
Karlheinz Hug

Informatik 3

**Folien-Skript
Teil 2**

Probleme, Konzepte und Lösungsmuster nebenläufiger Programmierung

**Bachelor-Studiengang
Medien- und Kommunikationsinformatik**

Fakultät Informatik

Hochschule Reutlingen
Reutlingen University

Maskulin - Feminin	Alle Berufs-, Rollen- und Personenbezeichnungen in diesem Skript und begleitenden Dokumenten wie Leser, Benutzer, Entwickler beziehen sich stets auf Menschen beiderlei Geschlechts.
Warenzeichen	<i>BlackBox</i> und <i>Component Pascal</i> sind eingetragene Warenzeichen von Oberon micro-systems. <i>Java</i> ist ein eingetragenes Warenzeichen von Sun Microsystems. <i>Ada</i>, <i>BON</i>, <i>C</i>, <i>C++</i>, <i>C#</i>, <i>Eiffel</i>, <i>HTML</i>, <i>Modula</i>, <i>Oberon</i>, <i>OC</i>L, <i>Pascal</i>, <i>Prolog</i>, <i>UML</i>, <i>XML</i> und andere Namen sind möglicherweise eingetragene Warenzeichen. Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Skript berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetze als frei zu betrachten wären und daher von jedermann benutzt werden dürften.
Garantieverzichts- erklärung	Alle in diesem Folien-Skript und begleitenden Dokumenten enthaltenen Informationen wie Texte, Abbildungen, Programme und Verfahren wurden sorgfältig erstellt und geprüft. Da Fehler trotzdem nicht auszuschließen sind, ist der Inhalt des Skripts mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Der Autor übernimmt für eventuell verbliebene fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung für Schäden, die in Zusammenhang mit der Verwendung dieses Skripts, der darin dargestellten Methoden und Programme, oder Teilen davon entstehen.
Missbrauchsverbot	Die durch dieses Folien-Skript vermittelten Informationen wie Prinzipien, Konzepte, Methoden, Verfahren und Techniken dürfen nicht für militärische, völkerrechtswidrige, rassistische oder sonstige inhumane Zwecke missbraucht werden.
Anschrift des Autors	Prof. Dr. Karlheinz Hug Hochschule Reutlingen - Reutlingen University Fakultät Informatik Studiengang Medien- und Kommunikationsinformatik Alteburgstraße 150 72762 Reutlingen Bundesrepublik Deutschland - Germany
Gebäude-Raum	9-126
Telefon	+49/7121/271- 4013
E-Mail	karlheinz.hug@reutlingen-university.de
WWW	http://www.mki.reutlingen-university.de
Online-Dienst	http://userserv.reutlingen-university.de/~hug ftp://studinf.reutlingen-university.de/MKI/Hug

Vorwort

Was?

Der zweite Teil des Skripts zu *Informatik 3* führt in das Thema Nebenläufigkeit ein. Grundlegende Probleme und Lösungsansätze der nebenläufigen Programmierung werden vorgestellt. Exemplarisch wird Javas Unterstützung nebenläufiger Programmierung durch Threads und Synchronisations- und Kommunikationskonstrukte behandelt.

Inhaltsverzeichnis

Vorwort	iii
Inhaltsverzeichnis	v
Abbildungsverzeichnis	ix
Aufgabenverzeichnis	xi
Leitlinienverzeichnis	xiii
Programmverzeichnis	xv
Tabellenverzeichnis	xix
1 Einführung	
1.1 Sequenzialität, Determinismus, Determiniertheit	1
1.1.1 Sequenzialität	1
1.1.2 Determinismus	2
1.1.3 Determiniertheit	3
1.1.4 Welche Eigenschaften sind notwendig?	4
1.1.5 Verzicht auf Determiniertheit	5
1.1.6 Verzicht auf Sequenzialität	6
1.1.7 Verzicht auf Determinismus	7
1.2 Nebenläufigkeit	9
1.3 Ebenen der Nebenläufigkeit in Rechnersystemen	10
1.4 Wozu Nebenläufigkeit studieren?	11
1.5 Nebenläufigkeit auf Hardware-Befehlsebene	12
1.6 Arten von Nebenläufigkeit	18
1.7 Nebenläufigkeit auf Betriebssystem-Prozessebene	21
1.7.1 Sequenzielles System	21
1.7.2 Prozess und nebenläufiges System	23
1.7.3 Eigenschaften von Prozessen	24
1.7.4 Prozessor-Multiplexing und -Virtualisierung	25
1.7.5 Gemeinsame Nutzung von Code und Daten	27
1.7.6 Arten von Prozessen	28
1.7.6.1 Klassifizierung nach Aufgaben	28
1.7.6.2 Klassifizierung nach Implementation	29
1.8 Koordination von Prozessen	30
1.8.1 Konzepte der Koordination	30
1.8.2 Zwecke der Koordination	31
2 Nebenläufige Abläufe	
2.1 Programmiersprachen-Ausdrucksebene	1
2.2 Programmiersprachen-Anweisungsebene	2
2.3 Programmiersprachen-Programmeinheitsebene	3
2.3.1 Einfaches prozedurähnliches Prozesskonstrukt	4
2.3.2 Prozedur und Prozess	5
2.3.3 Modul, Klasse, Prozess	6

2.3.4	Kommunikationskonzepte	7
2.3.5	Implementationen programmiersprachlicher Prozesse	8
2.4	Abriss der historischen Entwicklung	9
2.5	Unabhängige Threads in Java	14
2.5.1	Klasse java.lang.Thread	15
2.5.2	Erweitern der Klasse Thread	16
2.5.3	Schnittstelle java.lang.Runnable	20
2.5.4	Implementieren der Schnittstelle Runnable	21
2.5.5	Beispiel: Mehrere iterierende Threads	24
2.5.6	Beispiel: Mehrere iterierend schlafende Threads	27
2.6	Verzweigen und Zusammenführen	29
2.6.1	Petri-Netze	29
2.6.1.1	Verzweigen	29
2.6.1.2	Zusammenführen	30
2.6.1.3	Verzweigen und Zusammenführen	31
2.6.1.4	Symmetrisches Verzweigen und Zusammenführen	36
2.6.1.5	Geschachteltes Verzweigen und Zusammenführen	37
2.6.2	Nebenläufigkeitsanweisungen	39
2.6.2.1	Symmetrisches Verzweigen und Zusammenführen	39
2.6.2.2	Geschachteltes Verzweigen und Zusammenführen	39
2.6.3	Unix	40
2.6.3.1	Systemaufrufebene	40
2.6.3.2	Shell-Ebene	41
2.6.4	Java	42
2.6.4.1	Vater und Sohn	42
2.6.4.2	Vater und zwei Söhne	43
2.6.4.3	Vater, Sohn und Enkel	44
2.6.5	BlackBox	46
2.6.6	Beispiel: Sortiert-Prüfung	48
2.6.6.1	Rekursive Spezifikation durch Vertrag	48
2.6.6.2	Implementation mit Nebenläufigkeitsanweisung	49
2.6.6.3	Implementationen in Java und BlackBox	50
3	Synchronisationskonzepte	
3.1	Zwecke der Synchronisation	1
3.2	Probleme bei Synchronisation	2
3.3	Beispiel: Ein Parkhaus-Problem	4
3.3.1	Lösungsansatz 1.1	6
3.3.2	Lösungsansatz 1.2 mit Überlastschutz	7
3.3.3	Lösungsansatz 1.3 ohne aktives Warten	8
3.3.4	Lösungsansatz 1.1 auf Maschinenbefehlebene	9
3.3.5	Exemplarische Abläufe	10
3.3.6	Zwischenbilanz	12
3.4	Kritische Abschnitte	13
3.4.1	Unteilbarkeit	14
3.4.2	Wichtige Synchronisationsbedingungen	15
3.4.3	Zustände eines kritischen Abschnitts	16

3.4.4	Ausschluss und Ausschlussgraph	17
3.4.5	Beispiel: Parkhaus-Problem	18
3.5	Gegenseitiger Ausschluss	19
3.5.1	Implementationen mit Synchronisationsprimitiven	20
3.5.1.1	Ist Synchronisation mit einfachen Maschinenbefehlen möglich?	23
3.5.1.2	Dekkers Algorithmus	24
3.5.1.3	Lamports Bäckerei-Algorithmus	27
3.5.1.4	Unterbrechungssperre	30
3.5.1.5	Teste-und-Setze-Befehl	32
3.5.1.6	Unterbrechungssperre und Teste-und-Setze-Befehl	35
3.5.2	Fazit zu Synchronisationsprimitiven	39
3.6	Anforderungen an höhere Synchronisationsmechanismen	40
3.7	Semaphore	41
3.7.1	Gegenseitiger Ausschluss	44
3.7.2	Beispiel: Parkhaus-Problem	45
3.7.2.1	Abstrakter Lösungsansatz	46
3.7.2.2	Konkurrenzsynchronisation	48
3.7.2.3	Kooperationssynchronisation	50
3.7.2.4	Kooperations- und Konkurrenzsynchronisation	52
3.7.3	Fazit zu Semaphoren	55
3.8	Monitore	56
3.8.1	Parkhaus-Problem mit Konkurrenzsynchronisation	58
3.8.2	Kooperations-/Ereignissynchronisation	59
3.8.2.1	Konflikt zwischen signalisierendem und signalisiertem Prozess	60
3.8.2.2	Hoares Signal-Semantik	61
3.8.3	Parkhaus-Problem mit Kooperationssynchronisation	62
3.8.3.1	Signalisierter Prozess erhält Monitorkontrolle	62
3.8.3.2	Signalisierender Prozess behält Monitorkontrolle und läuft weiter	63
3.8.4	Gegenseitiger Ausschluss	65
3.8.5	Semaphore	66
3.8.6	Fazit zu Monitoren	68
3.9	Monitore in Java	69
3.9.1	Semantik verglichen mit Hoares Monitor	69
3.9.1.1	Ausschlussynchronisation	69
3.9.1.2	Ereignissynchronisation	70
3.9.1.3	Semantik von notify und notifyAll	71
3.9.1.4	Semantik von wait	72
3.9.2	Gegenseitiger Ausschluss	73
3.9.3	Semaphore	77
3.9.4	Parkhaus-Problem	78
3.10	Aufgaben	79

