LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

mobile and distributed systems group

Vorlesung Betriebssysteme

am 25. Oktober 2017

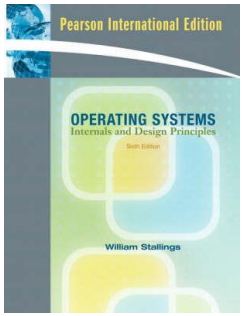
Prof. Dr. Claudia Linnhoff-Popien, Carsten Hahn

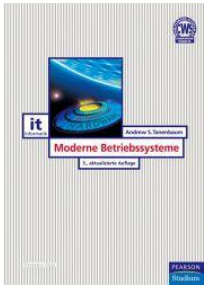
Gliederung

- Einführung
  - Was ist ein Betriebssystem?
- Ebene der physischen Geräte
  - Boolesche Algebra
  - Gatter
- Ebene der Mikroarchitektur
  - Der Befehlszyklus
  - Interrupts
- Ebene der Maschinensprache
  - Verweis auf SPIM
- Ebene des Betriebssystems
  - Prozesse und Threads
  - Threads in Java

Literatur

Dieser Foliensatz wurde auf Basis folgender Literatur erstellt:



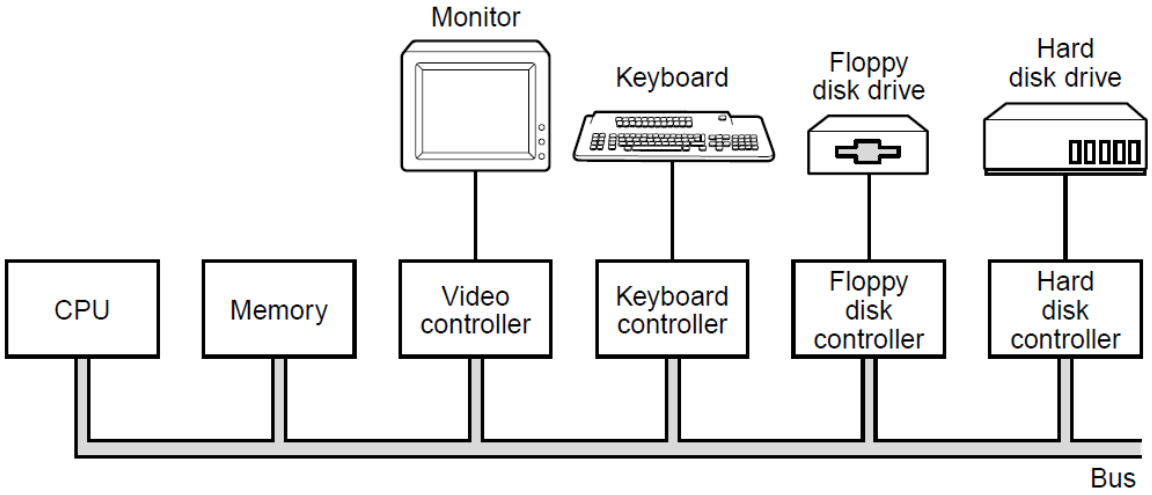


- Claudia Linnhoff-Popien: Skript zur Vorlesung Betriebssysteme im WS17/18
- William Stallings: Operating Systems – Internals and Design Principles, 7. Auflage, Pearson International Edition
- Andrew S. Tanenbaum: Moderne Betriebssysteme, 3. überarbeitete Auflage, Pearson Studium
- Erich Ehses, Lutz Köhler, Petra Riemer, Horst Stenzel, Frank Victor: Betriebssysteme – Ein Lehrbuch mit Übungen zur Systemprogrammierung in Unix/Linux
- Andrew S. Tanenbaum: Computerarchitektur – Strukturen – Konzepte - Grundlagen, 5. Auflage, Person Studium
- David A. Patterson, John L. Hennessy: Rechnerorganisation und – entwuf – Die Hardware/Software-Schnittstelle, 3. Auflage, Spektrum Verlag

EINFÜHRUNG

Geräte in Computern

- Anbindung des Prozessors an die E/A-Geräte:

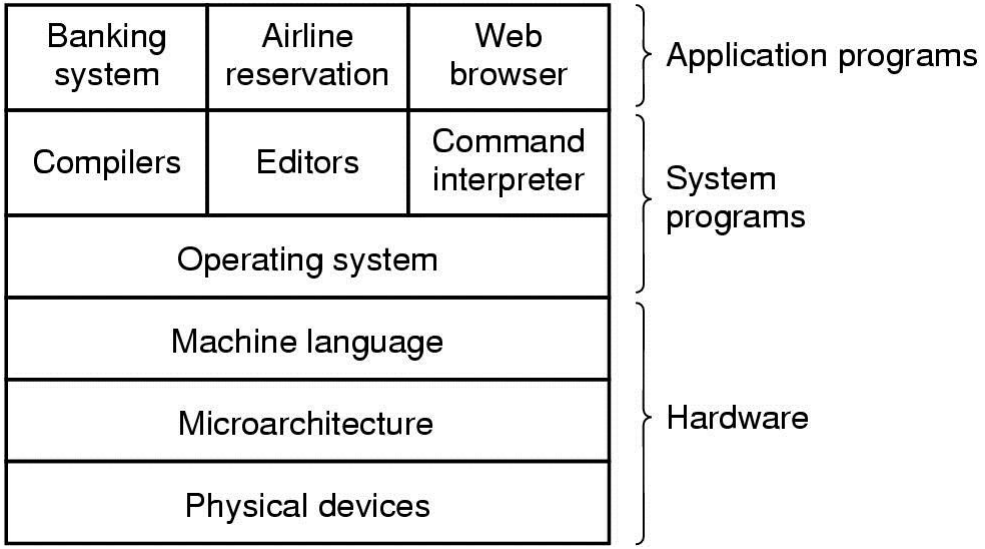


- Die **CPU**, der **Hauptspeicher** und alle **E/A-Module** sind mit dem **Systembus** verbunden
- Der Systembus kann weiter unterteilt werden in:
  - Datenbus (Übertragung von Daten)
  - Adressbus (Festlegung der Quell- bzw. Zieladresse)
  - Steuerbus (Festlegung der Operation Lesen, Schreiben, …)
- Die **E/A-Module** bestehen aus Controller und der eigentlichen Hardware (z.B. Laser, rotierende Platten usw.)
- Jedes Gerät ist über einen Controller (E/A-Modul) mit dem Systembus verbunden
- Die Controller nehmen Steuerungsbefehle von der CPU entgegen

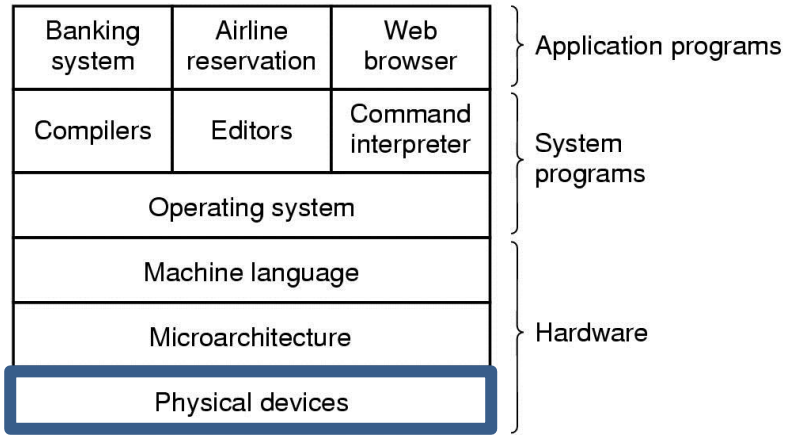
Einführung in Betriebssysteme

- Erstellung von Programmen, welche Geräte benutzen ist sehr komplex

→ Betriebssystem als zusätzliche Softwareschicht. Einfache Schnittstelle zur Hardware.



- Physische Geräte: Integrierte Schaltungen, Drähte, Stromversorgung, Bildschirm u.v.m…
- Mikroarchitektur: Funktionale Einheiten – z.B. Prozessor (CPU)
- Maschinensprache: Hardware-Anweisungen und Assembler
- Betriebssystem: Komplexität verstecken



PHYSISCHE GERÄTE

Boolesche Algebra (1)

- George Boole: englischer Mathematiker (1815 – 1864)
- Beschäftigung mit formaler Sicht heutiger digitaler Strukturen

- **Herangehensweise:**
  - Σ_2 = { 0, 1} (Alphabet) wird als B bezeichnet
  - Man betrachte die Variablen a, b $\in$ B und definiert drei Operationen
- Der **OR-Operator** (geschrieben + oder $\vee$) – auch logische Summe bezeichnet:

a	b	a OR b
0	0	0
0	1	1
1	0	1
1	1	1

Boolesche Algebra (2)

- Der AND-Operator (geschrieben * oder $\wedge$) – auch logisches Produkt bezeichnet:

a	b	a AND b
0	0	0
0	1	0
1	0	0
1	1	1

- Der NOT-Operator (geschrieben als $\bar{a}$ oder $\neg$ a) – auch Invertierung genannt:

a	NOT a
0	1
1	0

- → Boolesche Algebra ergibt sich als (B, AND, OR, NOT)

Boolesche Algebra (3)

- Gesetze zur Manipulation logischer Gleichungen:

Kommutativgesetz: $a \vee b = b \vee a$ und $a \wedge b = b \wedge a$

Assoziativgesetz: $(a \vee b) \vee c = a \vee (b \vee c)$ und $(a \wedge b) \wedge c = a \wedge (b \wedge c)$

Distributivgesetz: $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$ und $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$

Identitätsgesetz: $a \vee 0 = a$ und $a \wedge 1 = a$

Null– und Eins–Gesetz: $a \wedge 0 = 0$ und $a \vee 1 = 1$

Komplementärgesetz: $a \vee \overline{a} = 1$ und $a \wedge \overline{a} = 0$

Verschmelzungsgesetz: $(a \vee b) \wedge a = a$ und $(a \wedge b) \vee a = a$

de Morgansche Regeln: $\overline{a \vee b} = \overline{a} \wedge \overline{b}$ und $\overline{a \wedge b} = \overline{a} \vee \overline{b}$

Boolesche Funktionen

- Eine Funktion f: $B^n \rightarrow B$ heißt **n-stellige Boolesche Funktion**
- Im Falle n = 1 ergeben sich 4 mögliche einstellige Funktionen f(x) =0; f(x)=1; f(x)=x und f(x) = ¬ x:

x	0	x	$\overline{x}$	1
0	0	0	1	1
1	0	1	0	1

- Im Falle n = 2 ergeben sich 16 mögliche zweistellige Boolesche Funktionen:

x	y	0	AND	$x\overline{y}$	x	$\overline{x}y$	y	$\leftrightarrow$	OR	NOR	$=$	$\overline{y}$	$\overline{\overline{x}y}$	$\overline{x}$	$\overline{x\overline{y}}$	NAND	1
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
		f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}

- Es gibt 2^{2^n} n-stellige Boolesche Funktionen

Die Termdarstellung Boolescher Funktionen

- Die Funktion NAND in Termdarstellung…

x	y	0	AND	$x\bar{y}$	x	$\bar{x}y$	y	$\leftrightarrow$	OR	NOR	=	$\bar{y}$	$\overline{\bar{x}y}$	$\bar{x}$	$\overline{x\bar{y}}$	NAND	1
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
		f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}

- … lautet:

Distributiv Komplementär

$$\bar{x}\cdot\bar{y} + \bar{x}\cdot y + x\cdot\bar{y} = \bar{x}\cdot(y+\bar{y}) + x\cdot\bar{y} = \bar{x}\cdot 1 + x\cdot\bar{y} = \bar{x} + x\cdot\bar{y} = (\bar{x}+x)\cdot(\bar{x}+\bar{y}) = (\bar{x}+\bar{y}) = \overline{x\cdot y}$$

Null- und Eins Distributiv Komplementär Null- und Eins

- Die Darstellung für = (Äquivalenz) lautet:

$$x\cdot y + \bar{x}\cdot\bar{y}$$

Gatter und Logische Bausteine (1)

