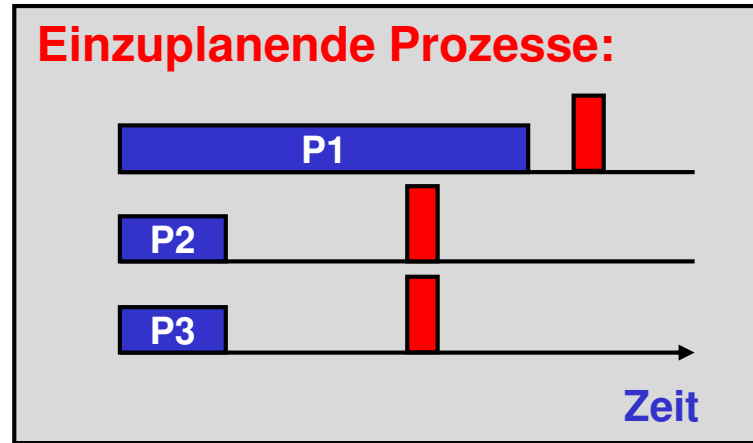
EDF erkennt Deadline-Verstöße erst, wenn sie bereits eingetreten sind !

Bei Einplanung nach Spielraum kann die Zeit für die Bearbeitung von Prozessen eingespart werden, die ohnehin nicht rechtzeitig fertig würden.

Alternativ kann das „Krisenmanagement“ durch den Nutzer bzw. das Umschalten auf alternative Software für Hochlastphasen ggf. frühzeitig einsetzen.

3.7.4. CPU-Scheduling für Mehrprozessorsysteme

Bei Mehrprozessorsystemen vermeidet die Einplanung nach Spielraum häufig Zeitverletzungen, die bei EDF eintreten. Dies wird mit folgendem Beispiel verdeutlicht:

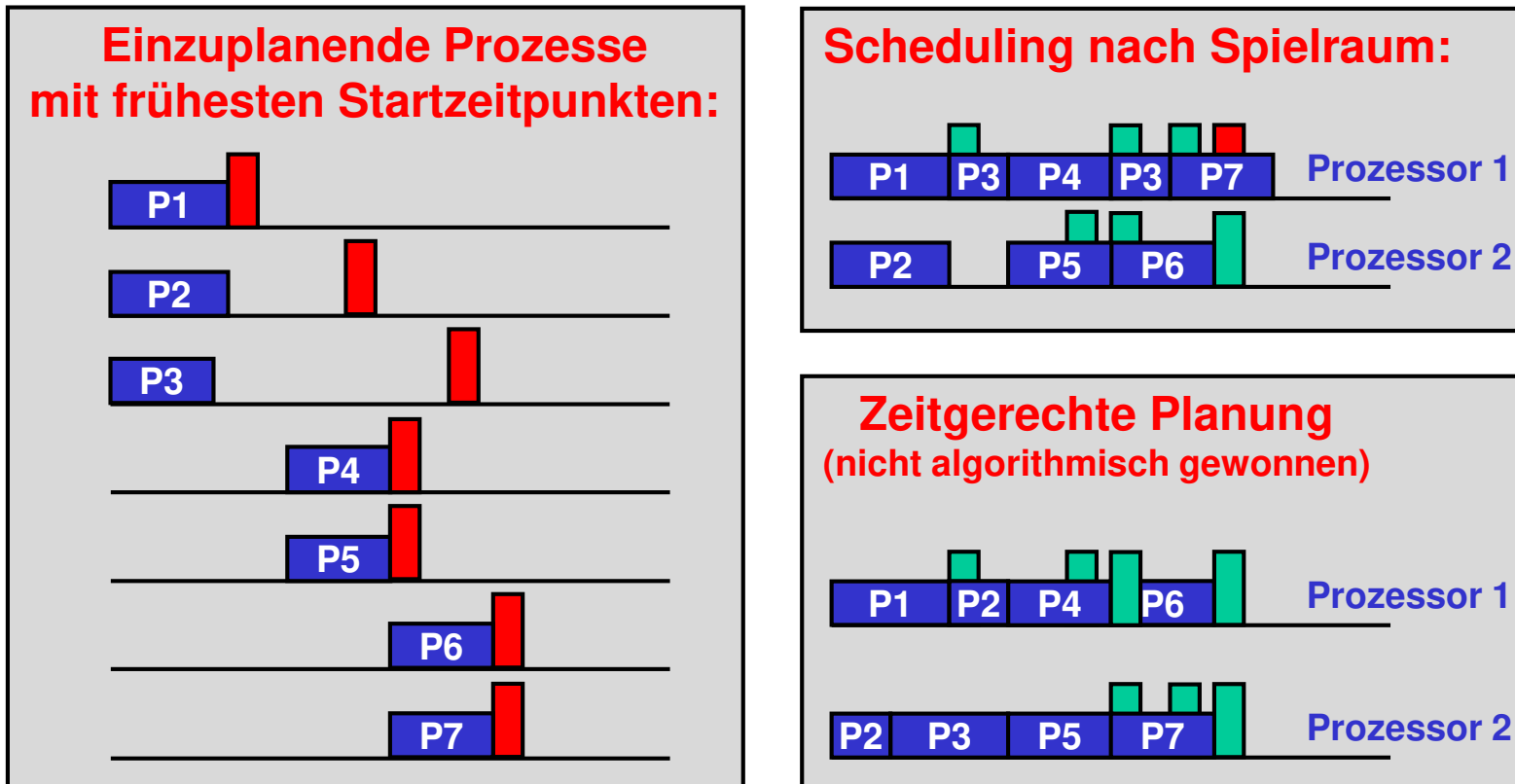


Bei Scheduling nach Antwortzeit wird die Deadline von P1 verletzt.

Scheduling nach Spielraum liefert hier dagegen eine zulässige Bearbeitung.

Unterschiedliche früheste Startzeiten

Das folgende Beispiel betrachtet Prozesse mit unterschiedlichen frühesten Startzeiten:



Bei Einplanung nach Spielraum verletzt der Prozess P7 die Deadline. Die Einplanung nach Spielraum ist hier offenbar nicht mehr optimal.

Es ist grundsätzlich nicht möglich, einen optimalen Scheduling-Algorithmus zu konstruieren, wenn die frühesten Startzeiten der Prozesse nicht a priori bekannt sind und berücksichtigt werden können.

3.7.5. Scheduling-Anomalien bei variabler CPU-Leistung

Durch **Absenkung der CPU-Leistung** kann die **Batterie-Laufzeit** bei mobilen Geräten erheblich verlängert werden.

Eigentlich würde man erwarten, dass mit zunehmender CPU-Leistung, insbesondere mit zunehmender Taktrate, keinerlei Verschlechterungen der Leistung des Gesamtsystems auftreten.

Die nachfolgenden Beispiele aus einer Fachzeitschrift des Jahres 2006 zeigen, dass bei Echtzeit-Systemen einige Anomalien zu beachten sind.

[Beispiel 1: Scheduling-Anomalie bei kritischen Abschnitten](#)

[Beispiel 2: Scheduling-Anomalie bei „Nonpreemptive Tasks“](#)

Der zugehörige Beitrag „Achieving Scalability in Real-Time Systems“ von Giorgio Buttazzo ist erschienen in