A Literaturverzeichnis

Abbildungsverzeichnis

1 Einführung	
Bild 1.1 Schichtenmodell eines Rechnersystems	10
Bild 1.2 Einprozessorsystem	12
Bild 1.3 System mit E/A-Prozessor	13
Bild 1.4 Eng gekoppeltes Mehrprozessorsystem	16
Bild 1.5 Lose gekoppeltes Mehrprozessorsystem	17
Bild 1.6 Physikalische Nebenläufigkeit	18
Bild 1.7 Logische Nebenläufigkeit	19
Bild 1.8 Prozessor-Multiplexing und Virtualisierung	25
2 Nebenläufige Abläufe	
Bild 2.1 Auswerten eines Ausdrucks	1
Bild 2.2 Thread-Objekte & Threads	18
Bild 2.3 Fork	29
Bild 2.4 Join	30
Bild 2.5 Fork & Join	31
Bild 2.6 Fork & Join	32
Bild 2.7 Fork & Join	33
Bild 2.8 Fork & Join	34
Bild 2.9 Fork & Join	35
Bild 2.10 Mehrfaches Fork & Join	36
Bild 2.11 Geschachteltes Fork & Join	37
Bild 2.12 Geschachteltes Fork & Join	38
3 Synchronisationskonzepte	
Bild 3.1 Verklemmung im Straßenverkehr	2
Bild 3.2 Aushungerung im Straßenverkehr	3
Bild 3.3 Parkhaus	4
Bild 3.4 Parkhaus mit Zähler & Aufpassern	5
Bild 3.5 Ausschlussgraph für einseitigen Ausschluss	17
Bild 3.6 Ausschlussgraph für gegenseitigen Ausschluss	17
Bild 3.7 Ausschlussgraph für Parkhauszähleraktualisierung	18
Bild 3.8 Parkhaus mit Zähler & Tickets	45
Bild 3.9 Monitor als synchronisierter ADT	56

Aufgabenverzeichnis

1	Einführung	
2	Nebenläufige Abläufe	
	Übung 2.1 Unabhängige Threads	28
	Übung 2.2 Verzweigte & zusammengeführte Threads	50
3	Synchronisationskonzepte	
	Übung 3.1 Fehlschlag	23
	Übung 3.2 Korrektheit zeigen (Prüfung SS 2006, 15 Punkte)	29
	Übung 3.3 Missgriff	35
	Übung 3.4 Irrtum	54
	Übung 3.5 Implementation von Monitoren mit Semaphoren	67
	Übung 3.6 Pkw-Parkhaus-Simulation	79
	Übung 3.7 Parkhaus mit Semaphoren	79

Leitlinienverzeichnis

1	Einführung	
	Regel 1.1 Nebenläufigkeit	20
2	Nebenläufige Abläufe	
3	Synchronisationskonzepte	

Programmverzeichnis

1 Einführung	
Code 1.1 Minimum	7
Code 1.2 Sortieren	8
2 Nebenläufige Abläufe	
Code 2.1 Tauschen sequenziell	2
Code 2.2 Tauschen nebenläufig	2
Code 2.3 Prozesse	4
Code 2.4 Java: Thread	15
Code 2.5 Java: SimpleThread	16
Code 2.6 Java: SimpleThread	19
Code 2.7 Java: Runnable	20
Code 2.8 Java: SimpleRunner	21
Code 2.9 Java: SimpleRunner	23
Code 2.10 Java: IteratingThread	24
Code 2.11 Java: IteratingSleeper	27
Code 2.12 Java: FatherSonThread	42
Code 2.13 Java: Father2SonsThread	43
Code 2.14 Java: FatherSonGrandsonThread	44
Code 2.15 CP: Services	46
Code 2.16 Sortiert spezifiziert	48
Code 2.17 Sortiert implementiert	49
3 Synchronisationskonzepte	
Code 3.1 Parkhaus 1.1	6
Code 3.2 Parkhaus 1.2	7
Code 3.3 Parkhaus 1.3	8
Code 3.4 Parkhaus 1.1MBE	9
Code 3.5 Abstrakte Sperre	20
Code 3.6 Undichte Sperre	23
Code 3.7 Dekkers Sperre	24
Code 3.8 Zwillinge	25
Code 3.9 n Prozesse	27
Code 3.10 Bäckerei-Sperre	28
Code 3.11 Sperre mit maskierten Unterbrechungen	30
Code 3.12 Teste-&-Setze-Befehl	32
Code 3.13 Kreisel-Sperre	33
Code 3.14 Gefährliche Sperre	36
Code 3.15 Korrekte Sperre	37
Code 3.16 Semaphor	42
Code 3.17 Semaphor-Sperre	44

Code 3.18 Abstraktes Parkhaus	46
Code 3.19 Pkw	47
Code 3.20 Ideales Parkhaus	48
Code 3.21 Einfaches Parkhaus	50
Code 3.22 Luxus-Parkhaus	52
Code 3.23 Tödliches Parkhaus	54
Code 3.24 Ideales Parkhaus	58
Code 3.25 Bedingung	59
Code 3.26 Luxus-Parkhaus	62
Code 3.27 Schlichtes Parkhaus	63
Code 3.28 Monitor-Sperre	65
Code 3.29 Monitor-Semaphor	66
Code 3.30 Java: Lock	73
Code 3.31 Java: Lock with Exception Handling	75
Code 3.32 Java: Bad Lock	76
Code 3.33 Java: Semaphore	77
Code 3.34 Java: CarPark	78

Tabellenverzeichnis

1	Einführung	
2	Nebenläufige Abläufe	
	Tabelle 2.1 Eigenschaften von Prozeduren & Prozessen	5
3	Synchronisationskonzepte	
	Tabelle 3.1 Nebenläufige Ablaufverfolgung	10
	Tabelle 3.2 Nebenläufige Ablaufverfolgung	11
	Tabelle 3.3 Namen für Semaphoroperationen	41

1.1 Sequenzialität, Determinismus, Determiniertheit

1.1.1 Sequenzialität

Sequenzialität :=

Eigenschaft eines dynamischen Systems, in dem Abläufe oder Ereignisse nur nacheinander auftreten können.

Sequenzielles System (*sequential system*) :=

Ereignisse treten in einer Reihenfolge nacheinander auf.

Sequenzielles Programm (*sequential program*) :=

Anweisungen werden nacheinander ausgeführt.

a := b; do_it; if c then d else e end; from f until g do h end

1.1.2 Determinismus

Determinismus :=

Eigenschaft eines dynamischen Systems, in dem vorbestimmt ist, wann & wo welche Abläufe oder Ereignisse auftreten.

Deterministisches System (*deterministic system*) :=

Ereignisse treten in einer vorbestimmten Weise auf.

Deterministisches Programm (*deterministic program*) :=

Anweisungen werden nach vorbestimmten Regeln ausgeführt.

- Alle Ausführungen mit gleichen Eingangsdaten verlaufen gleich.
- Abläufe sind reproduzierbar.

1.1.3 Determiniertheit

Determiniertheit :=

Eigenschaft eines dynamischen Systems,
bei gleichen Anfangszuständen zu gleichen Endzuständen zu führen.

Determiniertes System (*determined system*) :=

Ereignisse sind nicht vorbestimmt,
führen aber stets vom Start zum Ziel.

Determiniertes Programm (*determined program*) :=

Gleiche Eingaben führen zu gleichen Ausgaben.

- Alle Ausführungen liefern dieselben Ergebnisse.
- Ergebnisse sind reproduzierbar.

Beispiel: E-Mail-Versand determiniert, nicht deterministisch.

Deterministische Programme sind determiniert. Umkehrung gilt nicht.

1.1.4 Welche Eigenschaften sind notwendig?

„Klassische“ Programme sind

- sequenziell
- deterministisch
- determiniert.

Zum Lösen von Problemen oft wichtig:

- Determiniertheit.

Verzichtbar:

- Determinismus
- Sequenzialität.

1.1.5 Verzicht auf Determiniertheit

Randomisierung

- Zufallszahlengenerator benutzen
- bestimmte Variablen mit Zufallswerten initialisieren

Nützlich

- ☺ zum Testen
- ☺ bei evolutionären & genetischen Algorithmen

Streng genommen sind auch randomisierte Programme deterministisch, da Zufallszahlen durch gleiche Saat reproduzierbar sind.

1.1.6 Verzicht auf Sequenzialität

Sequenzielles Programm

a := x
b := x
x := y

äquivalente Sequenz

b := x
a := x
x := y

Nebenläufigkeitsanweisung (*Algol 68*)

conc a := x || b := x **end**
x := y

- || trennt Anweisungsfolgen, die nicht sequenziell ausgeführt werden müssen.