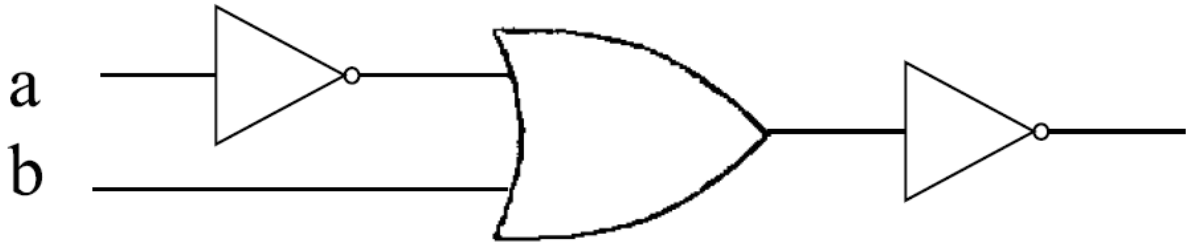
- Implementierung der Booleschen Funktionen nun als Gatter
- Schaltungen aus Verknüpfung einfachster Elemente aufbauen
- Gatter: Realisieren Funktionen zweiwertiger Signale
  - 0 – 1 Volt: Binäre 0
  - 1 – 5 Volt: Binäre 1

- Beispiele für Gatter in der IEEE-Darstellung:

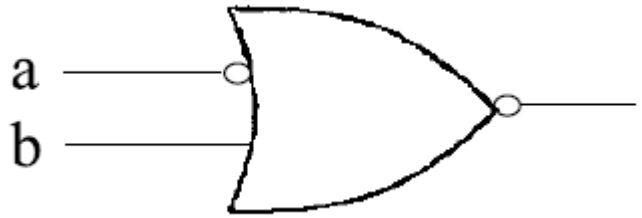
AND OR NOT

Gatter und Logische Bausteine (2)

- Darstellung der Funktion $\overline{\overline{a} + b}$:

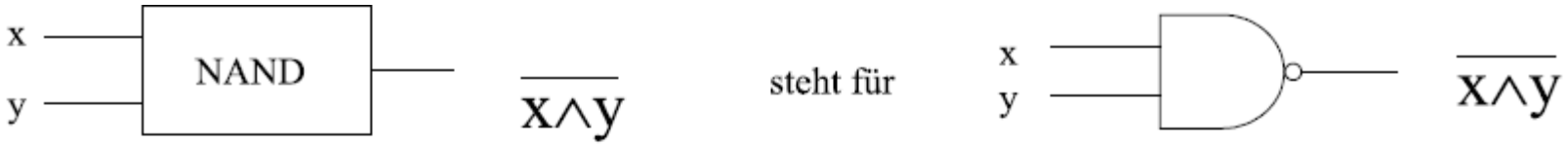


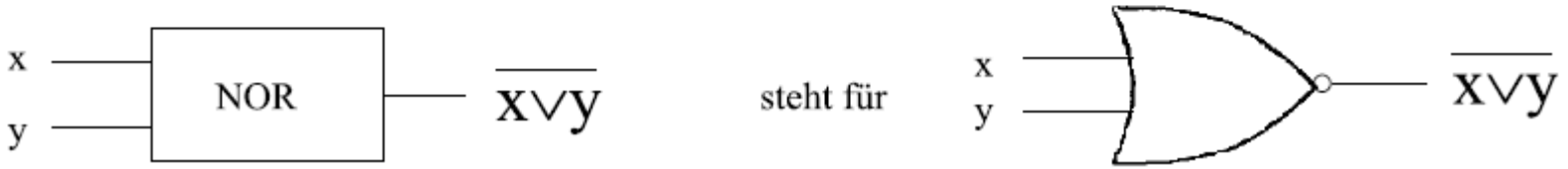
- Vereinfachte Darstellung:



Gatter und Logische Bausteine (3)

- Betrachte NAND und NOR:

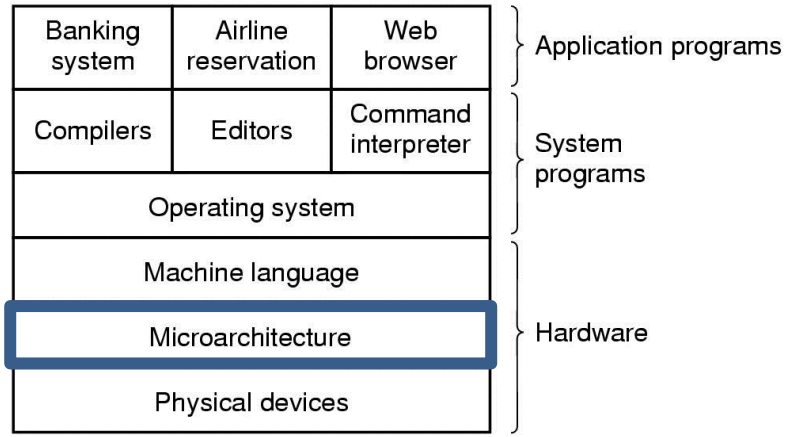




- Eine Funktion F: $B^n \rightarrow B^m$ mit m,n $\in$ N und n, m $\geq$ 1 heißt **Schaltfunktion**

→ Schaltfunktionen werden durch m Boolesche Funktionen dargestellt:

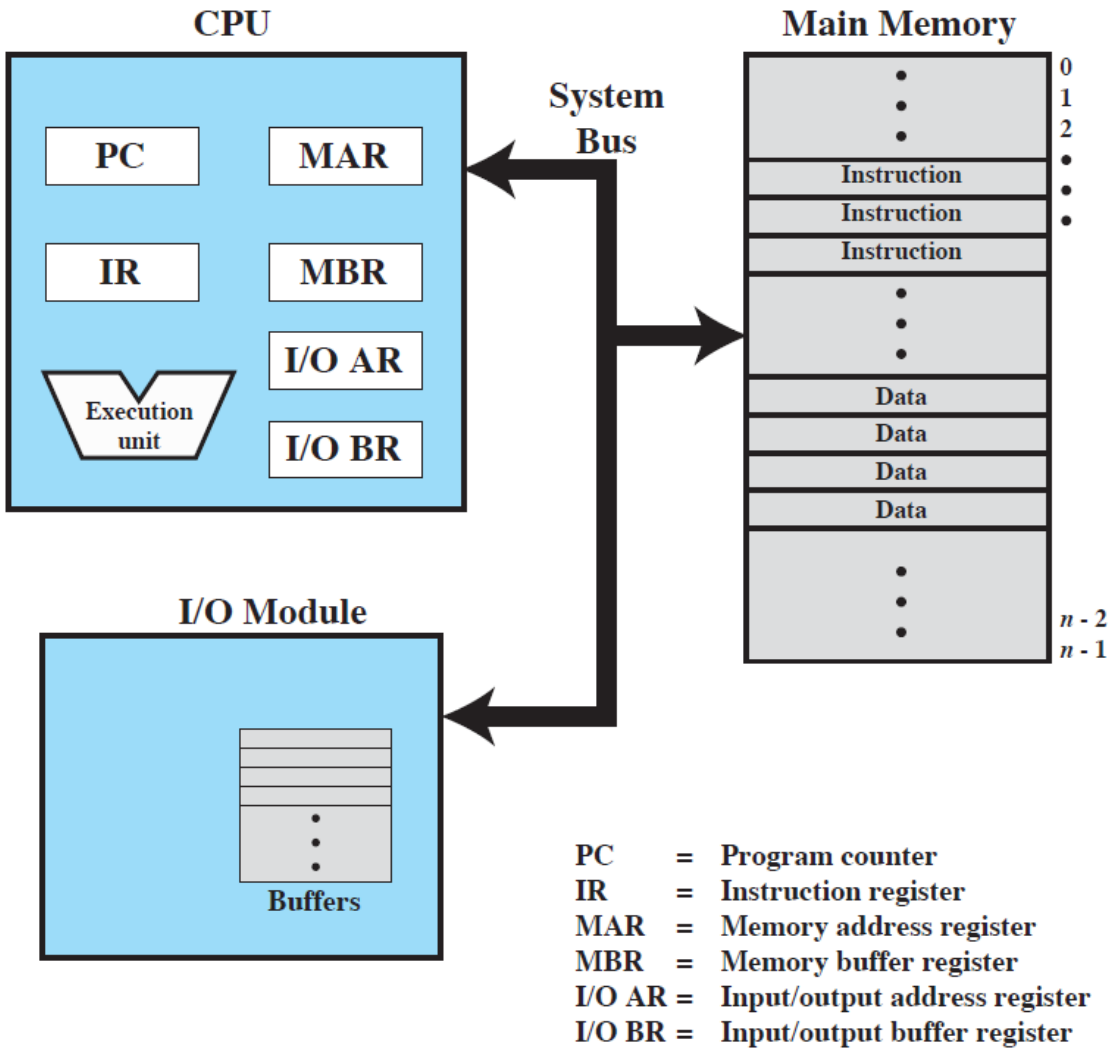
$$F(x_1, ..., x_n) = (f_1(x_1, ..., x_n), f_2(x_1, ..., x_n), ..., f_m(x_1, ..., x_n))$$



DIE MIKROARCHITEKTUR VON COMPUTERSYSTEMEN

Basiselemente

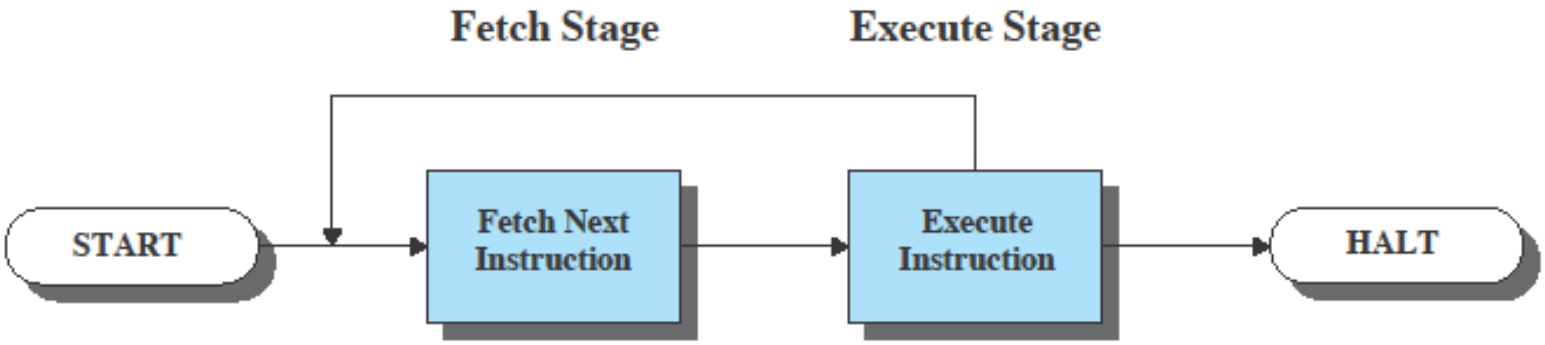
- **AR (address register)** beschreiben die Zieladresse von zu schreibenden/lesenden Daten
- **BR (buffer register)** stellen die Daten bereit oder dienen zum Einlesen von Daten.
- **PC (program counter)** beschreibt Adresse des nächsten Befehls
- **IR (instruction register)**
- **Hauptspeicher:** Satz von fortlaufend nummerierten Speicherzellen
- Inhalt des Hauptspeichers kann als **Befehl** oder **Daten** interpretiert werden



Befehlsausführung (1)

- Befehlsausführung in zwei Schritten:
  - Schritt 1 (Befehlsabruf): Prozessor liest Befehl aus Hauptspeicher
  - Schritt 2 (Befehlsausführung)

- Der Abrufzyklus bzw. Ausführungszyklus der CPU:



→ Maßgeblich ist der Program Counter (PC)

Befehlsausführung (2)

- Program Counter (PC) enthält Adresse des nächsten Befehls
- → Abruf dieses Befehls im ersten Schritt
- → Erhöhung des PC nach Befehlsabruf (falls nicht anders vorgesehen)

- Beispiel: Speicherwort belegt 16 Bit. Befehlsformat:

0001 = Akk. aus dem Speicher laden
0010 = Akk. im Speicher ablegen
0101 = Aus dem Speicher dem Akk. hinzufügen

- 2^4 = 16 mögliche Befehle
  - Prozessor ←→ Speicher: Übertragen von Daten
  - Prozessor ←→ E/A: Daten mit Peripheriegerät austauschen
  - Datenverarbeitung: Arithmetische bzw. logische Operationen durchführen
  - Steuerung: Änderung der Ausführungsreihenfolge (PC)
- 2^{12} = 4096 (4K) Speicherwörter direkt adressierbar