IEEE Computer, May 2006, pp. 54 - 59

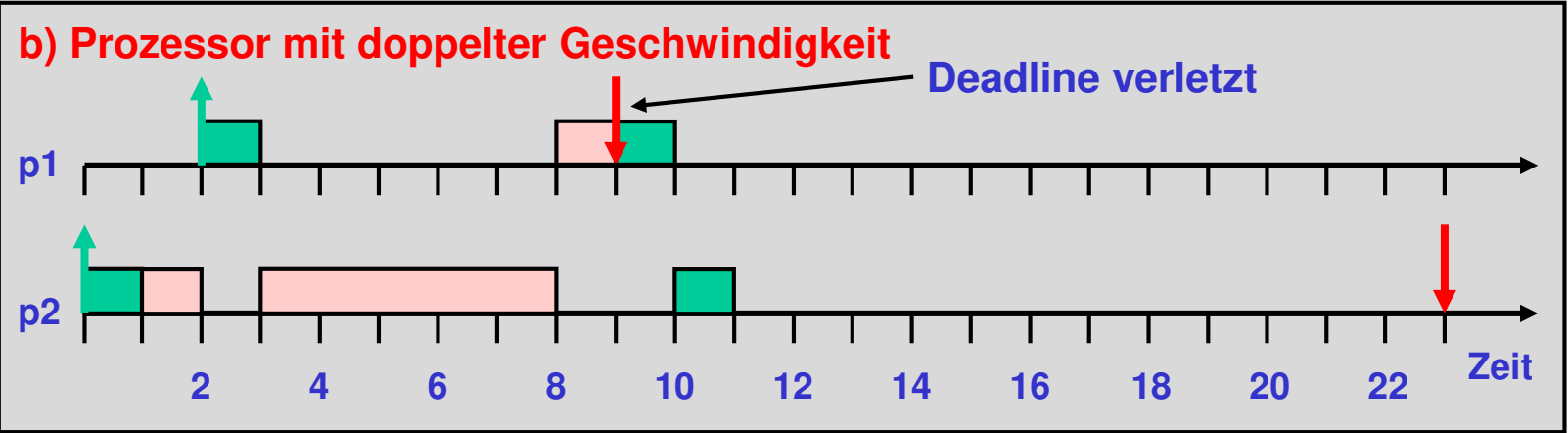
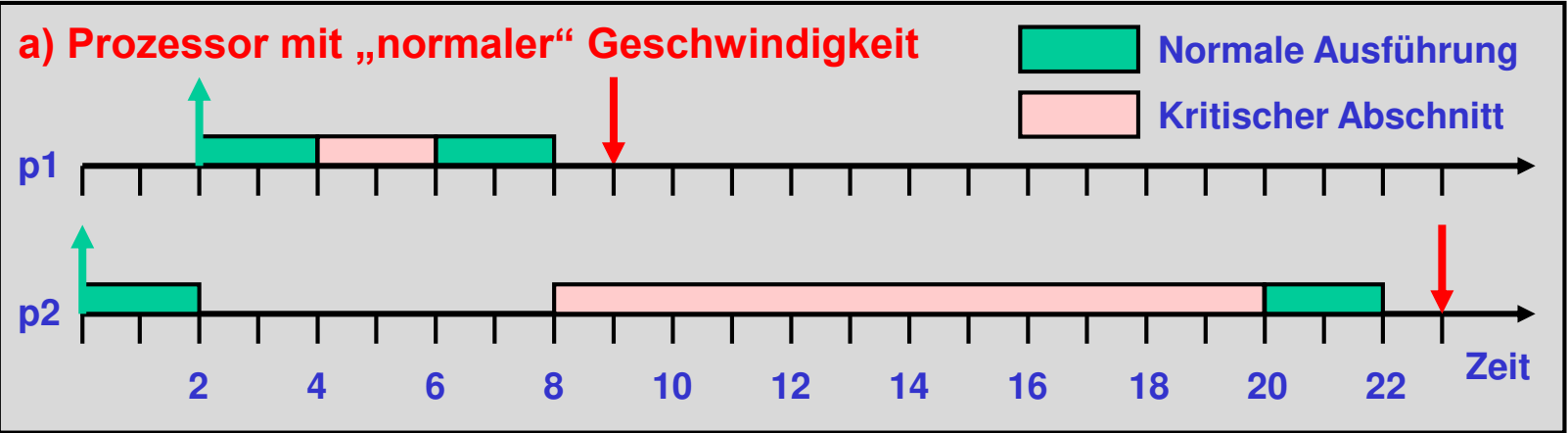
Auf dieses Paper kann in der Digital Library der IEEE Computer Society zugegriffen werden:
www.computer.org.