Kollektive Zuweisung (*Algol 68, Python*)

(a, b, x) := (x, x, y)

- erst rechte Seiten auswerten (Reihenfolge nicht festgelegt)
- dann Werte linken Seiten zuweisen (Reihenfolge nicht festgelegt)
- sequenziell oder nichtsequenziell implementierbar

1.1.7 Verzicht auf Determinismus

Bewachte Anweisungen, nichtdeterministische Auswahl (*guarded commands, non-deterministic selection*, Dijkstra 1975)

Code 1.1
Minimum

```
min (in a, b) is
do
  select
    when a <= b then result := a
    when b <= a then result := b
  end
end
```

nichtdeterministische Auswahl
bewachte Anweisung

- **Wächter** (*guard*): $a \leq b$, $b \leq a$
- Alternativen schließen sich nicht notwendig aus.
- willkürliche Wahl einer Alternative mit wahrem Wächter
- Falls $a = b$, wird entweder $\text{result} := a$ oder $\text{result} := b$ ausgeführt.
- nicht deterministisch, aber determiniert

Nichtdeterministische Wiederholung

Code 1.2
Sortieren

```
sort (inout a, b, c) is
  do
    loop
      when a > b then (a, b) := (b, a)
      when b > c then (b, c) := (c, b)
    end
  end
```

nichtdeterministische Auswahl

- Liefern alle Wächter false, so terminiert die Schleife.

Anwendungen

- *Ada*: **accept**-Anweisung zur Synchronisation nebenläufiger Tasks.
- *CSP, Occam*
- Parnas (ca. 1985) vereinheitlicht die beiden Anweisungen zu einer einzigen nichtdeterministischen strukturierten Anweisung.

1.2 Nebenläufigkeit

Nebenläufigkeit (*concurrency*) :=

Eigenschaft eines dynamischen Systems, in dem Abläufe oder Ereignisse kausal unabhängig voneinander auftreten können.

Kausale Abhängigkeit impliziert zeitliche Nacheinanderfolge.

Umkehrung gilt nicht.

Zeitliche Parallelität (Gleichzeitigkeit) impliziert kausale Unabhängigkeit.

Umkehrung gilt nicht.

Nebenläufiges System (*concurrent system*) :=

Ereignisse können kausal unabhängig voneinander auftreten.

Einfaches Modell

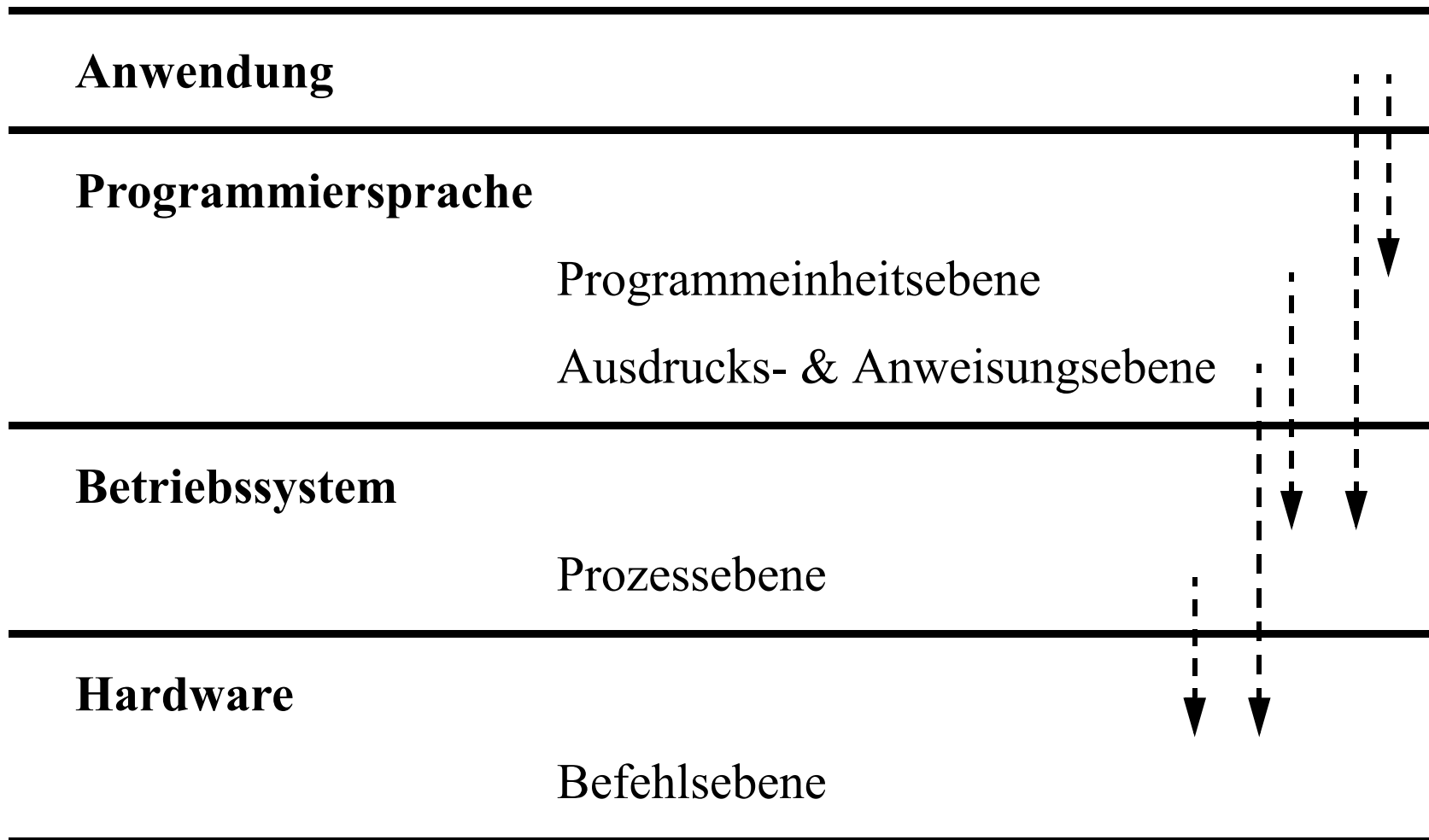
Nebenläufiges System = System sequenzieller Teilsysteme

1.3

Ebenen der Nebenläufigkeit in Rechnersystemen

Bild 1.1

Schichten-
modell
eines
Rechner-
systems



1.4 Wozu Nebenläufigkeit studieren?

Sicht der Anwendung

- Nebenläufigkeit = Konzept zur Lösung von Problemen
Analogie: Rekursion
Nebenläufige Probleme verlangen nebenläufige Lösungen.

Sicht des Systems

- Möglichkeiten von
 - Mehrprozessorarchitekturen
 - Betriebssystemen
 - Programmiersprachennutzen

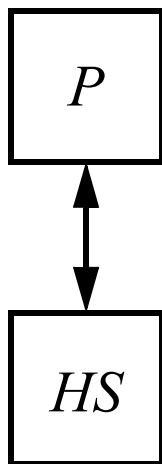
1.5 Nebenläufigkeit auf Hardware-Befehlsebene

Prozessorarchitekturen

Hardware führt Befehle nur sequenziell aus:

- **SISD-Einprozessorsystem** (*single instruction single data*)
 - klassische von neumannsche Architektur
 - Ende 1940er Jahre
 - ermöglicht mit Spezialbefehlen Nebenläufigkeit auf BS-Prozessebene: → Mehrprozessbetrieb

Bild 1.2
Ein-
prozessor-
system

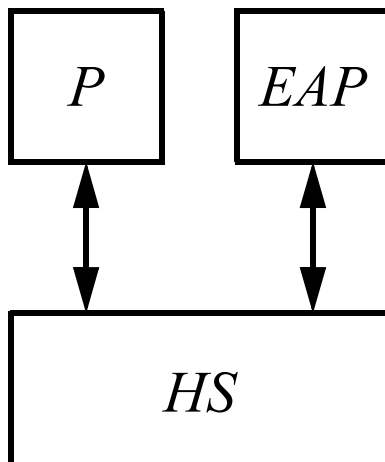


noch Prozessorarchitekturen

Hardware führt mehrere Befehle nebenläufig aus:

- Einprozessorsystem mit **speziellen E/A-Prozessoren**
 - Unterbrechungskonzept
 - Ende 1950er Jahre

Bild 1.3
System mit
E/A-
Prozessor



noch Prozessorarchitekturen

- **SIMD-Mehrprozessorsystem** (*single instruction multiple data*)

- mehrere gleiche Prozessoren
- mit je eigenem lokalem Hauptspeicher
- jeder Prozessor führt dieselbe Befehlsfolge aus
- unterstützt Nebenläufigkeit auf HW-Befehlsebene

$z[1] := x[1] * y[1]; z[2] := x[2] * y[2]$

single instruction: MUL

multiple data: $(x[1], y[1], z[1]), (x[2], y[2], z[2])$

- **Spezialfall Vektorprozessorsystem**

single instruction: Vektoroperation

multiple data: Vektoroperand in Registergruppe

$D1 = (x[1], x[2]), D2 = (y[1], y[2]), D3 = (z[1], z[2])$

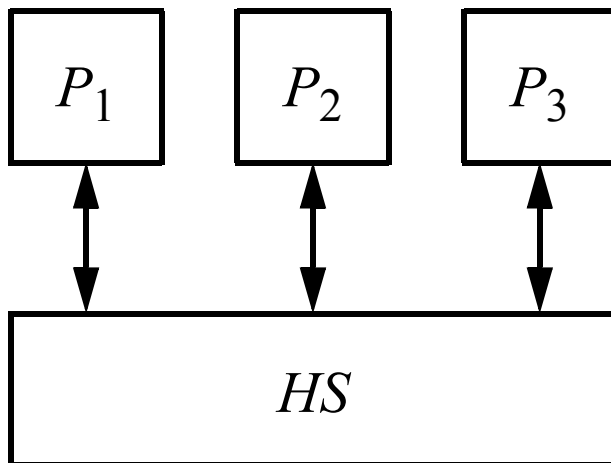
noch Prozessorarchitekturen

- **MIMD-Mehrprozessorsystem** (*multiple instruction multiple data*)
 - mehrere gleiche Prozessoren
 - jeder Prozessor führt eigene Befehlsfolge aus
 - Prozessoren
 - ◆ arbeiten unabhängig
 - ◆ sind synchronisierbar

noch Prozessorarchitekturen

- Spezialfall eng gekoppeltes MIMD-Prozessorsystem
 - gemeinsamer Hauptspeicher
 - synchronisierte Speicherzugriffe
 - Anfang 1960er Jahre
 - unterstützt Nebenläufigkeit auf BS-Prozessebene:
→ Mehrprozessbetrieb