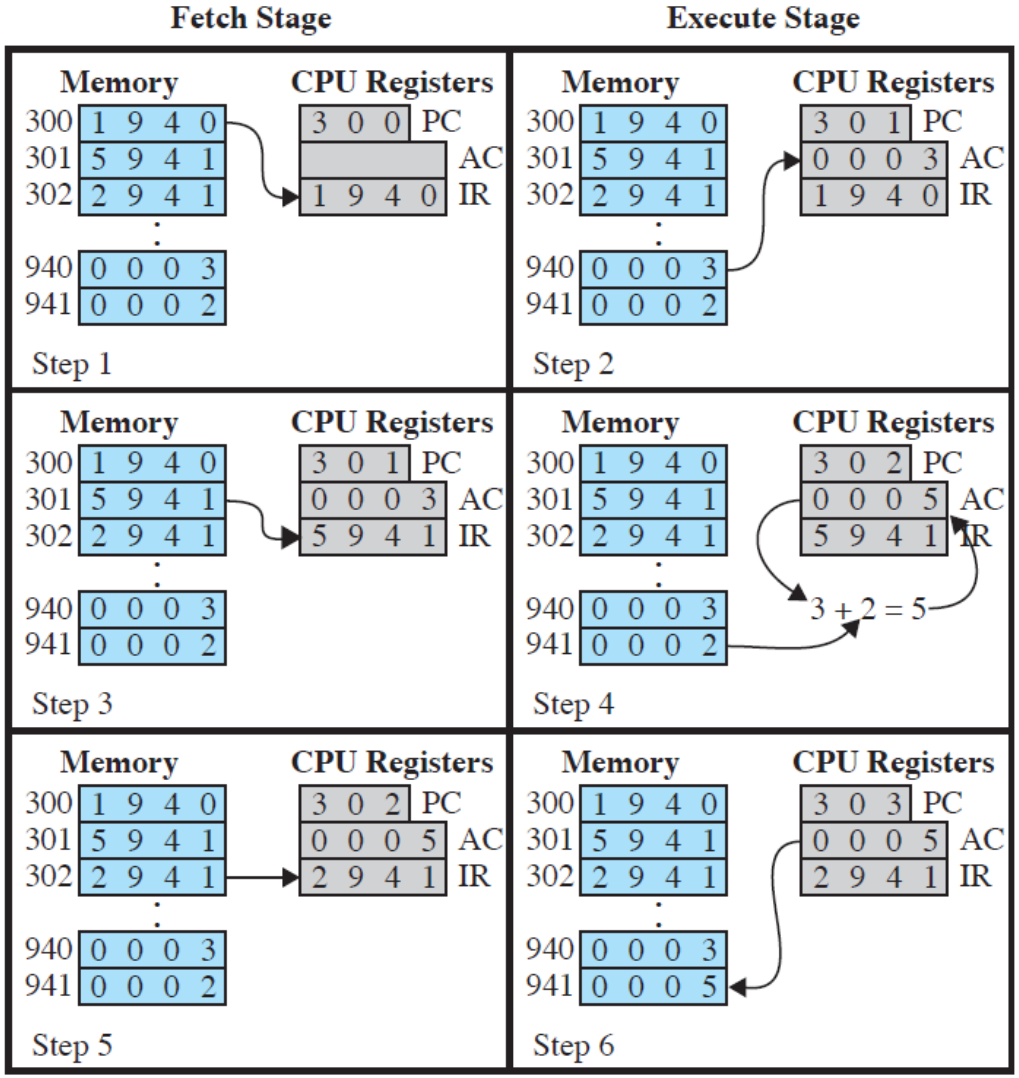
Befehlsausführung (3)

Addiere Inhalt des Speicherworts 940 zu Speicherwort 941. Speichere Ergebnis in Speicherwort 941.

1. Lade ersten Befehl von 300, erhöhe PC
2. Lade Akk. mit Adr. 940
3. Lade Befehl von 301, erhöhe PC
4. Addiere Inhalt von Akk. und 941
5. Lade Befehl von 302, erhöhe PC
6. Speichere Akk. in 941

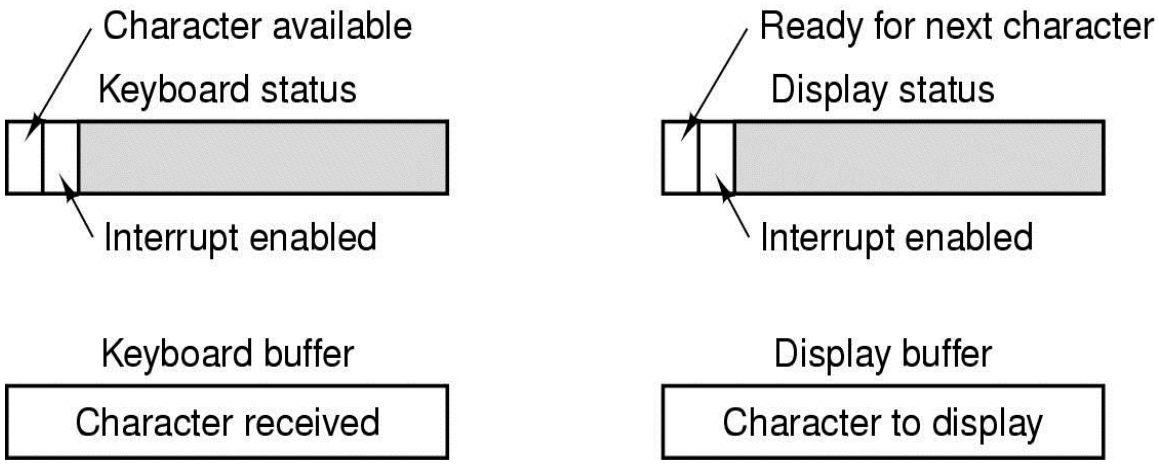
→ Ähnliche Befehlssequenz bei E/A Zugriff möglich

→ Alternativ: DMA (direct memory access). E/A-Modul greift eigenständig auf Speicher zu



Der Zugriff auf E/A-Geräte

- Beispiel: Ein Terminal, bestehend aus Tastatur und Display:



- Am Keyboard eingegebene Zeichen sollen auf Display angezeigt werden

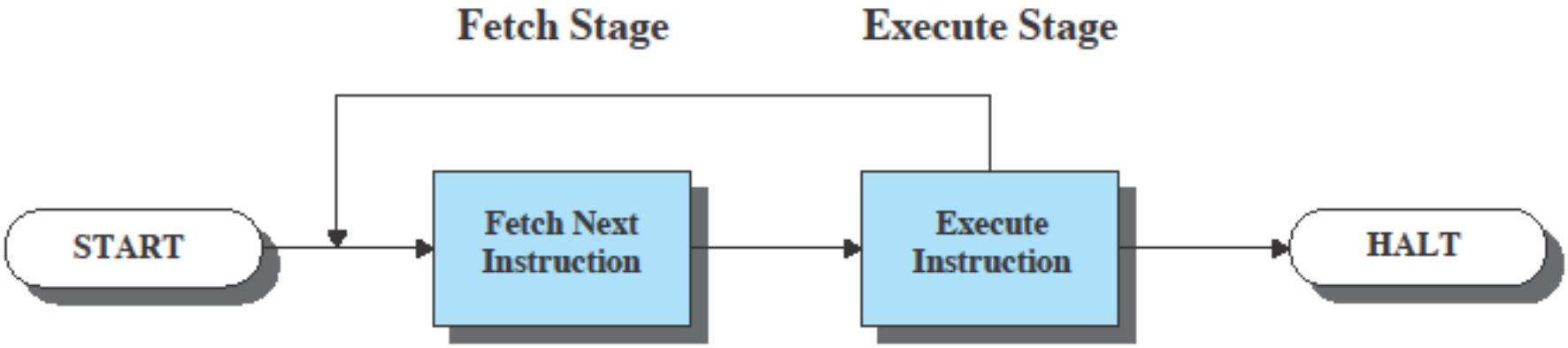
→ Kommunikation nötig

- Jedes Gerät besitzt ein **Statusregister** und ein **Pufferregister**
- **CPU** muss Daten aus dem **Keyboard buffer** in den **Display buffer** kopieren

Frage: Welche Schritte sind dazu nötig?

Übertragen der Datenpuffer

- Wir gehen vorerst weiterhin vom zweistufigen CPU-Zyklus aus:



Zum Übertragen des **Keyboard buffer** in den **Display buffer** sind folgende Schritte nötig:

1. Warten bis `CHARACTER AVAILABE` auf 1 ist
2. Warten bis `READY FOR NEXT CHARACTER` auf 1 ist
3. Kopiere den **Keyboard buffer** in ein Register `R1`
4. Kopiere `R1` in den **Display buffer**
5. Warte bis `READY FOR NEXT CHARACTER` auf 1 und Befehl abgeschlossen ist

Programmiertes Warten (1)

Angenommen, wir schreiben ein Terminal Programm (ähnlich Linux Shell):

WHILE True

(1) **WHILE** `CHARACTER AVAILABLE` != 1:
pass/NOP

LOAD `KEYBOARD BUFFER` **to** `R1` // Controller setzt `CHARACTER AVAILABLE` auf 0

(2) **WHILE** `READY FOR NEXT CHARACTER` != 1:
pass/NOP

STORE `R1` **to** `DISPLAY BUFFER` // Controller setzt `READY FOR NEXT CHARACTER` auf 0

(3) **WHILE** `READY FOR NEXT CHARACTER` != 1:
pass/NOP

- Die CPU muss an den Stellen (1), (2), (3) aktiv auf die E/A-Geräte warten
  - (1) bis ein Zeichen eingegeben wurde
  - (2) bis das Display bereit ist
  - (3) bis das Display die Anzeige abgeschlossen hat

→ Programmiertes Warten (busy waiting)

Programmiertes Warten (2)

Frage: Ist programmiertes Warten wirklich ein Problem?

Beispiel zu den Auswirkungen

- CPU mit 1 GHz ~ 10^9 Befehle pro Sekunde
- Festplatte mit 7200 RPM und 4ms Zugriffszeit

→ Festplatte ist 4 Millionen mal langsamer als CPU

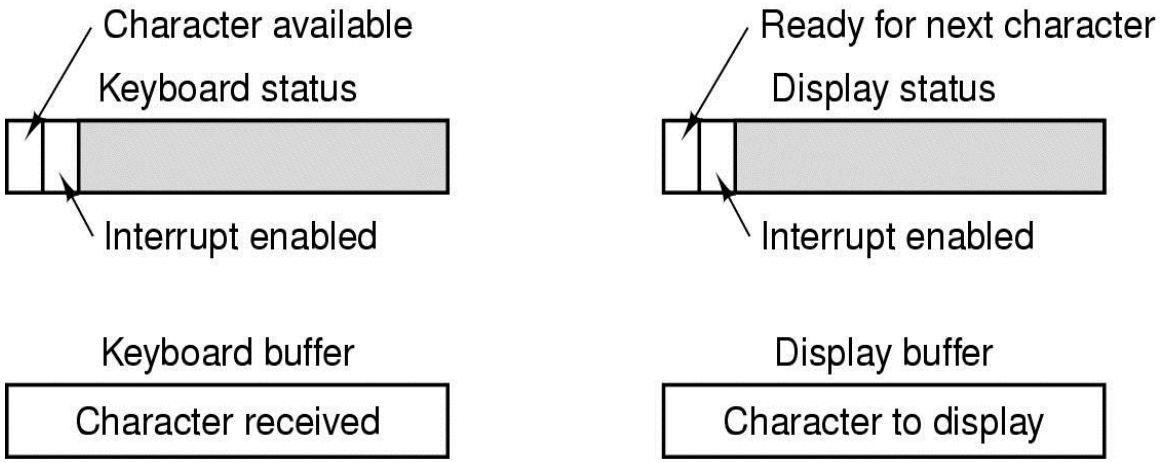
→ Beim Schreiben vergehen im Worstcase 4ms in denen die CPU nur wartet

- **Vorteil:** Abgeschlossene E/A-Operationen werden sofort erkannt (Echtzeit OS)
- **Nachteil:** Die CPU leistet durch die Warteschleifen keine „sinnvolle“ Arbeit

→ Optimal wäre es z.B. einen anderen Prozess in der Zwischenzeit auszuführen

E/A mit Interrupts (1)

