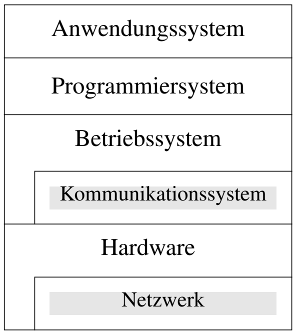
Einführung

1. Moderne Rechnersysteme



1.1 Programmiersysteme

✎ **Klassische Programmiersysteme**

- Ein Rechner führt genau ein Programm aus.
- Ein Programm wird auf genau einem Rechner ausgeführt.

✎ **Moderne Programmiersysteme**

- Nebenläufigkeit
- Verteiltheit

1.2 Betriebssysteme

✎ 70er Jahre ein Rechner - viele Benutzer

- Stapelbetrieb
- Mehrfach-Stapelbetrieb

✎ 80er Jahre: ein Rechner - ein Benutzer

- DOS
- Multi-Prozessor-Betriebssysteme
- Timesharing-Betriebssysteme

✎ 90er Jahre: viele Rechner - ein Benutzer

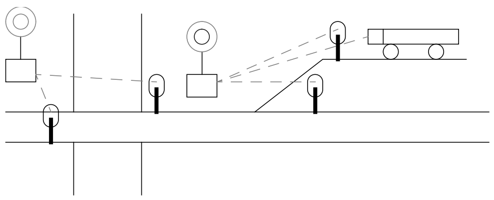
- Netzwerkbetriebssysteme
- Verteilte Betriebssysteme

1.3 Anwendungen moderner Programmiersysteme

✎ Moderne Datenbanksysteme

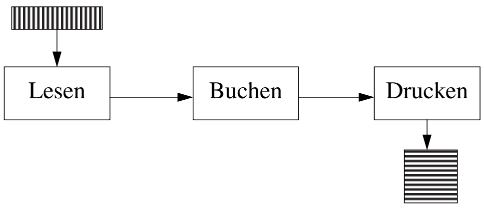
✎ Prozeßdatenverarbeitung

⇨ Verteiltes Bahn-Kontrolle-System



✎ Verwaltungs- und Büroautomatisierung

⇨ Ein Buchungssystem

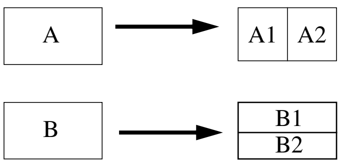


2 Grundkonzepte

2.1 Prozesse

✎ **Aktivitäten** (*activities*)

- Aktivitäten sind die Einzelschritte, durch deren Ablauf die Leistung eines Programms erbracht wird.
- Abstraktion beschreibt wie man Aktivitäten voneinander abgrenzt.
- Die Ausführung einer Aktivität muß nicht unteilbar (*indivisible*) sein, sie kann in mehreren Schritten erfolgen.
- Zusammengesetzte Aktivität kann sequentielle order nebenläufige Teilaktivitäten enthalten.



✎ **Prozesse**

Ein Prozeß ist eine sequentielle Folge von Aktivitäten, durch die eine in sich abgeschlossene Aufgabe bearbeitet wird. In Abgrenzung zu anderen Prozeßbegriffen nennt man einen solchen Prozeß auch einen Rechenprozeß (*computation process*).

✎ Sequentielle Programme

• Nach jedem Schritt der Ausführung eines <u>sequentiellen Programms</u> steht eindeutig fest, welches der nächste sein wird.

➪ Sequentielle Variablenzuweisungen

```
v = x;    w= x;    x = y;
```

• Eigenschaften sequentieller Programme

<u>Sequentialität</u>: Alle Schritte eines Programms werden nacheinander ausgeführt.
<u>Determinismus</u>: Der Ablauf eines Programms erfolgt immer in gleicher Reihenfolge.
<u>Determiniertheit</u>: Bei gleichen Eingaben produziert ein Programm gleiche Ausgaben.

✎ Nebenläufige Programme

• Ein Programm ist dann <u>nebenläufig</u> (*concurrent*) wenn die Aktivitäten parallel von mehreren Prozessoren ausgeführt werden können (<u>parallele Ausführung</u>),
und/oder
die Aktivitäten in einer beliebigen Folge sequentiell von einem Prozessor ausgeführt werden können (<u>sequentielle Ausführung</u>).