Bild 1.4
Eng
gekoppel-
tes Mehr-
prozessor-
system

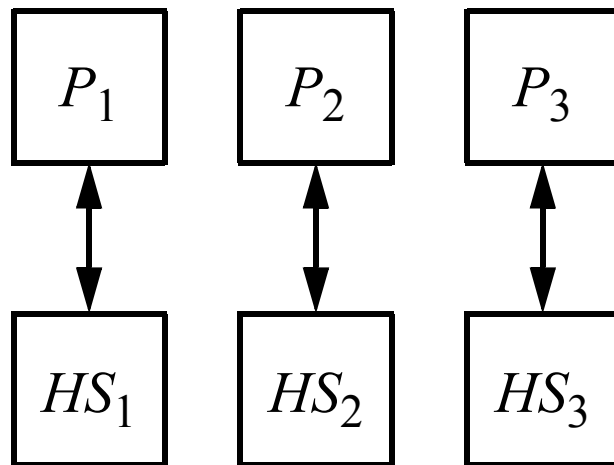


noch Prozessorarchitekturen

- Spezialfall lose gekoppeltes MIMD-Prozessorsystem
 - kein gemeinsamer Hauptspeicher
 - physikalisch
 - (a) konzentriert in einem Kasten oder
 - (b) verteilt auf ein Rechnernetz
 - unterstützt Nebenläufigkeit auf HW-Rechnerebene:
verteilte Systeme

Bild 1.5

Lose
gekoppel-
tes Mehr-
prozessor-
system

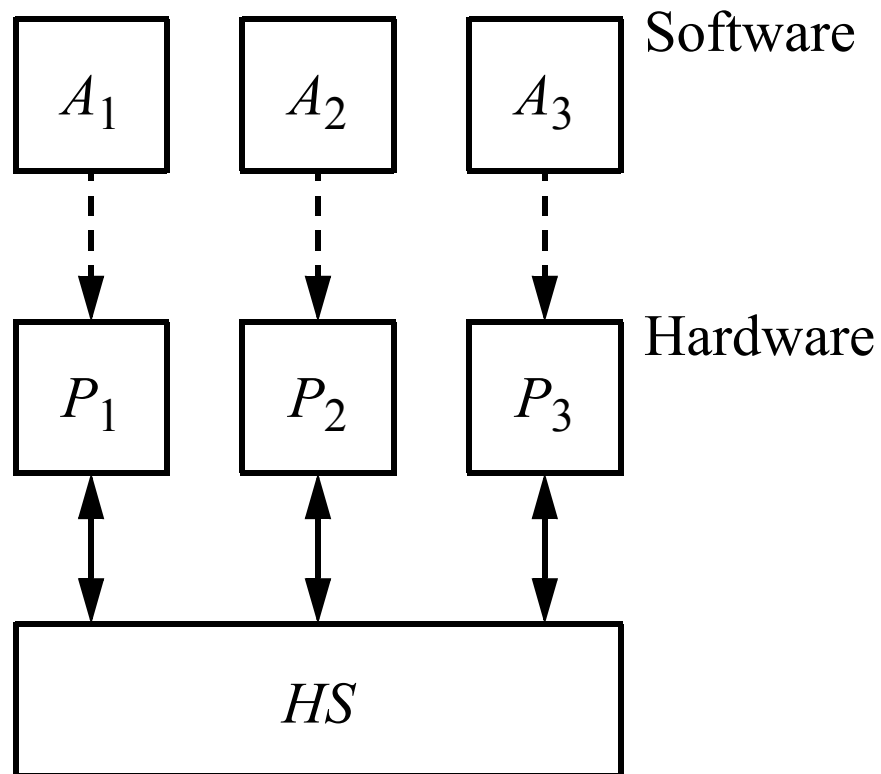


1.6 Arten von Nebenläufigkeit

- **Physikalische Nebenläufigkeit ((echte) Parallelität) :=**

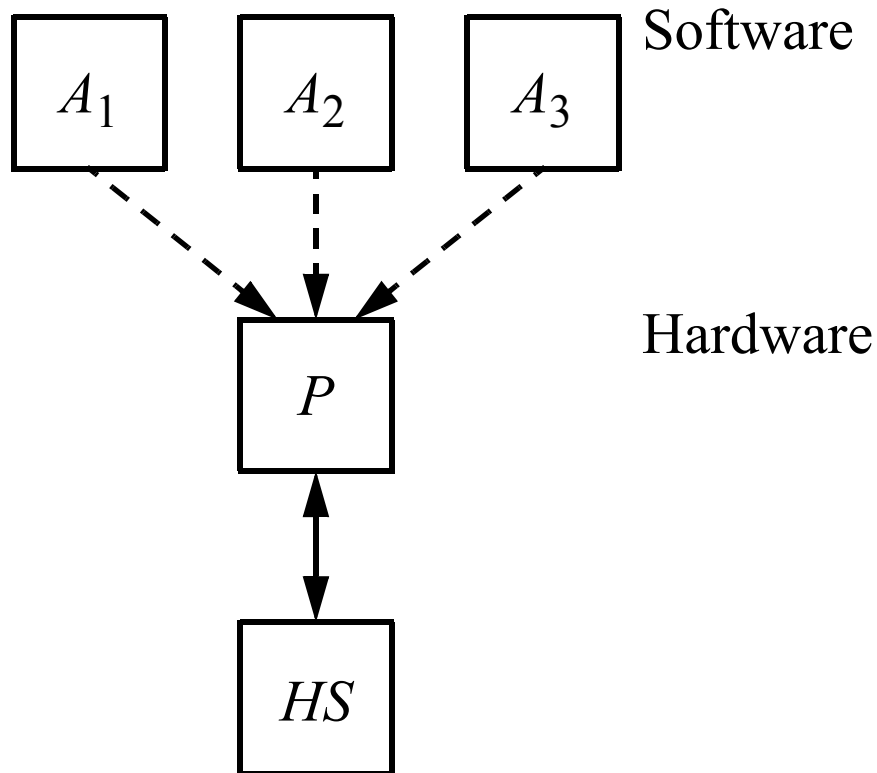
jede nebenläufige Ablaufeinheit
auf eigenen physikalischen Prozessor abbilden

Bild 1.6
Physika-
lische
Neben-
läufigkeit



- **Logische Nebenläufigkeit (Pseudo-, Quasiparallelität) :=**
alle nebenläufigen Ablaufeinheiten sequenzialisieren &
auf denselben physikalischen Prozessor abbilden
(→ Multiplexing)

Bild 1.7
Logische
Neben-
läufigkeit



Regel 1.1
Neben-
läufigkeit

Entwerfe ein nebenläufiges System unabhängig davon, ob es mit physikalischer oder logischer Nebenläufigkeit realisiert wird.

1.7 Nebenläufigkeit auf Betriebssystem-Prozessebene

1.7.1 Sequenzielles System

Sequenzielles Programm :=

maschinell ausführbarer Algorithmus mit aufeinander folgenden Schritten.

Prozessor (*processor*) :=

aktives Organ, das ein Programm ausführen kann.

Umgebung, Kontext :=

Menge der Dinge, die bei einer Programmausführung manipulierbar sind, d.h. Register des Prozessors, Daten im Hauptspeicher, Betriebsmittel wie Drucker, Dateien usw., das Programm selbst.

Voraussetzungen bei einem sequenziellen Programm

- Anweisungen unteilbar ausgeführt
- Umgebung nur durch Programmausführung selbst verändert
- Ergebnisse einer Programmausführung zeitunabhängig, beliebig reproduzierbar

Folgerung

Sequenzielles Programm =
geschlossenes deterministisches System ohne externen Einfluss.

Bei einem nebenläufigen System gelten diese Voraussetzungen nicht!

1.7.2 Prozess und nebenläufiges System

Prozess (*process, task*) :=

Ausführung eines sequenziellen Programms auf einem Prozessor in einer Umgebung.

Prozesszustand (*process state*) :=

Schnappschuss seiner Umgebung zu einem Zeitpunkt.

Nebenläufiges System (*concurrent system*) :=

System mit ineinander verschränkten sequenziellen Abläufen.

Prozess = Einheit der sequenziellen Abläufe

Mehrere Prozesse laufen nebenläufig.