- **Idee:** Wenn fertig, schickt der E/A-Controller eine Nachricht an den Prozessor
- **Beispiel aus dem Alltag:**
  - Schicken eines wichtigen Formulars an eine Behörde
  - Permanentes Warten auf Antwort am Briefkasten
  - ODER: Briefträger bitten zu klingeln und in der Zwischenzeit etwas anderes tun
- Ein Programm setzt beim Absetzen der E/A-Operation das **`INTERRUPT ENABLE`**-Flag

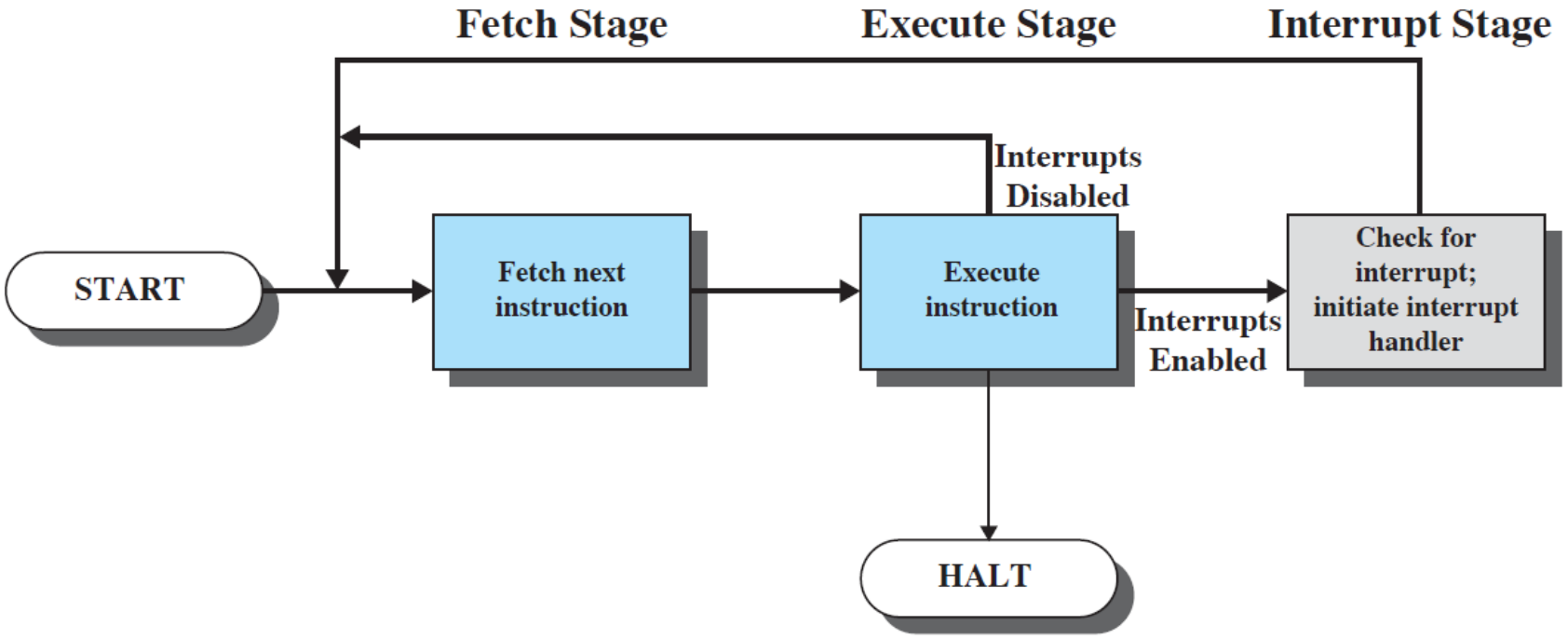


→ Im E/A-Controller : `INTERRUPT ENABLE` **`AND`** `READY FOR NEXT CHARACTER`

→ Absetzen eines Interrupt-Signals auf dem Bus durch den E/A-Controller

E/A mit Interrupts (2)

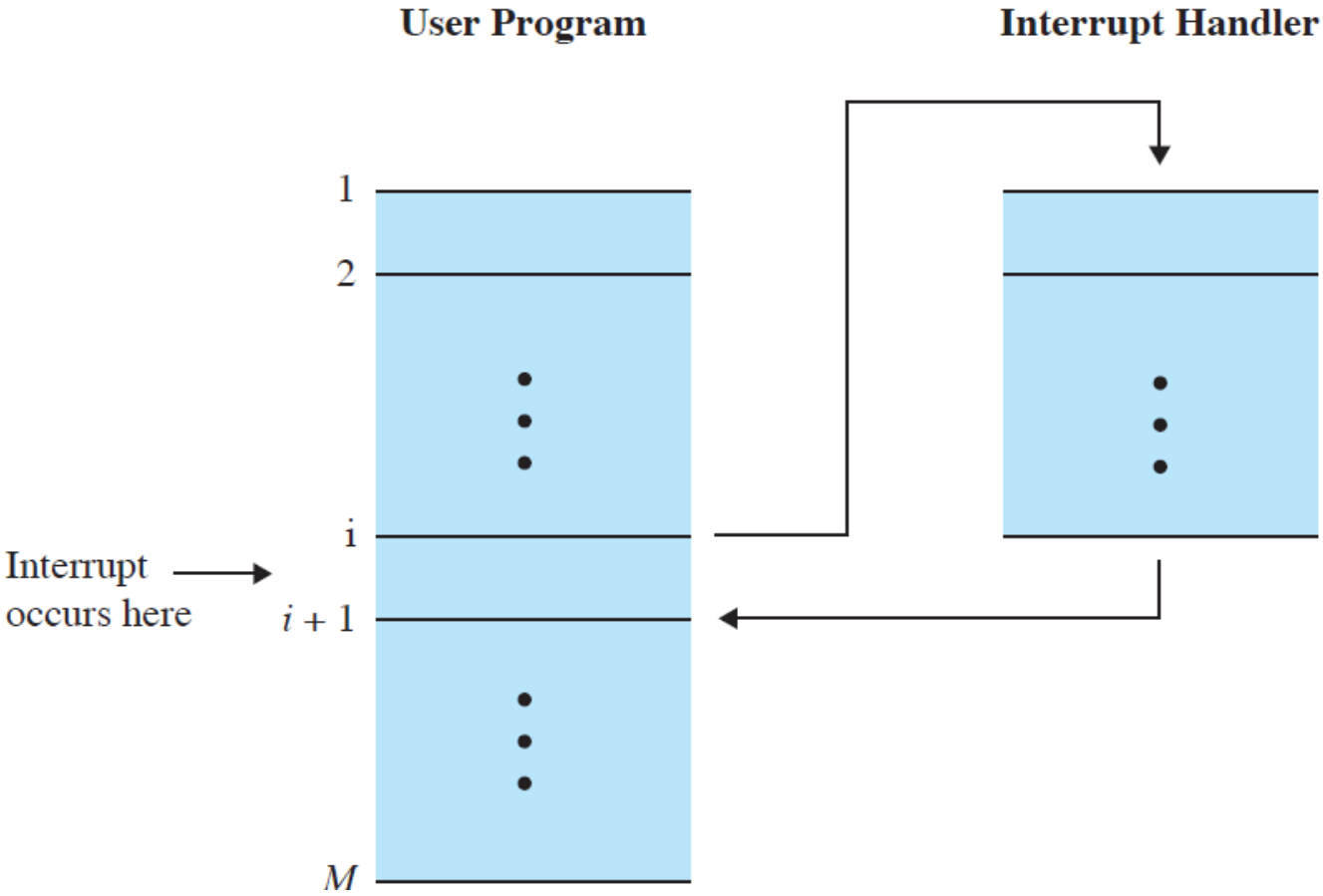
- Bei Interrupt-fähigen Systemen wird der Befehlszyklus im Prozessor angepasst:



- Mit jedem IR ist ein spezieller Interrupt-Handler verknüpft
- CPU springt an die Adresse des Interrupt-Handlers
- Das Betriebssystem kann Systemroutinen, z.B. **WRITE** bereitstellen und blockiert den Nutzerprozess bis alle Worte geschrieben worden sind

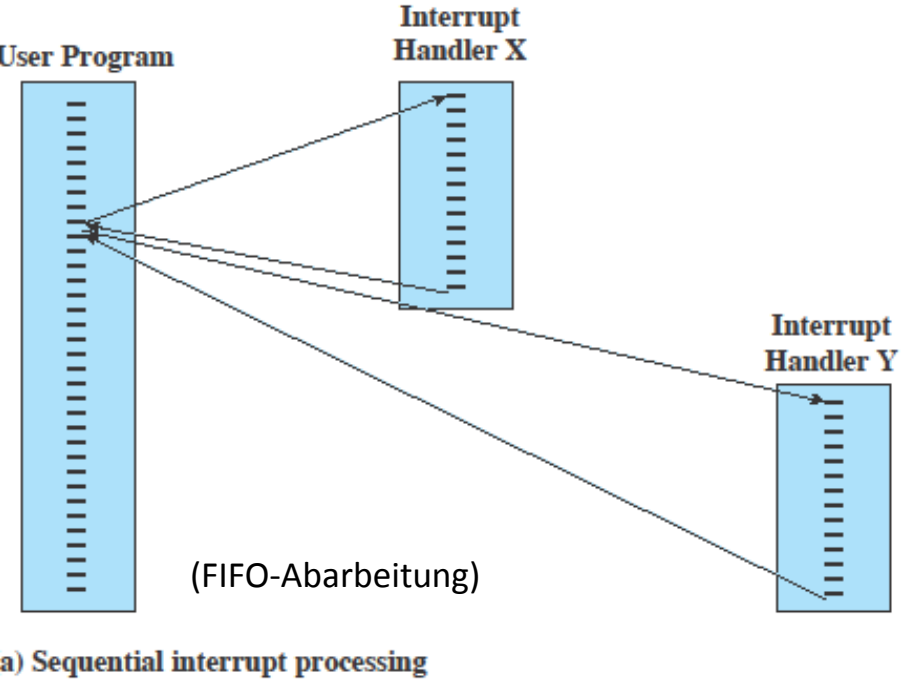
E/A mit Interrupts (3)

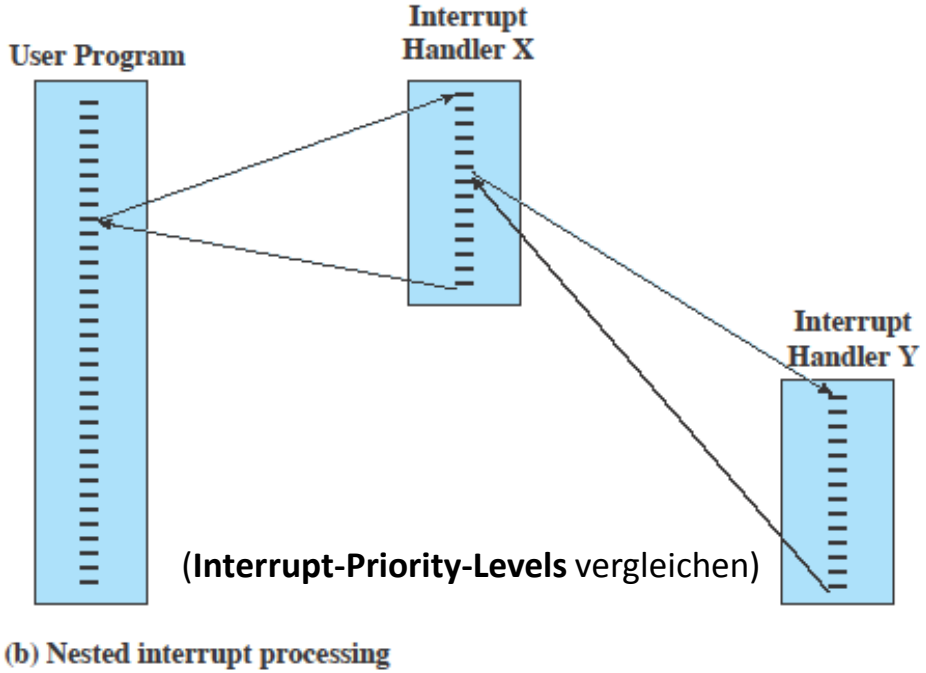
- Prozessor führt nun die nächsten Befehle oder ein anderes User Programm aus…
- … bis der E/A-Controller den Interrupt auslöst:



E/A mit Interrupts (4)

- Ein Interrupt besitzt eine ID
- Die Speicheradresse des passenden Interrupt-Handlers wird aus der ID abgeleitet
- Es gibt zwei Möglichkeiten Interrupts abzuarbeiten:

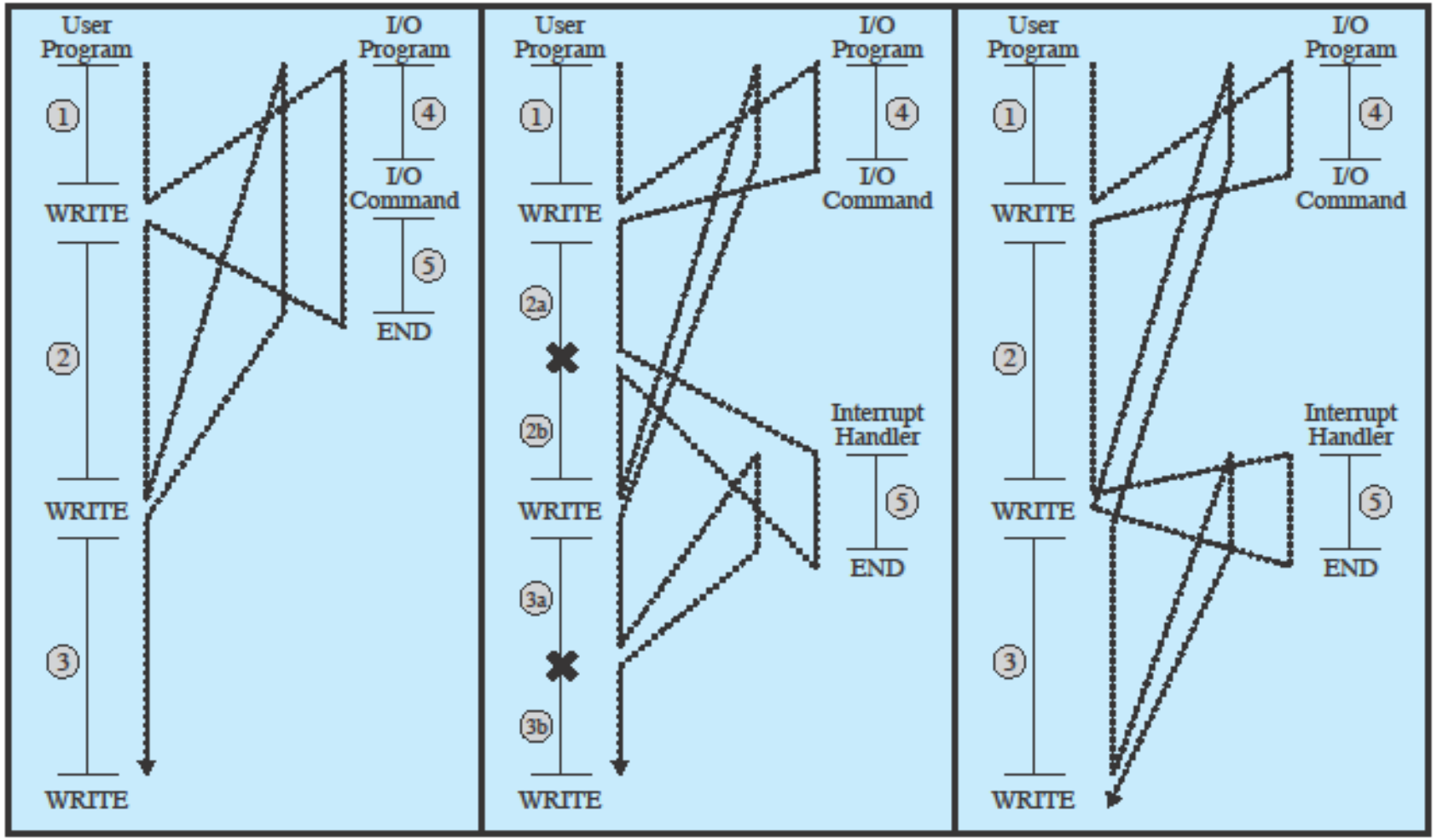




Sequentielle Abarbeitung (langsam) vs. **Verschachtelte Abarbeitung (teuer)**

E/A mit Interrupts (5)

- Der Programmablauf **ohne** und **mit** Interrupts:



(a) No interrupts (b) Interrupts; short I/O wait (c) Interrupts; long I/O wait

E/A mit Interrupts (6)

a) Ablauf ohne Interrupts
   - Nutzerprogramm bleibt blockiert, bis E/A-Operation abgeschlossen ist
   - Beispiel mit programmiertem Warten

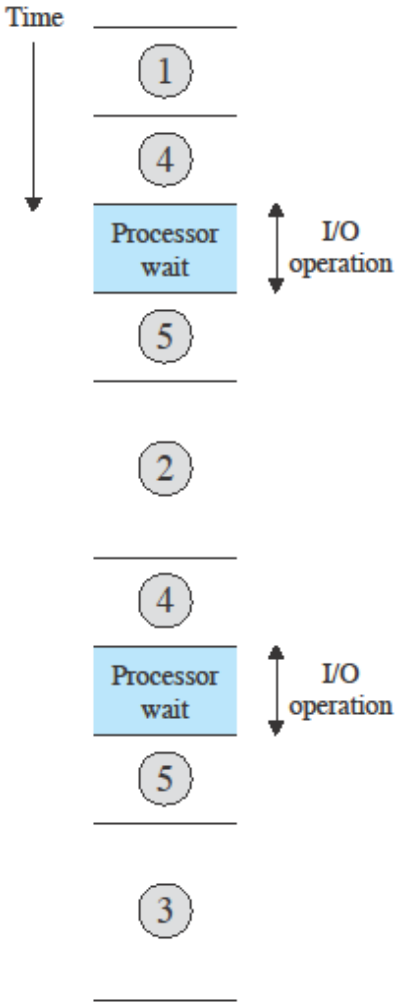
b) Ablauf mit Interrupts für kurze E/A-Operationen
   - Nutzerprogramm kann echt parallel zur E/A-Operation arbeiten
   - Wird erst durch Interrupt und Interrupt-Handler unterbrochen

c) Ablauf mit Interrupts für langwierige E/A-Operationen
   - Die nächste E/A-Operation darf erst ausgelöst werden, wenn die 1. abgeschlossen ist
   - Das Nutzerprogramm muss trotz Interrupts warten, bis die erste E/A-Operation beendet ist
   - In diesem Fall bringen Interrupts weniger Gewinn

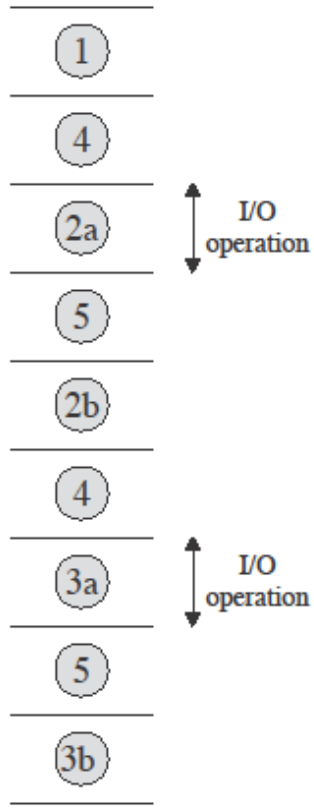
E/A mit Interrupts (7)

- Linke Seite:
  - Ohne Interrupts
  - Prozessor wartet bis E/A-Operation abgeschlossen ist
- Rechte Seite:
  - Programmcode 2a kann während der E/A-Operation ausgeführt werden

→ Eine Abarbeitung mit Interrupts trägt zur besseren CPU-Auslastung bei



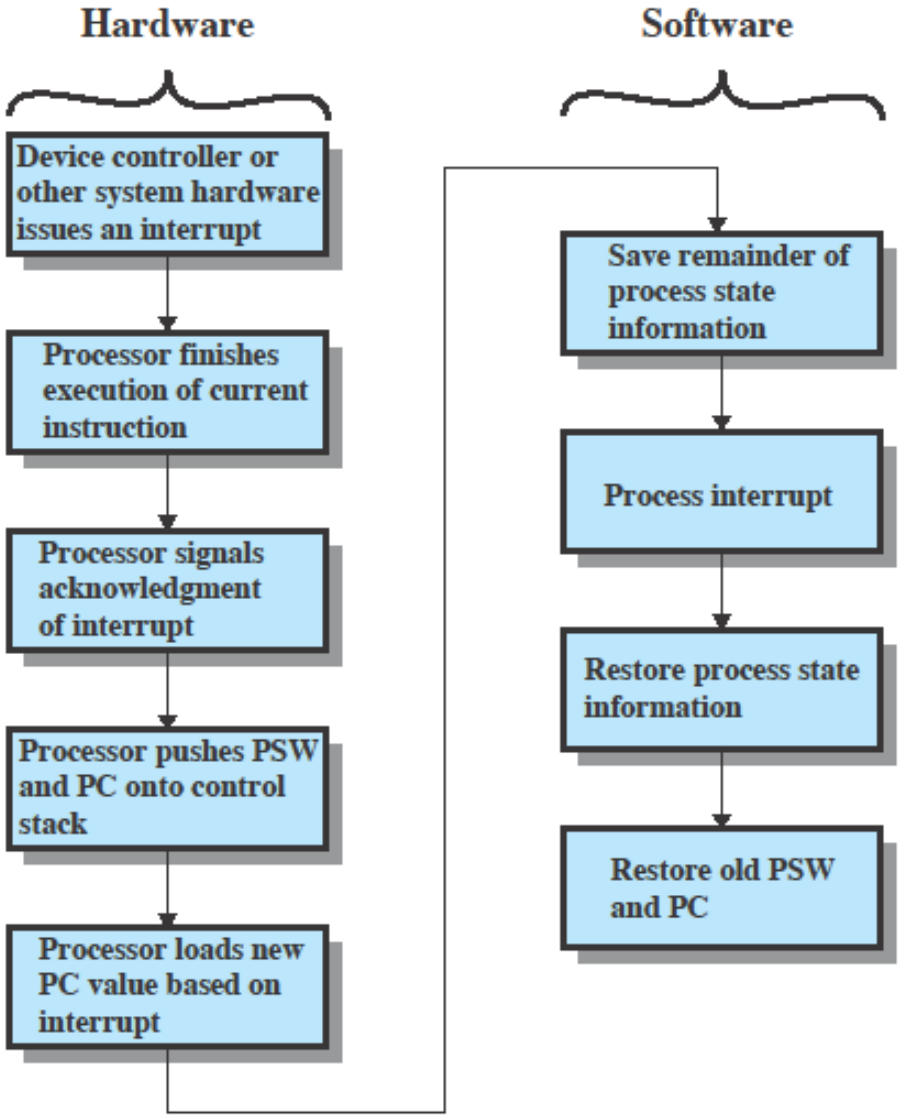
(a) Without interrupts (circled numbers refer to numbers in Figure 1.5a)



(b) With interrupts (circled numbers refer to numbers in Figure 1.5b)

Abarbeitung von Interrupts (2)

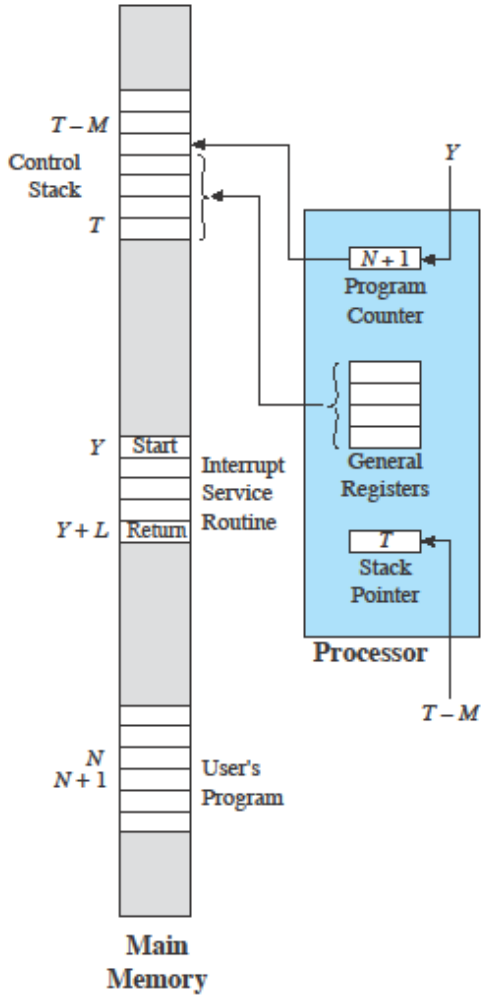
- Bei einem eintreffenden Interrupt übernimmt die Hardware einen Teil der Arbeit…