Beispiel 1: Anomalie bei kritischen Abschnitten (1 Prozessor)

Prozess p1: Hohe Priorität, bereit zur Zeit 2, max. Verzögerung = 7 ZE (ab Bereitstellung)
Prozess p2: Niedrige Priorität, bereit zur Zeit 0, max. Verzögerung = 23 ZE (ab Bereitstellung)

Bearbeitungsdauer:
p1: 2 + 2 (k.A.) + 2 ZE
p2: 2 + 12 (k.A.) + 2 ZE

k.A. = kritischer Abschnitt
ZE = Zeiteinheiten



Bearbeitungsdauer:

Bei doppelter Prozessor-Geschwindigkeit jeweils halb so lang.

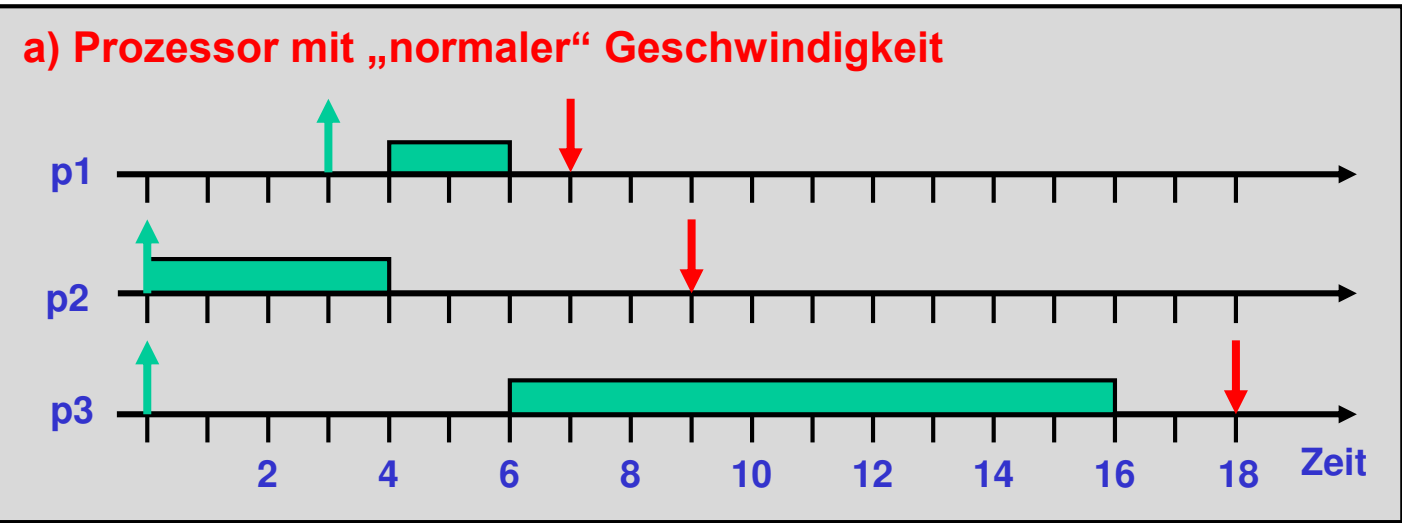
Bei schnellerem Prozessor wird **bei Prozess p1 die Deadline verletzt**.

Beispiel 2: Anomalie bei „Nonpreemptive Tasks“ (1 Prozessor)

Im zweiten Beispiel gehen wir davon aus, dass die Prozesse nicht unterbrochen werden dürfen.

Bearbeitungsdauer:
p1: 2 ZE
p2: 4 ZE
p3: 10 ZE

ZE = Zeiteinheiten



Prozess p1:

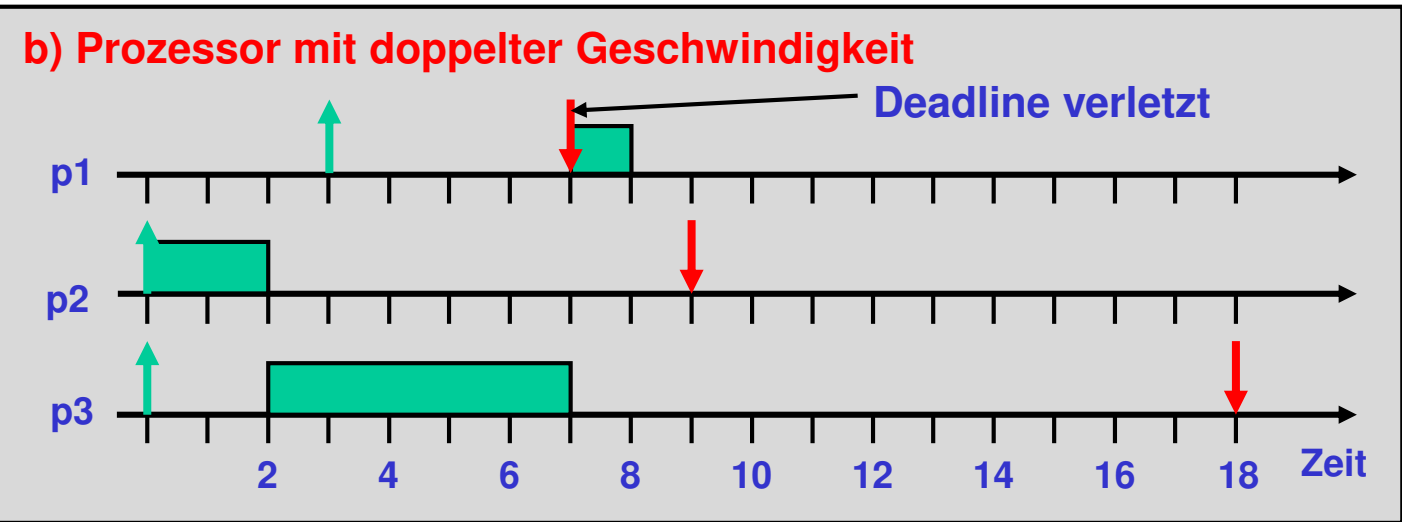
- Hohe Priorität,
- bereit zur Zeit 3,
- max. Verzögerung = 4 ZE

Prozess p2:

- mittlere Priorität,
- bereit zur Zeit 0,
- max. Verzögerung = 9 ZE

Prozess p3:

- Niedrige Priorität,
- bereit zur Zeit 0,
- max. Verzögerung = 18 ZE



↑ Prozess bereit

↓ Deadline

Bearbeitungsdauer:

Bei doppelter Prozessor-Geschwindigkeit jeweils halb so lang.

3.8. Zusammenfassung (Kapitel 3)

- Ein Prozess ist ein in Ausführung befindliches Programm.
- Threads sind „lightweight processes“ mit gemeinsam genutztem Speicher.
- Monitore sind Software-Module, die Daten und Operationen auf diesen Daten kapseln. Monitor-Operationen werden unter wechselseitigem Ausschluss durchgeführt. Sie können auch zur Verwaltung und Vergabe gleichartiger Ressourcen eingesetzt werden.
- Bei einem Deadlock blockieren sich mehrere Prozesse (oder Threads) gegenseitig.
- Die vier notwendigen Bedingungen für Deadlocks sind: „Mutual Exclusion“, „Hold and Wait“, „No Preemption“, „Circular Wait Condition“.
- Beim Scheduling muss ein Kompromiss zwischen gegensätzlichen Anforderungen gefunden werden. Häufig erweist sich „Round Robin Scheduling“ als besonders geeignet.
- Rate-Monotonic Scheduling ist eine Optimalstrategie, wenn bei periodischer Last mit statischen Vorab-Prioritäten gearbeitet werden muss.
- Zeitgerechtes Scheduling lässt bei statischen Vorab-Prioritäten zuweilen nur eine geringe Gesamtlast zu.
- Mit Earliest-Deadline-First kann dagegen (theoretisch) eine CPU-Auslastung von 100 % erreicht werden.
- Einplanung nach Spielraum liefert bei Einprozessorsystemen frühzeitige Warnungen vor Deadline-Verstößen und ist bei Mehrprozessorsystemen dem Verfahren Earliest-Deadline-First überlegen.
- Bei Echtzeit-Scheduling mit variabler CPU-Leistung sind Anomalien zu beachten.