System mit **nebenläufigen** (parallelen, *concurrent*) **Prozessen**

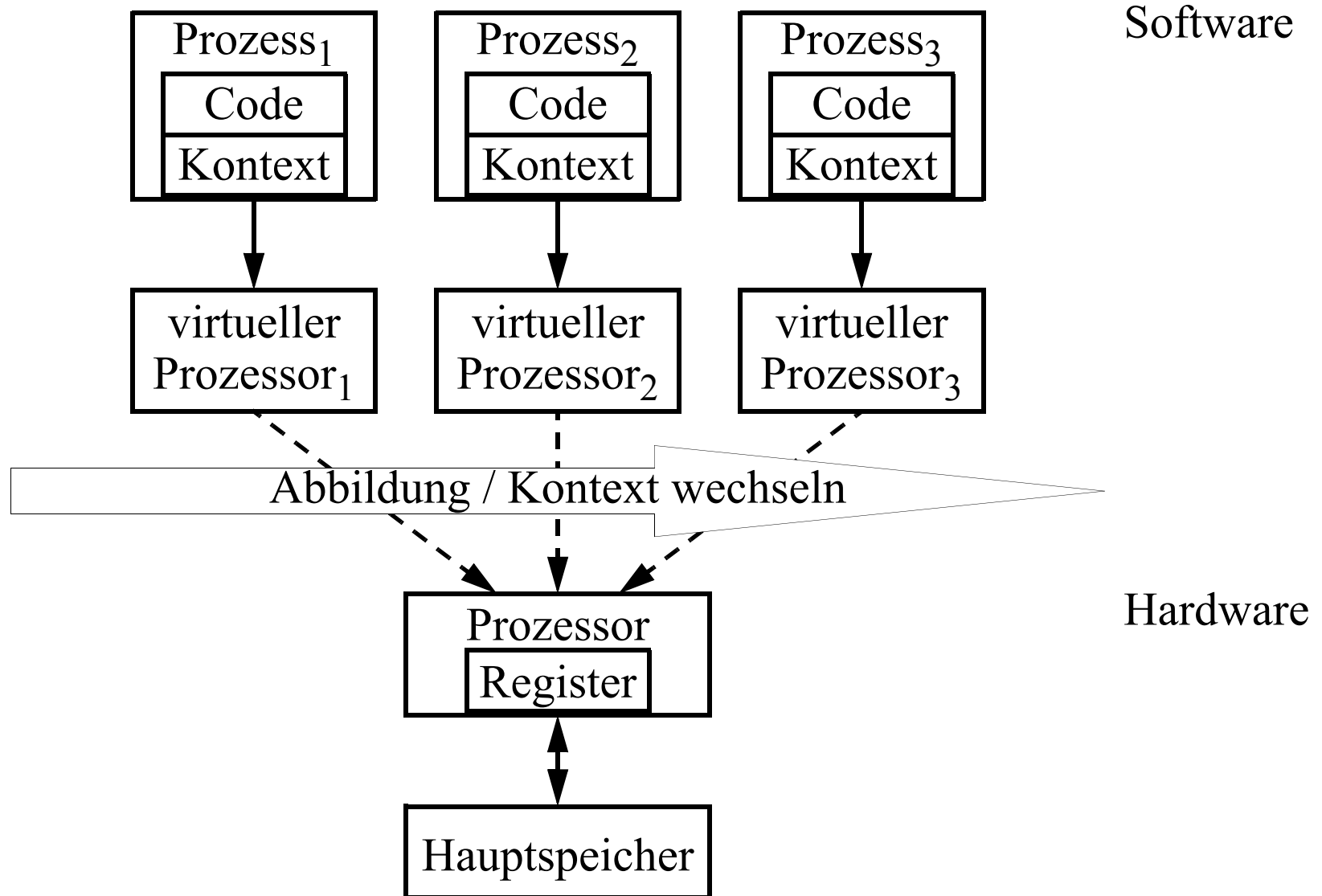
Mehrprozesssystem, -betrieb (*multiprocessing, multitasking*)

1.7.3 Eigenschaften von Prozessen

- Strukturierungseinheiten
 - Programm = statische Strukturierungseinheit
 - Prozess = dynamische Strukturierungseinheit
- Ein Programm kann von mehreren Prozessen ausgeführt werden.
- Ein Prozess kann nacheinander verschiedene Programme ausführen.
- Prozess $\neq$ geschlossenes System
- Ein Prozess kann beliebig gestoppt & fortgesetzt werden; Prozesszustand muss erhalten bleiben.
- Prozesse können
 - sich synchronisieren &
 - miteinander kommunizieren.

1.7.4 Prozessor-Multiplexing und -Virtualisierung

Bild 1.8
Prozessor-Multi-
plexing und
Virtualisie-
rung



Prozessor-Multiplexing :=

Prozessor wird nach gegebener Strategie zwischen ablaufbereiten Prozessen umgeschaltet;
ermöglicht Mehrprozessbetrieb auf Einprozessormaschine.

Virtueller Prozessor, Pseudoprozessor :=

per Software realisierte Kopie eines realen Prozessors;
Abstraktion des Begriffs „Prozessor“.

- Jeder Prozess läuft auf seinem zugeordneten virtuellen Prozessor.
- Prozess & virtueller Prozessor: zwei Sichtweisen
 - Prozess = Ablauf eines Programms
 - virtueller Prozessor = Akteur,
Verwaltungseinheit der BS-Komponente „Prozessverwaltung“

1.7.5 Gemeinsame Nutzung von Code und Daten

- *Code sharing*: Mehrere Prozesse führen gleichzeitig dasselbe Programm aus, z.B. Editor, Übersetzer.
- *Data sharing*: Prozesse kommunizieren über gemeinsame Daten.

Voraussetzung für gemeinsame Codenutzung

Code ist **eintrittsinvariant** (*reentrant*), d.h. getrennt in

- invarianten Codeteil aus sich nicht modifizierenden Befehlen & Konstanten
- varianten Datenteil.

Prozessglobale Daten :=

Daten, auf die mehrere Prozesse zugreifen können.

Prozesslokale Daten :=

Daten, die nur zu einem Prozess gehören.

1.7.6 Arten von Prozessen

Klassifizierung nach

- Aufgaben
- Implementationen

1.7.6.1 Klassifizierung nach Aufgaben

- **Systemprozess** führt Aufgabe des Betriebssystems aus.
- **Benutzerprozess** führt Anwendungsprogramm aus.

1.7.6.2 Klassifizierung nach Implementation

- **Schwergewichtiger (*heavyweight*) Prozess**
 - Beispiele: *process* in *Unix*, *task* in *MS-Windows*
 - eigener Adressraum, eigener Laufzeitkeller
 - ☺ ⇒ gegenseitig geschützt
 - ☹ Prozesswechsel teuer
 - Synchronisations-/Kommunikationsmechanismen
- **Leichtgewichtiger (*lightweight*) Prozess, Faden (*thread*)**
 - Beispiele: *thread* in *Unix*, *MS-Windows*
 - gemeinsamer Adressraum, gemeinsamer Laufzeitkeller
 - ☹ ⇒ gegenseitig ungeschützt
 - ☺ Threadwechsel billig
 - Synchronisationsmechanismen

1.8 Koordination von Prozessen

1.8.1 Konzepte der Koordination

Synchronisation :=

Prozesse steuern ihren (kausalen, zeitlichen) Ablauf relativ zueinander durch Austausch von Signalen.

Kommunikation :=

Prozesse tauschen untereinander Botschaften (nicht nur Signale) aus.

Koordination :=

Oberbegriff für Synchronisation & Kommunikation.

1.8.2 Zwecke der Koordination

Kooperation :=

Mehrere Prozesse arbeiten gemeinsam an einer Aufgabe.

Konkurrenz :=

Mehrere (nicht notwendig kooperierende) Prozesse bewerben sich um gemeinsam benutzbare, rare Betriebsmittel.

2

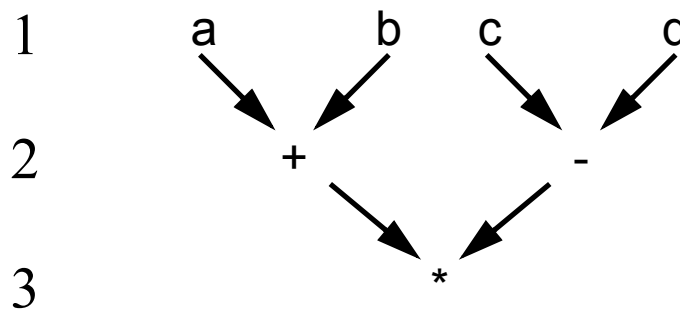
Nebenläufige Abläufe

2.1 Programmiersprachen-Ausdrucksebene

Ausdruck: $(a + b) * (c - d)$

Bild 2.1

Auswerten
eines
Ausdrucks



Kantorovic-Baum

- Teilausdrücke logisch nebenläufig auswertbar
- Sequenzialisierung durch Compiler

ADD a, b, D1

SUB c, d, D2

MUL D1, D2, D1

2.2 Programmiersprachen-Anweisungsebene

Prozedur zum Vertauschen der Werte zweier Variablen

- Sequenzielle Lösung

Code 2.1
Tauschen
sequenziell

```
swap (inout a, b) is
  local old_a
  do
    old_a := a; a := b; b := old_a
  end
```

☹ Sequenzialisierung durch Programmierer

- Nebenläufige Lösung

Code 2.2
Tauschen
nebenläufig

```
swap (inout a, b) is
  do
    (a, b) := (b, a)
  end
```

Kollektive Zuweisung (*Algol 68, Python*)

- erst rechte Seiten auswerten
- dann Werte linken Seiten zuweisen

☺ Sequenzielle / nebenläufige Implementation durch Übersetzer

2.3 Programmiersprachen-Programmeinheitsebene

Prozess (*process, task, thread*) :=

Programmeinheit, die nebenläufig zu anderen Programmeinheiten desselben Programms ausführbar ist.

Jeder Prozess eines Programms hat zur Laufzeit eigenen Ablaufzustand.