• Bei Ausführung eines nebenläufigen Programms von einem Prozessor entsteht eine gemischte Sequenz von Aktivitäten aus Teilen nebenläufiger Prozesse. Diese Sequenz wird als <u>Verzahnung</u> (*interleaving*) genannt.

• verzicht auf Sequentialität

➪

```
conc v = x || w =  x end conc;
x = y;
```

• verzicht auf Determinismus

➪

```
select     a ≥ b → max = a
        []   a ≤ b → max = b
end select;
```

• verzicht auf Determiniertheit

➪

```
conc v = x || w =  x || x = y end conc;
```

➪

```
select   a ≥ b → x = 1
      []   a ≤ b → x = 0
end select;
```

➪ Frage: ist das folgende Programm sequentiell, deterministisch, und determiniert?

```
conc x = x + 1 || x = x + 1 end conc;
```

- + 1 ist unteilbar

- + 1 ist teilbar

✎ Probleme nebenläufiger Programme

• enger Zusammenhang von Nebenläufigkeit, Nichtdeterminismus, Nichtdeterminiertheit

• Nichtreproduzierbarkeit (schwer zu testen)

• komplizierte Zustandsübergänge bei der Programmausführung (schwer für den Nachweis der Korrektheit)

➪

```
{x=i, y = j} v := x;  w :=  x;  x := y; {v=w=i, x=y=j}
```

```
{x=i, y = j}
conc v := x || w :=  x end conc;
x = y;
{v=w=i, x=y=j}
```

```
{x=i, y = j}
conc v := x || w :=  x || x := y end conc;
{v=w=i, x=y=j}
```

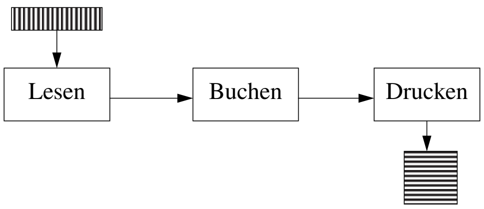

2.2 Synchronisation und Kommunikation

✎ Kooperierende und konkurrierende Prozesse

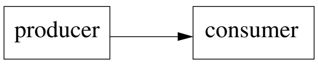
➪ A1 muß vor A2 und vor A3 ausgeführt werden.

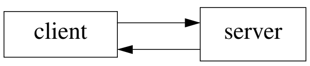
Lösungen:
1. A1 und A2 bilden einen Prozeß, A3 bildet einen anderen Prozeß.
2. A1 und A3 bilden einen Prozeß, A2 bildet einen anderen Prozeß.
3. A1, A2 und A3 bilden je einen eigenen Prozeß.

➪ Ein modernes Buchungssystem



➪ - Produzent/Konsumenten (*producer/consumer*) System
- Auftraggeber/Auftragnehmer (*client/server*) System





➪ Konkurrenzsituation im Eisenbahnverkehr

✎ **Synchronisation und Kommunikation**

• Die Koordination der Kooperation und Konkurrenz zwischen Prozessen wird Synchronisation genannt.

➪ Im Buchungssystem

➪ Beim Eisenbahnverkehr

• Der Datenaustausch über Prozeßgrenzen hinweg wird als Kommunikation genannt.

➪ Im Client/Server-System

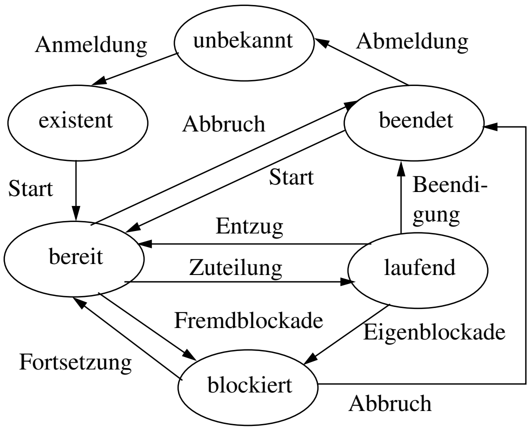
• Kommunikation und Synchronisation bedingen einander gegenseitig: Zum Abstimmen zweier Aktivitäten können Prozesse Informationen austauschen, und zum Zweck des Austauschs müssen sie die Aktivitäten der Informationsabgabe und -aufnahme aufeinander abstimmen.