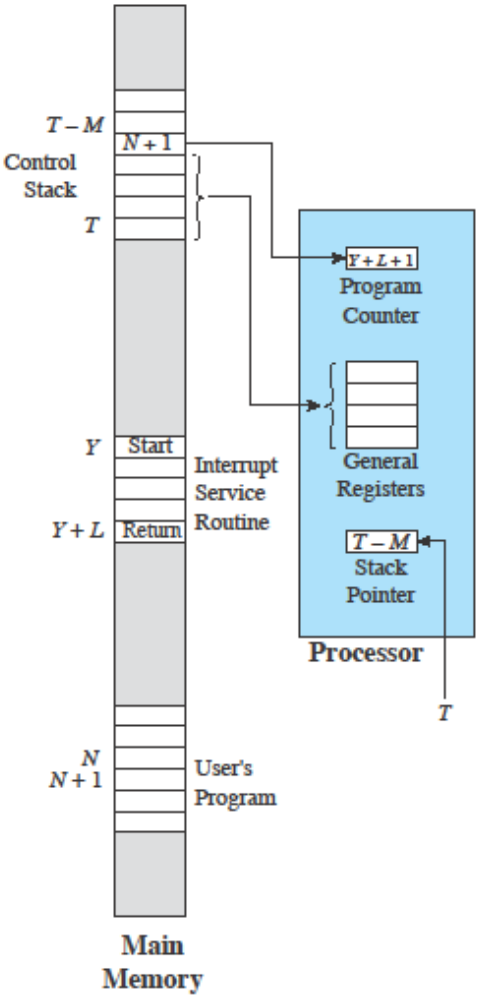
Abarbeitung von Interrupts (1)

- Im Hauptspeicher liegen
  - Controll Stack (OS)
  - Interrupt Handler
  - Nutzerprogramm

- Oft ruft IR-Handler dann OS-Routine auf
- Bevor IR-Handler startet:
  - Sichern aller Daten, so dass Nutzerprogramm später fortgesetzt werden kann
  - Umfasst z.B.:
    - Registerinhalte
    - Program Counter
    - Stack Pointer
    - …



(a) Interrupt occurs after instruction at location N



(b) Return from interrupt

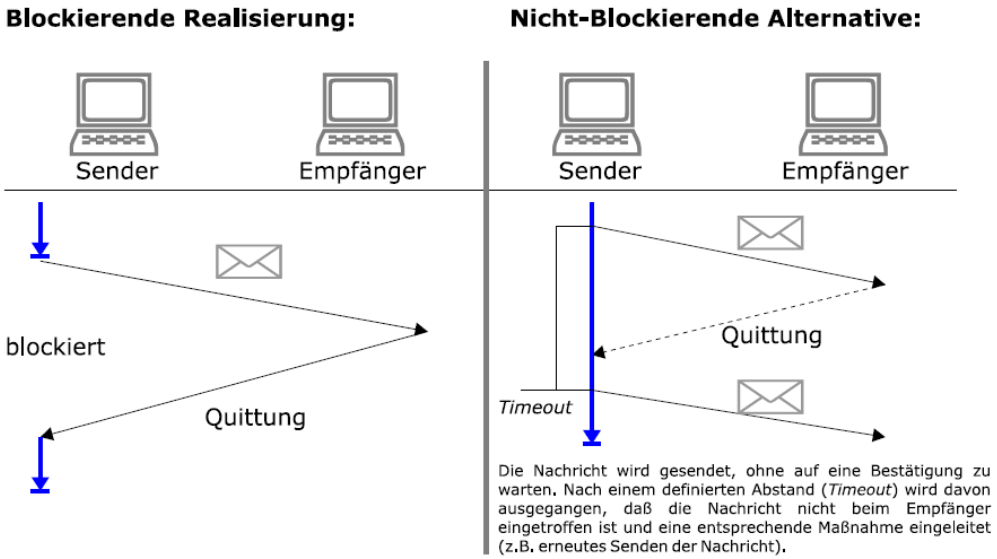
Beispiel: Netzwerkkommunikation

- Eine OS-Routine zum Senden einer Nachricht über das Netzwerk kann z.B. **blockierend** oder **nicht-blockierend** realisiert werden

- Szenario: Empfänger soll Empfang mit Quittung bestätigen
- Zwei Realisierungen

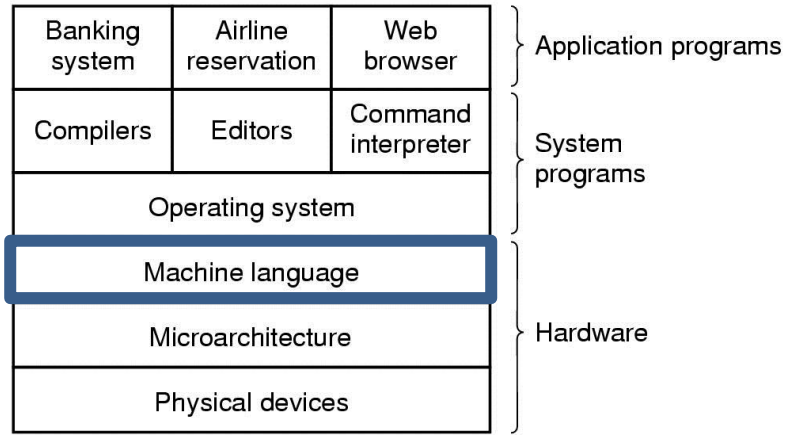
(1) Sender blockiert bis Empfang bestätigt

(2) Sender geht davon aus, dass Übertragung erfolgreich war. Falls **timeout** wird ein Interrupt ausgelöst.



→ Im Fall (1) kann die CPU in der Wartezeit nicht arbeiten (bsp. Programmiertes Warten)

→ Im Fall (2) kann die CPU anderen Aufgaben in der Wartezeit nachgehen



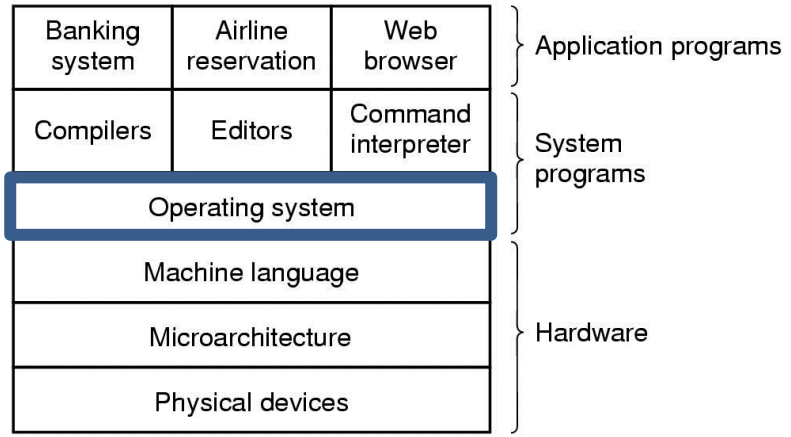
DIE MASCHINENSPRACHE AM BEISPIEL VON SPIM (RECHNERARCHITEKTUR)

Assemblerprogrammierung

- Assemblersprache:
  - Hardwarenahe Programmiersprache
  - Wird von Assembler direkt in ausführbaren Maschinencode umgewandelt
  - Alle Verarbeitungsmöglichkeiten des Mikrokontrollers werden genutzt
  - Hardwarekomponenten können direkt angesteuert werden
  - Erlauben Namen für Instruktionen, Speicherplätze, Sprungmarken, etc.
  - I.d.R. effizient, geringer Speicherplatzbedarf
  - Anwendung:
    - Gerätetreiber
    - Eingebettete Systeme
    - Echtzeitsysteme
    - Neue Hardware (Keine Bibliotheken vorhanden)
    - Programmierung von Mikroprozessoren (Bsp.: MIPS)

```
lw      $t0,($a0)
add     $t0, $t1,$t2
sw      $t0,($a0)
jr      $ra
```

Beispiel: Assemblercode SPIM



PROZESSE UND THREADS IM BETRIEBSSYSTEM

Hintergründe (1)

- Ein Betriebssystem hat viele Aufgaben:
  - Zuordnung von Ressourcen zu Prozessen
  - **Multiprogramming** zur Verbesserung der Auslastung
  - Bereitstellung von Möglichkeiten zur **Interprozesskommunikation**

- Konzeptuelle Grundlagen:
  1. Ein Computer besitzt eine Menge **von Hardware Ressourcen**
  2. Computerprogramme werden für bestimmte **Aufgaben** geschrieben
  3. Anwendungen sollten **plattformunabhängig** entwickelt werden:
     - Mehrere Programme könnten redundanten Code enthalten
     - Die CPU selbst unterstützt nur begrenzt Multiprogramming… eine Software muss diese Aufgabe übernehmen
     - Daten, E/A-Zugriffe auf Ressourcen usw. müssen bei mehreren aktiven Applikationen vor gegenseitigen Zugriffen geschützt werden
  4. OS ist eine Schicht mit konsistentem **Interface zwischen Hardware und Software** womit Applikationen bequem, sicher und vielfältig zugreifen können
  5. **Abstrakte Ressourcen** und Verwaltung von Zugriffen auf diese

Hintergründe (2)

- Ein OS soll Applikationen verwalten, so dass diese:
  - auf Ressourcen zugreifen können
  - sich einen gemeinsamen Prozessor teilen können
  - den Prozessor und die E/A-Geräte effizient nutzen können

→ Alle modernen Betriebssysteme basieren auf einem Modell, wo die Ausführung einer Applikation der Existenz von mindestens einem Prozess entspricht

Prozesse

Ein Prozess…

- kann als Eigentümer und Verwalter von Ressourcen betrachtet werden.
- verfügt über einen Bereich im Hauptspeicher.
- kann z.B. über zugeordnete E/A-Geräte, E/A-Kanäle und Dateien verfügen.

- kann als Einheit, die eine gewisse Aufgabe erledigt, betrachtet werden.
- ist ein in Ausführung befindliches (nicht zwingend aktives) Maschinenprogramm.

Beide Aufgaben sind unabhängig voneinander

- Einheit, die den **Eigentümer von Ressourcen** bezeichnet, wird weiter **Prozess** genannt
- Einheit, die eine **gewisse Aufgabe** erledigt, wird **Thread (Faden)** genannt

Threads

- Innerhalb eines Prozesses kann es mehrerer Threads geben
- Sind „leichtgewichtige Prozesse“
  - Teilen sich gleichen Adressraum
  - Haben gemeinsamen Zugriff auf Speicher und Ressourcen d. Prozesses
  - Eigener Stack für lokale Variablen
  - Eigener Deskriptor

Nebenläufigkeit

Mögliche Abläufe:

Sequentielles Programm:
Anweisungen werden Schritt für Schritt hintereinander ausgeführt ("single thread of control")

Paralleles Programm (Nebenläufig):
Anweisungen oder Teile der Anweisungen eines Programms werden nebeneinander ausgeführt ("multi thread of control")

Echt gleichzeitig parallel

Prozesse/Threads werden auf mehreren Kernen echt parallel ausgeführt

Zeitlich verzahnt (quasi-parallel)

Prozesse/Threads teilen sich einen Kern und werden somit durch Scheduling unterbrochen und verzahnt ausgeführt

Vor- und Nachteile paralleler Programmierung

Vorteile:

- Komplexität in parallele Teilbereiche zerlegen
- Höherer Durchsatz
- Performanz
- Ausnutzung bei eingebetteten und verteilten Systemen

Nachteile:

- Erhöhte Komplexität
- Abläufe sind häufig schwer zu durchschauen
- Sehr fehleranfällig
- Schwer zu Debuggen bei Laufzeitfehlern
- Konzepte zur Synchronisation und Thread-Sicherheit erforderlich, um deterministisches Verhalten zu gewährleisten

Threads in Java

Bei früheren BS: Nur Simulation der verzahnten Ausführung durch die JVM

- Keine echte Parallelisierung
  - Falls keine direkte Thread-Unterstützung durch das BS möglich

Heute i.d.R.: JVM bildet die Threadverwaltung auf BS ab

- native Threads (bzw. KLT)
- Echte parallele Ausführung auf mehreren Kernen möglich!