Prozesse ähneln syntaktisch in

- | | |
|-------------------------------|---|
| ● prozeduralen Sprachen | Prozeduren |
| ● modularen Sprachen | Modulen |
| ● objektorientierten Sprachen | Klassen
(subjektorientierte Programmierung). |

2.3.1 Einfaches prozedurähnliches Prozesskonstrukt

Code 2.3

Prozesse

global declarations

single is process

einzelnes Prozessexemplar

local

local declarations

do

statements

end -- single

multiple (self : 1 .. n) is process

Prozesstyp mit n Exemplaren

local

local declarations

do

statements

end -- multiple

single & multiple (1) bis multiple (n)

- werden implizit gestartet &
- arbeiten nebenläufig, d.h. laufen gleichzeitig, parallel.

2.3.2 Prozedur und Prozess

Tabelle 2.1

Eigen-
schaften
von
Prozeduren
&
Prozessen

Prozedur	Prozess
muss explizit aufgerufen werden	kann implizit gestartet werden
wird synchron aufgerufen, d.h. ● Aufrufer gibt Kontrolle an Aufgerufenen & ● wartet auf Rückkehr der Kontrolle vom Aufgerufenen	wird asynchron aufgerufen, d.h. ● Aufrufer behält seinen Kontrollfluss & ● wartet nicht auf Rückkehr der Kontrolle vom Aufgerufenen
derselbe Kontrollfluss durchläuft Aufrufer & Aufgerufenen	verschiedene Kontrollflüsse durchlaufen Aufrufer & Aufgerufenen
am Ende des Anweisungsteils kehrt Kontrollfluss an Aufrufstelle zurück	am Ende des Anweisungsteils kann Aufrufer benachrichtigt werden, muss aber nicht

2.3.3 Modul, Klasse, Prozess

- Modul hat i.A. mehrere Schnittstellenoperationen.
- Prozess kann mehrere **Einstiege** (*entry*) haben.
- Klasse hat Typeigenschaft.
- Von **Prozesstyp** lassen sich beliebig viele **Prozessexemplare**
 - statisch vereinbaren & ggf.
 - dynamisch erzeugen &
 - starten.

2.3.4 Kommunikationskonzepte

Prozesse können miteinander kommunizieren durch

- gemeinsame globale (= nicht lokale) Daten
- synchronen Parametertausch
- asynchronen Botschaftenaustausch.

Unabhängige Prozesse kommunizieren nicht & beeinflussen sich nicht.

2.3.5 Implementationen programmiersprachlicher Prozesse

Gegeben

Nebenläufiges Programm einer Hochsprache mit Prozesskonstrukt

Alternativen

- Jeder Prozess des Programms wird auf einen
 - Prozess
 - Threaddes Betriebssystems abgebildet.
- Das Programm wird von einem Prozess des Betriebssystems ausgeführt. Das Laufzeitsystem der Sprache verwaltet die Prozesse des Programms.

2.4 Abriss der historischen Entwicklung

- 1962 Petri-Netz (Petri)
Abstrakter Modellierungsformalismus für Nebenläufigkeit
- 1965 Semaphore (Dijkstra)
Synchronisationskonstrukt, mit dem viele Synchronisationsprobleme lösbar sind
- 1966 *PL/I* (IBM)
nebenläufiger Prozeduraufruf (**task**)
Synchronisationskonstrukt **event**: unbrauchbar, um selbst einfache Synchronisationsprobleme zu lösen
- 1967 *Simula 67* (Dahl, Nygaard)
Koroutine

- 1969 *Algol 68* (van Wijngaarden et al.)
 Nebenläufigkeitsanweisung, kollektive Zuweisung, Semaphor
- 1972 Bedingte kritische Region (Brinch Hansen, Hoare)
- 1973-74 Monitor (Hoare, Brinch Hansen)
 imperativ synchronisierter ADT
- 1974 Pfadausdruck (Campbell, Habermann)
 deklarativ synchronisierter ADT
- Unix* (Ritchie, Thompson)
 Prozess, *fork*, *exit* & *wait*, Signal, *pipe*
- 1975 *Concurrent Pascal* (Brinch Hansen)
 = *Pascal* + Prozess + Klasse (*Simula 67*) + Monitor
- Bewachte Anweisung (Dijkstra)

- 1978 *CSP - Communicating Sequential Processes* (Hoare)
Nebenläufigkeitsanweisung, Prozess
synchroner Botschaftenaustausch
- 1979 *Mesa* (Xerox PARC, Mitchell)
Prozess, Monitor
- 1980 *PEARL - Process and Experiment Automation Real-time Language*
Echtzeitprogrammierung, Prozess, Signal, Semaphor, Riegel
Path Pascal (Kolstad, Campbell)
= *Pascal* + Prozess + ADT + Pfadausdruck
Occam (Inmos, May)
Abkömmling von *CSP*, Systemsprache für Transputer
- 1982 *Modula-2* (Wirth)
Koroutine, Prozess (Bibliothek), Monitor

- 1983 *Ada* (DoD, Ichbiah)
 task analog zu **package**
 Rendezvous-Konzept: synchroner Parametertausch,
 bewachte Anweisung
- 1984 *CHILL - CCITT High Level Language* (CCITT)
 Prozess (**PROCESS**) ähnlich Prozedur
 Monitor (**REGION**), Ereignis (**EVENT**)
 asynchrone Kommunikation mit Signal (**SIGNAL**), Puffer (**BUFFER**)
- 1995 *Java* (Sun, Gosling et al.)
 Thread als Klasse, Runnable als Schnittstelle
 Anweisung synchronized, Methoden wait & notify, Monitor

2000

C# (Microsoft, Hejlsberg et al.)

nebenläufiger Methodenaufruf mit *Delegate*-Klasse ThreadStart &
Klasse Thread

Anweisung lock, Klassen Interlock, Monitor

2.5 Unabhängige Threads in Java

Zwei Möglichkeiten, Threads zu definieren:

- Erweitern der Klasse Thread
- Implementieren der Schnittstelle Runnable

2.5.1 Klasse java.lang.Thread

Code 2.4 `public class Thread {`

```
Java:      public Thread ();
Thread     public Thread (Runnable r);
           public Thread (String name);
           public Thread (Runnable r, String name);
           ...
           public final void setName (String name);
           public final String getName ();
           ...
           public void start () throws IllegalStateException;
           public void run ();
           ...
           public static void sleep (long millis)
                                   throws InterruptedException;
           ...
           public final boolean isAlive ();
           public final void join () throws InterruptedException;
           ...
        }
```

2.5.2 Erweitern der Klasse Thread

Code 2.5

Java:
Simple-
Thread

```
public class SimpleThread extends Thread {  
    public SimpleThread (String name) {  
        super(name);           Aufruf der Basisklassenversion  
    }  
  
    public void run () {  
        System.out.println(getName());  
    }  
  
    public static void main (String[] args) {  
        SimpleThread thread =           Vereinbaren &  
            new SimpleThread("SimpleThread");   dynamisches Erzeugen  
                                                eines SimpleThread-Objekts  
        thread.start();           Starten des SimpleThread-Objekts,  
                                das dann nebenläufig  
                                seine run-Methode ausführt  
    }  
}
```

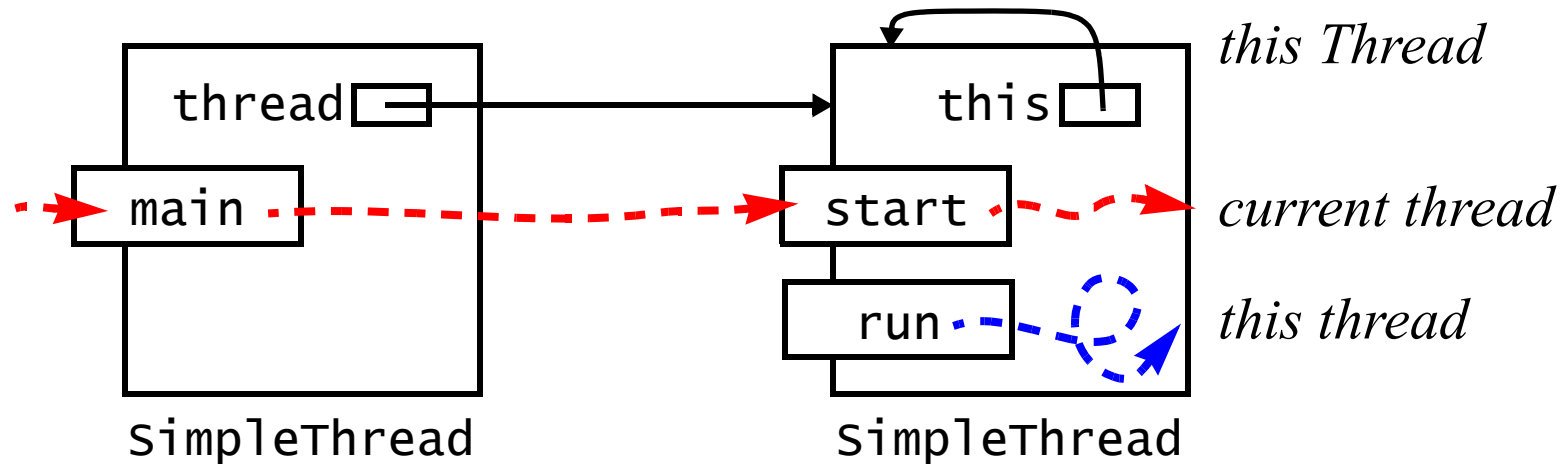
Bemerkungen

- *Java*-Programm läuft in einem BS-Prozess.
- Alle Threads des Programms laufen innerhalb des Prozesses.
- Prozess terminiert, nachdem alle Threads terminiert haben.
- Thread erweitern
 - ☺ möglich bei Klassen, die keine andere Klasse erweitern
 - ☹ nicht möglich bei Erweiterungen anderer Klassen
(da kein mehrfaches Erben möglich)

Beachten

- Von jedem Thread-Objekt darf `start` höchstens einmal aufgerufen werden (Wiederstart verboten)!
- $\Rightarrow$ Zu jedem Thread-Objekt gibt es höchstens einen Thread, der dessen `run`-Methode ausführt.