2.3 Prozeßverwaltung

✎ **Zustand eines Prozesses**

• Ein Prozeß ist existent: wenn er der Prozeßverwaltung bekannt ist, jedoch noch keinen anderen Zustand durchlaufen hat
• Ein Prozeß ist bereit: wenn er den Prozessor nutzen kann, sobald ihm dieser zugeteilt wird oder er gestartet ist
• Ein Prozeß ist laufend: wenn er sich im Besitz des Prozesors befindet
• Ein Prozeß ist blockiert: wenn ein Prozeß auf das Eintreffen einer Bedingung oder ein Ereignis wartet
• Ein Prozeß ist beendet: wenn ein Prozeß seine letzte Anweisung ausgeführt hat, oder

✎ **Zustandsübergänge von Prozessen**



2.4 Prozesse in Java

✎ **Java-Threads**

• java.lang.Thread

➪
```
public class OurThread extends Thread {
    ...
    public OurThread() {...}
    public void run() {...}
    ...
}
OurThread ot = new OurThread();
ot.start();
```

• java.lang.Runnable

➪
```
public class OurRunnable implements Runnable {
    ...
    public OurRunnable() {...}
    public void run() {...}
    ...
}
OurRunnable or = new OurRunnable();
Thread th = new Thread(or);
th.start();
```


✎ **Zustandsübergänge von Threads**

• *start*: ein neuer Thread wird generiert und die *run*-Methode wird aufgerufen

• *interrupt*: ein laufender Thread wird terminiert

• *sleep*: ein laufender Thread wird für spezifizierte Zeit in Bereitzustand versetzt

• *wait*: ein laufender Thread wird solange in Bereitzustand versetzt, bis eine Bedingung auftritt oder die Wartezeit abgelaufen ist

• *notify* und *notifyAll*: Ein wartender Thread oder alle wartende Threads wird oder werden informiert, daß die Bedingung aufgetreten ist

2.5 Unterbrechungen

Ein Prozeß wird unterbrochen (*interrupted*), wenn dessen Aktivitäten nicht auf die normale Weise fortgesetzt werden, weil er auf ein bestimmtes Ereignis warten muß.

✎ **Unterbrechungsarten**

• Ein externes Ereignis, oft auch as echte Unterbrechung bezeichnet, tritt zu einem unvorhersehbaren Zeitpunk asynchron zur Programmausführung auf.

• Ein internes Ereignis oder eine Ausnahmebedingung (*exception*) tritt synchron zur Programmausführung auf.

➩ Java

Asynchrone Ausnahmen sind selten in Java. Sie können nur resultieren aus:

1. einem Aufruf der stop-Methoden der Klasse Thread oder ThreadGroup
2. einem internen Fehler in der virtuellen Java-Maschine.

Synchrone Ausnahmen können resultieren aus:
1. eine anormale Ausführungsbedingung wurde von einer virtuellen Java-Maschine synchron entdeckt
2. eine throw-Anweisung wurde im Java-Code ausgeführt.

```
try {
   i = Integer.parserInt(arg[0]);
}
if (i == 0)
   {throw new
      ArithmeticException("integer division by zero");
   }
else {return(a/i);}
```

✎ Unterbrechungsbehandlung

➩ Java

public class Throwable extends Object
implements Serializable

public class Exception extends Throwable
Diese Klasse ist der Wurzel der Ausnahmehierarchie in Java. Eine Ausnahme signalisiert eine Ausnahmesituation, die speziell behandelt werden muß.

public class Error extends Throwable
Diese Klasse ist der Wurzel der Fehlerhierarchie in Java. Ein Fehler muß nicht speziell behandelt werden und beendet die Ausführung des Programms.

```
class MyException extends Exception {
 public MyException() { super(); }
 public MyException(String s) { super(s); }
}

public int c(int i) throws MyException {
  if (i==0) { throw new MyException("integer 0"); }
   else {return(i);}
}

public int b(int i, int a) {
   int j=-1;
   try { j = c(i); }
   catch(MyException e)
     {System.out.println(e.getMessage());}
   return(a/j);
}
```