Bringt einige Probleme mit sich:

- Nicht deterministisches Verhalten
- Deadlocks
- Sicherheit
- Lebendigkeit (bsp.: Fairness)
- Ressourcenverbrauch

Achten auf Thread-Sicherheit und Synchronisation!

Nebenläufigkeit in Java

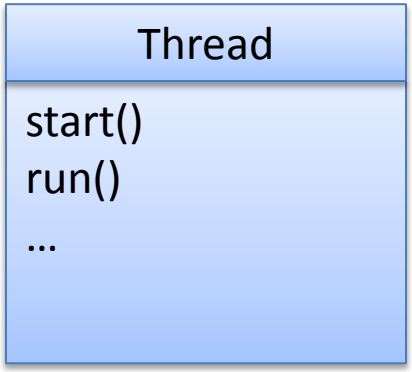
Die Java-Bibliothek besitzt eine Reihe von Klassen, Schnittstellen und Aufzählungen für Nebenläufigkeit:

- Thread
  - Jeder laufende Thread stellt ein Exemplar dieser Klasse dar
- Runnable
  - Programmcode, der parallel ausgeführt werden soll
- Lock
  - Mit Lock können kritische Bereiche markiert werden (nur 1 Thread innerhalb krit. Bereich)
- Condition
  - Threads können auf Benachrichtigungen anderer Threads warten

Threads in Java: Realisierung

Threads in Java stellen Objekte der Klasse `java.lang.Thread` dar

- Die Klasse beinhaltet eine `run()` Methode, die den Code beinhaltet, der parallel ausgeführt werden soll
- Die `run`-Methode muss mittels `threadinstanz.start()` ausgeführt werden, damit der Code parallel zum aufrufenden Thread ausgeführt wird!
- **Achtung**: Ruft man `threadinstanz.run()` auf, wird der Code in der `run-Methode` „ganz normal“ also sequentiell ausgeführt



Threads in Java: Erzeugung

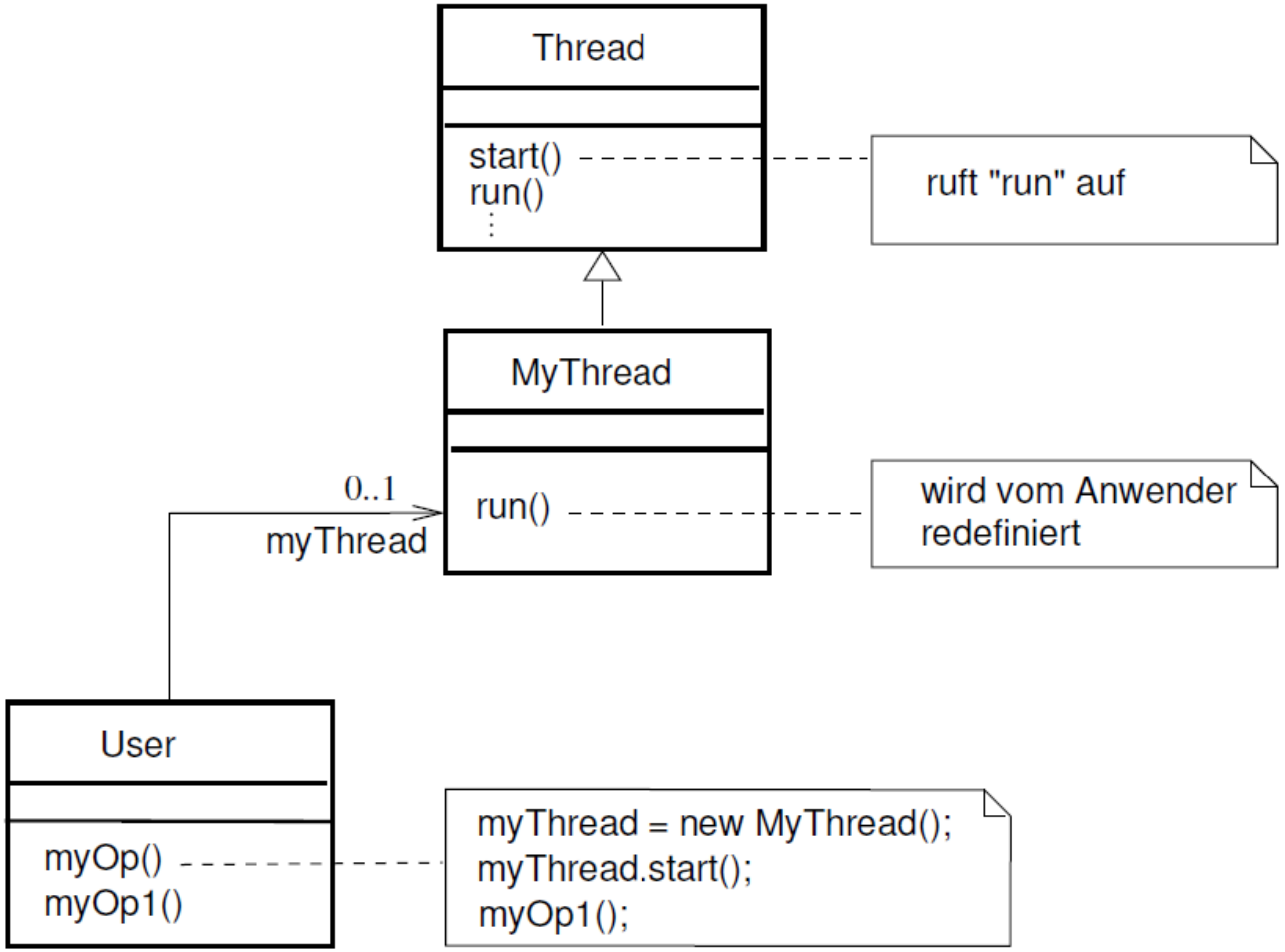
Prinzipiell stehen uns 4 Möglichkeiten zur Verfügung, um Threads in Java programmatisch zu erzeugen:

- Thread als eigene Klasse (Datei):
  - Erben von der Klasse Thread
  - Implementieren des Interfaces `Runnable`
- Threads direkt im Quellcode:
  - Anonyme Klasse
    - Entweder `Runnable` oder im Konstruktor von Thread
  - Lambda Ausdrücke (Seit Java 8)

Die 4 Möglichkeiten basieren auf den beiden Konzepten:

- Threads über Vererbung
  - Nachteil: Das Erben einer weiteren Klasse ist nicht möglich!
- Threads über Interface `Runnable`

Threads mittels Vererbung



Quelle: Skript Parallele Programmierung. R. Hennicker 2011

Threads mittels Vererbung

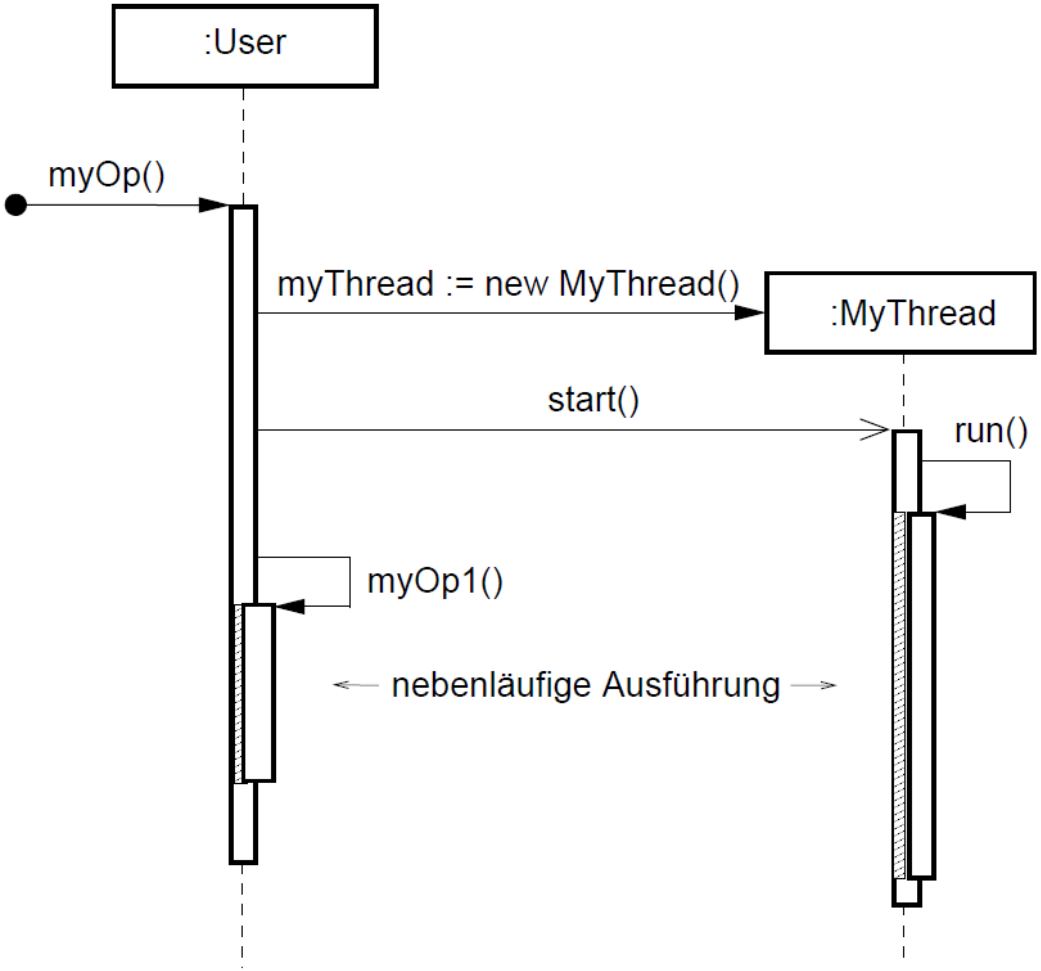
Beispiel-Implementierung mittels Vererbung:

```
// User
public class User {

   public void myOp(){
      MyThread t = new MyThread("Peter");
      t.start();
   }

   public void myOp1(){
      System.out.println("Operation 1.");

   }

   public static void main(String[] args) {
      User u  = new User();
      u.myOp();
      u.myOp1();
   }
}
```

```
// MyThread
public class MyThread extends Thread{


   private String name = "";


   public MyThread(String name){
      this.name = name;
   }

   @Override
   public void run() {
      System.out.println("Hallo ich bin
   "+this.name);
   }
}
```

Threads mittels Vererbung

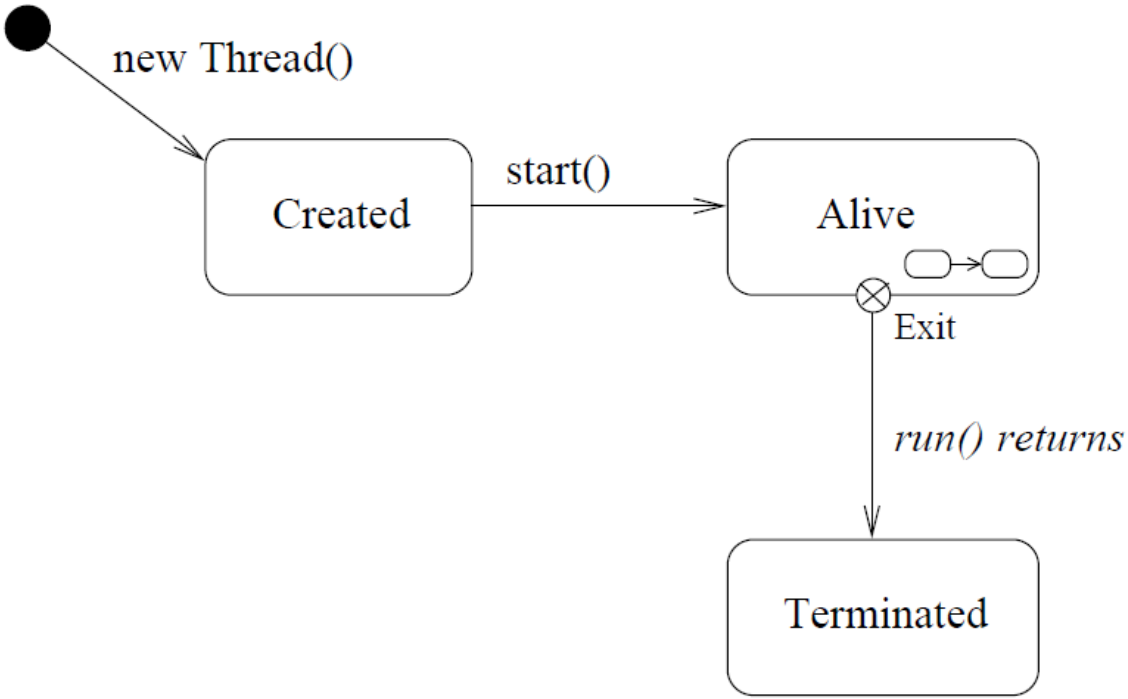
Was passiert: Sequenzdiagramm:



Quelle: Skript Parallele Programmierung. R. Hennicker 2011

Threads in Java: Lebenszyklus

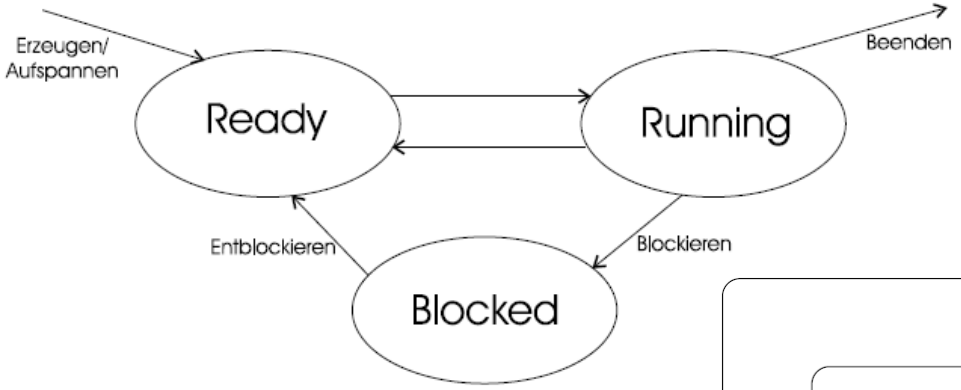
Lebenszyklus eines Java-Threads



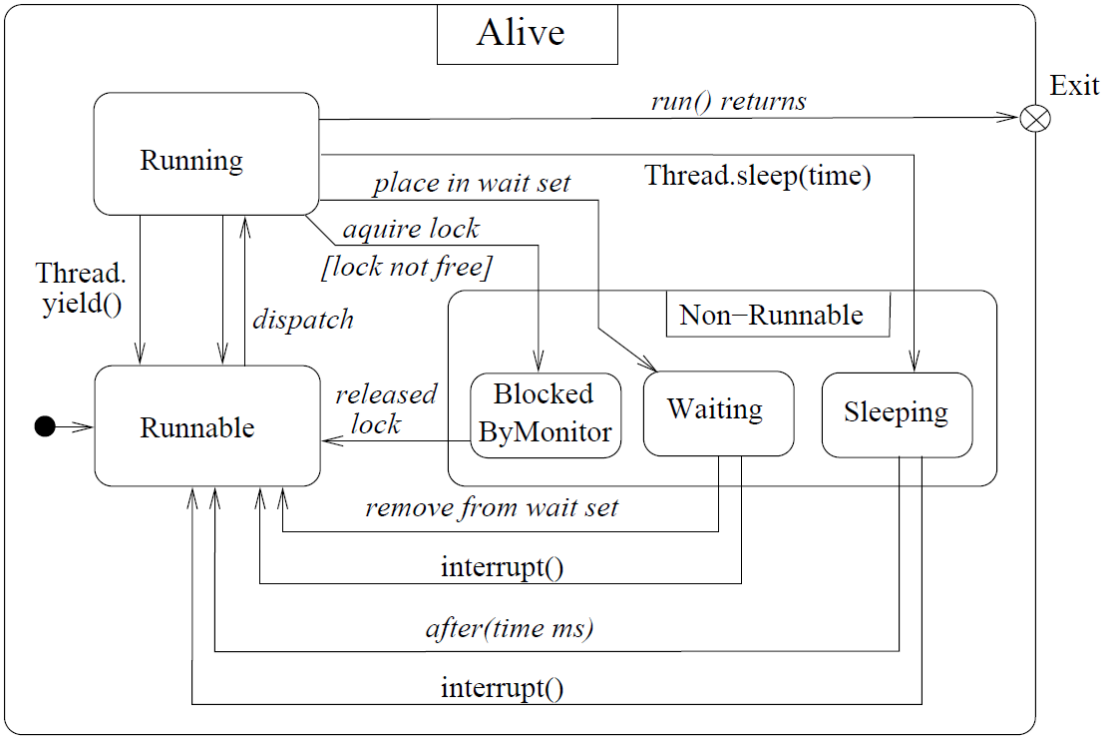
Quelle: Skript Parallele Programmierung. R. Hennicker 2011

Threads in Java: Zustände

- Threadzustände (Alive)



Quelle: Skript Betriebssysteme. C. Linnhoff 2015



Quelle: Skript Parallele Programmierung. R. Hennicker 2011

Kritische Abläufe

Motivation

- Threads verwalten ihre eigenen Daten
  - lokale Variablen
  - und einen Stack
- Sie stören sich also selbst nicht
- Auch Lesen von gemeinsamen Daten ist unbedenklich
- Schreiboperationen sind jedoch kritisch!

- Probleme können durch Scheduling entstehen:
  - Ein Thread arbeitet gerade an Daten, die ein anderer Thread bearbeitet.
  - Hier können gravierende und schwer vorhersehbare Inkonsistenzen entstehen!

- Wir brauchen also Mechanismen, die uns davor schützen!

Kritische Abschnitte

- Wenn wir mehrere Threads haben und Programmblöcke, auf die immer nur ein Thread zugreifen sollte, dann müssen diese kritischen Abschnitte geschützt werden!
  - Ist zur gleichen Zeit immer nur ein Thread beim Abarbeiten eines Programmteils, dann liegt ein wechselseitiger Ausschluss bzw. eine atomare Operation vor
  - Arbeitet ein Programm bei nebenläufigen Threads falsch, ist es nicht *thread-sicher* (engl. *thread-safe*).

Was ist zu Schützen?

Prinzipiell sollten kritische Abschnitte und nicht atomare Schreibeoperationen geschützt sein.

- Manche Befehle sehen atomar aus, sind es aber nicht.
  - Bsp.: `i++`

```
// Was passiert bei i++?

1. i wird gelesen und auf dem Stack abgelegt
2. Danach wird die Konstante 1 auf dem Stack abgelegt
3. Und anschließend werden beide Werte addiert
4. Das Ergebnis wird nun vom Stack geholt und in i geschrieben
```

Aus diesem Grund müsste `i++` geschützt ausgeführt werden.

- Java-Konstrukte zum Schutz der kritischen Abschnitte
  - Synchronized
  - Die Schnittstelle java.util.concurrent.locks.Lock
    - Wird u.a. von ReentrantLock implementiert

Das Java Monitor-Konzept

Monitore:

- Werden implizit durch die JVM erstellt
- Werden durch eine automatisch verwaltete Sperre realisiert
- Jedes Objekt verfügt über eine Sperre (Lock)
- Eintrittspunkte der Monitore müssen mit dem Schlüsselwort `synchronized` markiert sein
  - Für Klassenmethoden
    - Bsp.: `public synchronized static void doIt(){…}`
  - Für Objektmethoden
    - Bsp.: `public synchronized void makeIt(){…}`
  - Für Blöcke
    - Bsp.: `synchronized (objMitMonitor){…}`
- Ein solcher Eintrittspunkt kann nur betreten werden, wenn das Lock verfügbar ist.
  - Ansonsten muss der Thread warten, solange bis das Lock verfügbar ist.

Das Java Monitor-Konzept

- Ein freies Lock wird beim Betreten einer `synchronized` Methode/Block durch den aufrufenden Thread belegt (oder gehalten)
  - Daraufhin kann kein anderer Thread mehr eine synchronisierte Methode des Objekts betreten (solange bis das Lock wieder freigegeben wird)

- Die statische Methode `Thread.holdsLock()` zeigt an, ob der aktuelle Thread das Lock hält.

- Ein gehaltenes Lock wird freigegeben, wenn:
  - … die synchronisierte Methode verlassen wird
  - … eine Ausnahme erfolgt
  - … ein `wait()` Aufruf getätigt wird.

Lock Freigabe

Durch Ausnahme (Exception)

- Lock wird bei einer unbehandelten RuntimeException in einer `synchronized` Methode/Block automatisch durch JVM freigegeben
  - Da bei einer Exception der Block automatisch verlassen wird

Durch Aufruf von `wait()`

- Thread beendet die Abarbeitung
- Geht in den „Blocked“ Zustand
  - Reiht sich in die Warteschlange des Objekts ein
- Das Lock wird freigegeben
- Der Thread wartet nun auf eine Benachrichtigung durch (`notify()` oder `notifyAll()`), dass er wieder weiterabreiten darf
- Wartende Threads können auch durch einen Interrupt unterbrochen werden
  - Daher: `throws InterruptedException` (Muss abgefangen werden!)
- Aufruf von `wait()`, `notify()` oder `notifyAll()` nur möglich, wenn Lock gehalten wird!
  - Ansonsten Laufzeitfehler: `IllegalMonitorStateException`

Zusammenspiel zwischen `wait()`, `notify()` und `notifyAll()`

Zusammenfassung der Methoden:

- `void wait() throws InterruptedException`
  - Thread wartet an dem aufrufenden Objekt darauf, dass er nach einem notify() bzw. notifyAll() weiterarbeiten kann.
- `void wait(long timeout) throws InterruptedException`
  - Wartet auf ein notify()/notifyAll() maximal aber eine gegebene Anzahl von Millisekunden. Nach Ablauf dieser Zeit ist er wieder rechenbereit.
- `void wait(long timeout, int nanos) throws InterruptedException`
  - Etwas spezifischer als vorher
- `void notify()`
  - Weckt einen beliebigen Thread auf, der an diesem Objekt wartet und sich wieder um das Lock bemühen kann.
    - Erhält er das Lock, kann er die Bearbeitung fortführen.
- `void notifyAll()`
  - Benachrichtigt alle Threads, die auf dieses Objekt warten.

Hinweis: `notify()` bzw. `notifyAll()` i.d.R., wenn aufrufender Thread dann auch das Lock freigibt (also am Ende eines `synchronized` Blocks)

- Sonst werden die Threads zwar aufgeweckt, aber das Lock ist immer noch vom aktuellen Thread belegt („signal and continue“-Prinzip)

Zusammenspiel zwischen `wait()`, `notify()` und `notifyAll()`


```
Allgemeines Beispiel zum Zusammenspiel zwischen wait() und notify()

public class MeinKlasse{

   private Data state;

   public synchronized void op1() throws InterruptedException {
      while (!cond1) wait();
      // modify monitor state
      notify();
      // or notifyAll();
   }

   public synchronized void op2() throws InterruptedException {
      while (!cond2) wait();
      // modify monitor state
      notify();
      // or notifyAll();
   }
}
```

Die `while`-Schleife wird deshalb gebraucht, da bei Fortführung die Synchronisationsbedingung nicht notwendigerweise gelten muss!Ein einfaches `if` kann evtl. nicht ausreichen

Warum nicht gleich alles synchronisieren?

Unerwünschte Nebeneffekte können also durch Markieren der kritischen Abschnitte mittels `synchronized` verhindert werden!

Warum dann nicht gleich jede Methode synchronisieren?

Antwort: Führt zu anderen Problemen:

- Synchronisierte Methoden müssen von JVM verwaltet werden, um wechselseitigen Ausschluss zu ermöglichen.
  - Threads müssen auf andere warten können
  - Das erfordert eine Datenstruktur, in der wartende Threads eingetragen und ausgewählt werden
    => Kostet Zeit und Ressourcen!
- Unnötig und falsch synchronisierte Blöcke machen die Vorteile von Mehrprozessormaschinen zunichte.
  - Lange Methodenrümpfe erhöhen die Wartezeit für die anderen!
- Deadlock-Gefahr steigt!

Quelle: http://openbook.rheinwerk-verlag.de/javainsel9/bilder/365_java_09_004.gif

VIELEN DANK