Bild 2.2
Thread-
Objekte &
Threads



Unterscheiden

- **aktueller Thread** (*current thread*): Thread, der (irgend)eine Methode (irgend)eines Objekts, z.B. `start` eines Thread-Objekts, ausführt
- **aktuelles Thread-Objekt** (*this Thread*): das Thread-Objekt, dessen Methode aufgerufen wurde
- **Thread des aktuellen Thread-Objekts** (*this thread*): Thread des Thread-Objekts, dessen Methode aufgerufen wurde

Variante mit Start beim Erzeugen

Code 2.6

Java:
Simple-
Thread

```
public class SimpleThread extends Thread {  
    public SimpleThread (String name) {  
        super(name);           Aufruf der Basisklassenversion  
                                Starten des SimpleThread-Objekts,  
                                das dann nebenläufig  
                                seine run-Methode ausführt  
    }  
    public void run () {  
        System.out.println(getName());  
    }  
    public static void main (String[] args) {  
        SimpleThread thread =          Vereinbaren &  
            new SimpleThread("SimpleThread"); dynamisches Erzeugen  
                                                eines SimpleThread-Objekts,  
                                                das sofort gestartet wird  
    }  
}
```

2.5.3 Schnittstelle java.lang.Runnable

Vollständige Schnittstelle

Code 2.7 `public interface Runnable {`
Java: `public abstract void run ();`
Runnable `}`

2.5.4 Implementieren der Schnittstelle Runnable

Code 2.8

Java:

Simple-
Runner

```
public class SimpleRunner extends SomeClass
                                implements Runnable {
    protected String name;
    public SimpleRunner (String name) {
        this.name = name;
    }
    public void run () {
        System.out.println(name);
    }
    public static void main (String[] args) {
(1)        SimpleRunner runner = new SimpleRunner("SimpleRunner");
(2)        Thread thread = new Thread(runner);
(3)        thread.start();
    }
}
```

Bemerkungen

- (1) Vereinbaren & dynamisches Erzeugen eines `SimpleRunner`-Objekts
- (2) Vereinbaren & dynamisches Erzeugen eines `Thread`-Objekts, dem das `SimpleRunner`-Objekt übergeben wird
- (3) Starten des `Thread`-Objekts, das dann nebenläufig die `run`-Methode des `SimpleRunner`-Objekts ausführt

☺ Möglich bei allen Klassen

Variante mit Start beim Erzeugen

Code 2.9

Java:

Simple-Runner

```
public class SimpleRunner extends SomeClass
                                implements Runnable {

    protected String name;

    public SimpleRunner (String name) {
        this.name = name;
        Thread thread = new Thread(this);
        thread.start();
    }

    public void run () {
        System.out.println(name);
    }

    public static void main (String[] args) {
        SimpleRunner runner = new SimpleRunner("SimpleRunner");
    }
}
```

2.5.5 Beispiel: Mehrere iterierende Threads

Code 2.10 Java:
Iterating-Thread

```
public class IteratingThread extends Thread {  
    protected static int numberOfThreads = 3;  
    protected static int numberOfIterations = 100;  
    public IteratingThread (String name) {  
        super(name);  
    }  
  
    protected Thread newThread (int number) {    Fabrikmethode  
        IteratingThread thread =  
            new IteratingThread("IteratingThread " + number);  
        return thread;  
    }  
  
    protected void parseMoreArgs (String[] args) {}  
    protected void doIt () {}  
}
```

```

private IteratingThread (String[] args) {
    super("Root Thread");
    try {
        if (args.length > 0) {
            numberOfThreads = Integer.parseInt(args[0]);
            if (args.length > 1) {
                numberOfIterations = Integer.parseInt(args[1]);
                if (args.length > 2) {
                    parseMoreArgs(args);
                }
            }
        }
    } catch (NumberFormatException e) {
        System.out.println(e.toString() + " int args exp.");
    }
    for (int i = 0; i < numberOfThreads; ++i) {
        Thread thread = newThread(i);
        thread.start();
    }
}

```

Schablonenmethode

```
public void run () {  
    for (int i = 0; i < numberOfIterations; ++i) {  
        System.out.println(getName() + ": " + i);  
        doIt();  
    }  
}  
  
public static void main (String[] args) {  
    IteratingThread thread = new IteratingThread(args);  
}  
}
```

Schablonenmethode

2.5.6 Beispiel: Mehrere iterierend schlafende Threads

Code 2.11 Java:
Iterating-Sleeper

```
public class IteratingSleeper extends IteratingThread {  
    protected static long sleepingTime = 0;  
    public IteratingSleeper (String name) {  
        super(name);  
    }  
  
    protected Thread newThread (int number) {      Fabrikmethode  
        IteratingSleeper thread =  
            new IteratingSleeper("IteratingSleeper " + number);  
        return thread;  
    }  
  
    protected void parseMoreArgs (String[] args) {  
        try {  
            sleepingTime = Integer.parseInt(args[2]);  
        } catch (NumberFormatException e) {  
            System.out.println(e.toString() + " int args exp.");  
        }  
    }  
}
```

```
protected void doIt () {  
    try {  
        sleep(sleepingTime);  
    } catch (InterruptedException e) {  
        System.out.println(e.toString() + " ItrExc caught");  
    }  
}
```

Übung 2.1 Laden Sie obige Beispiele von

Unab-
hängige
Threads

[ftp://studinf.reutlingen-university.de/MKI/Hug/Lehrveranstaltungsmaterial/I3/
Java/Source/](ftp://studinf.reutlingen-university.de/MKI/Hug/Lehrveranstaltungsmaterial/I3/Java/Source/)

herunter & experimentieren Sie damit!

2.6 Verzweigen und Zusammenführen

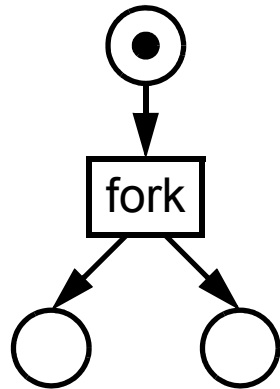
2.6.1 Petri-Netze

Carl Adam Petri: *Kommunikation mit Automaten*
TH Darmstadt, Uni Bonn (1962) Dissertation

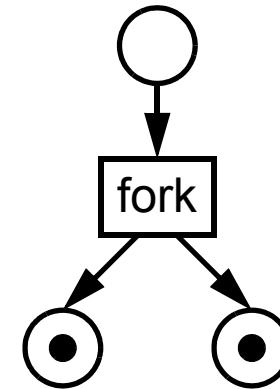
2.6.1.1 Verzweigen

Bild 2.3
Fork

vorher

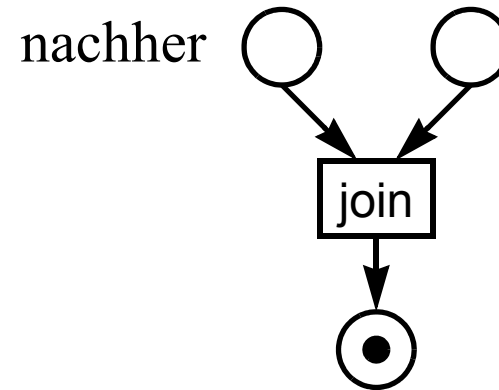
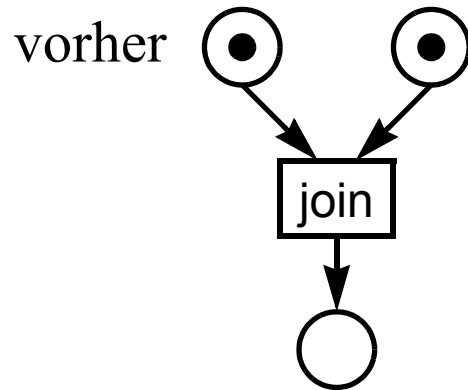


nachher



2.6.1.2 Zusammenführen

Bild 2.4
Join



2.6.1.3 Verzweigen und Zusammenführen

Bild 2.5

Fork & Join

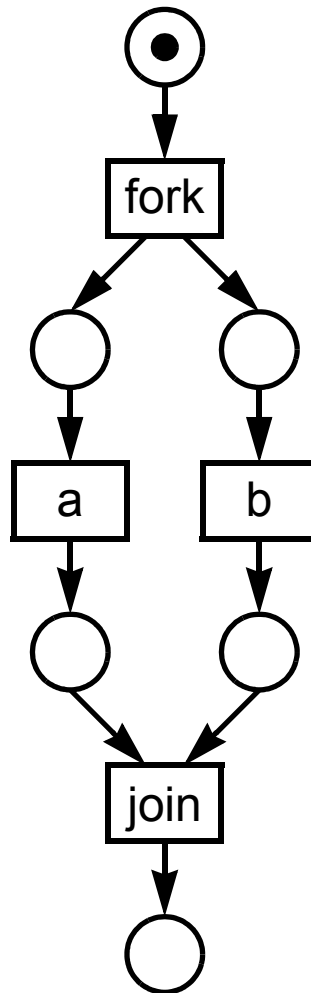


Bild 2.6
Fork & Join

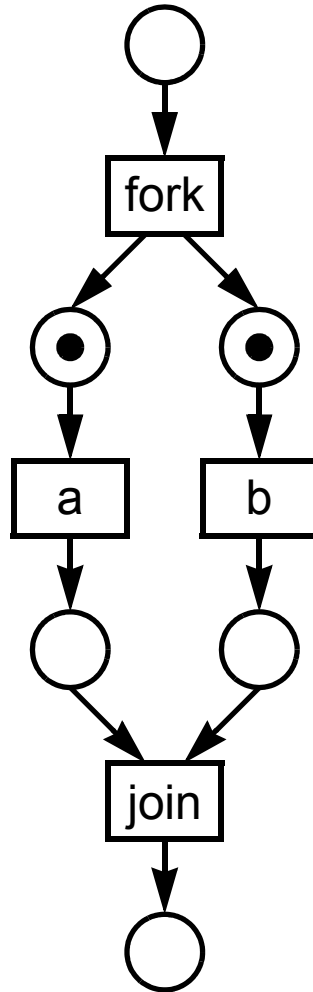


Bild 2.7
Fork & Join

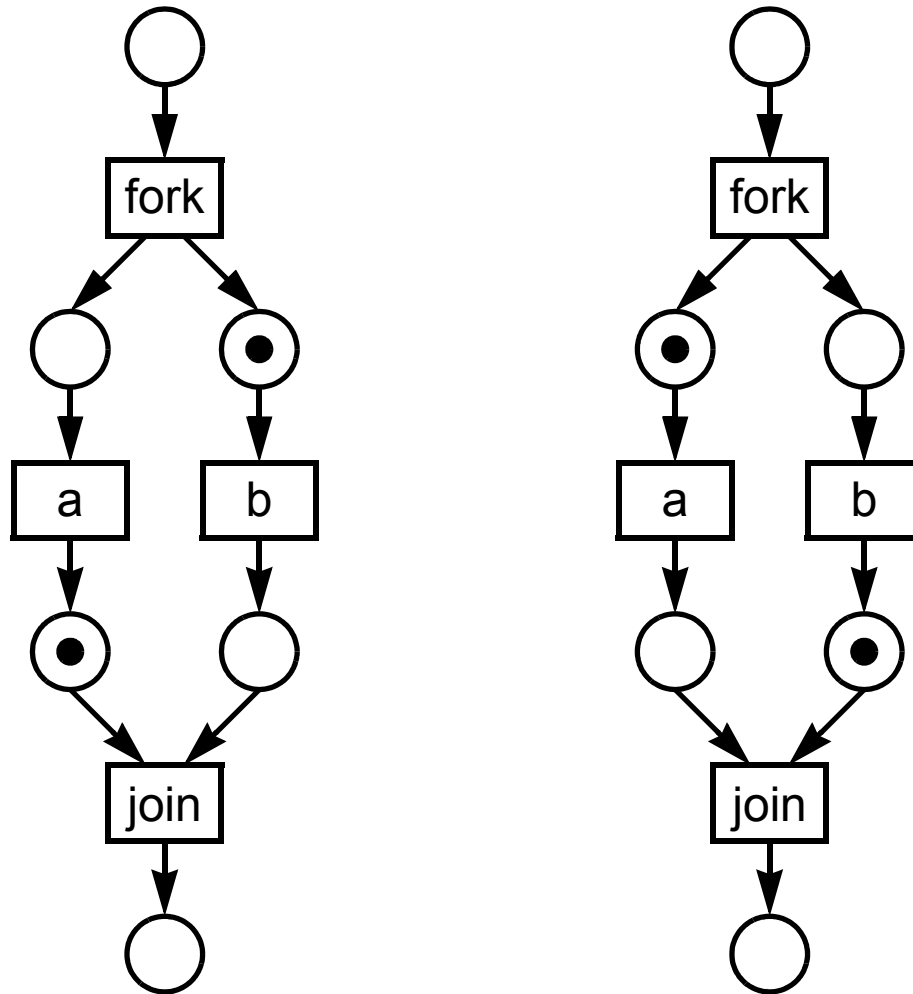


Bild 2.8
Fork & Join

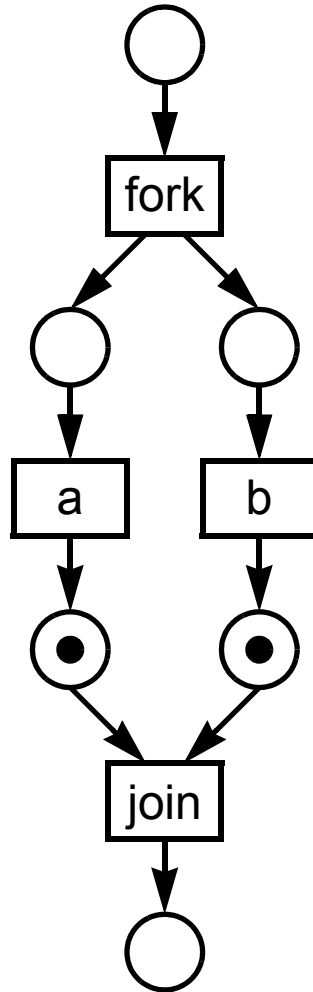
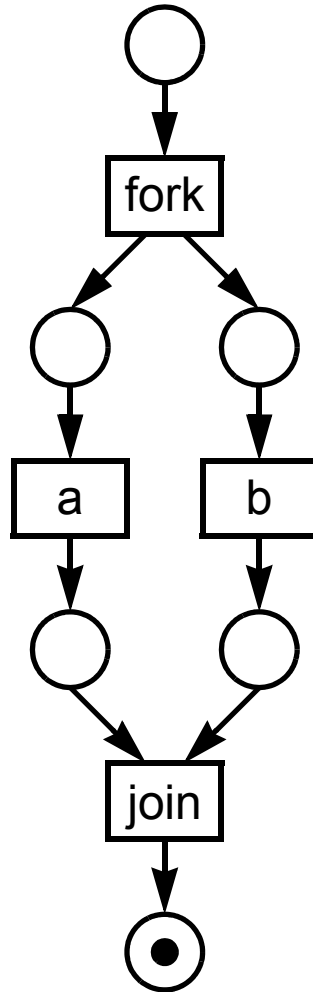


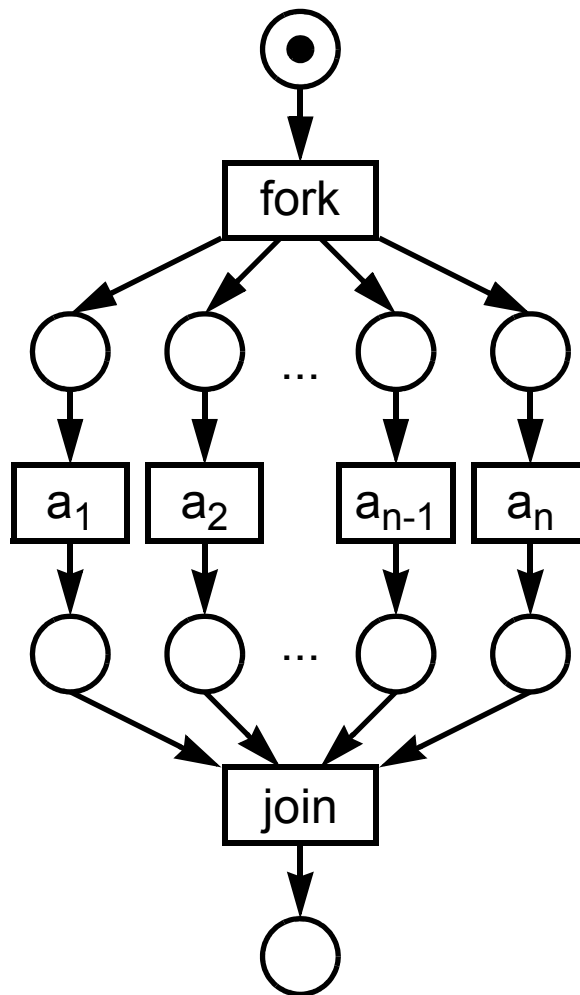
Bild 2.9
Fork & Join



2.6.1.4 Symmetrisches Verzweigen und Zusammenführen

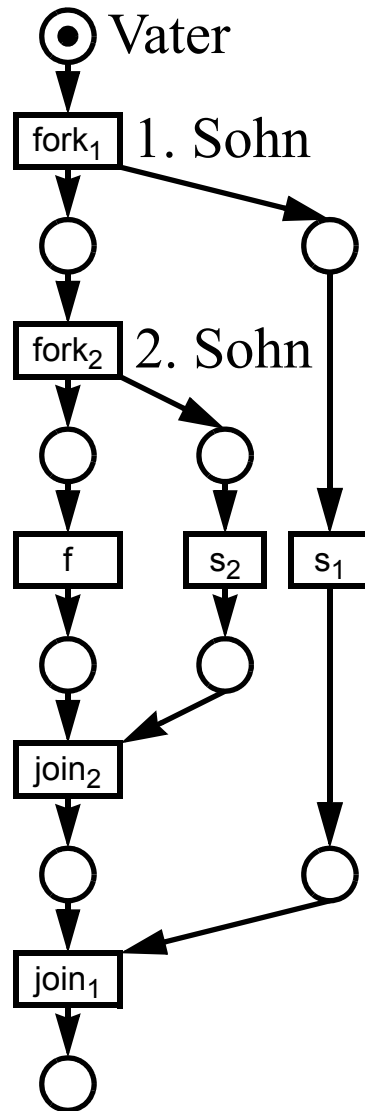
Bild 2.10

Mehr-
faches
Fork & Join



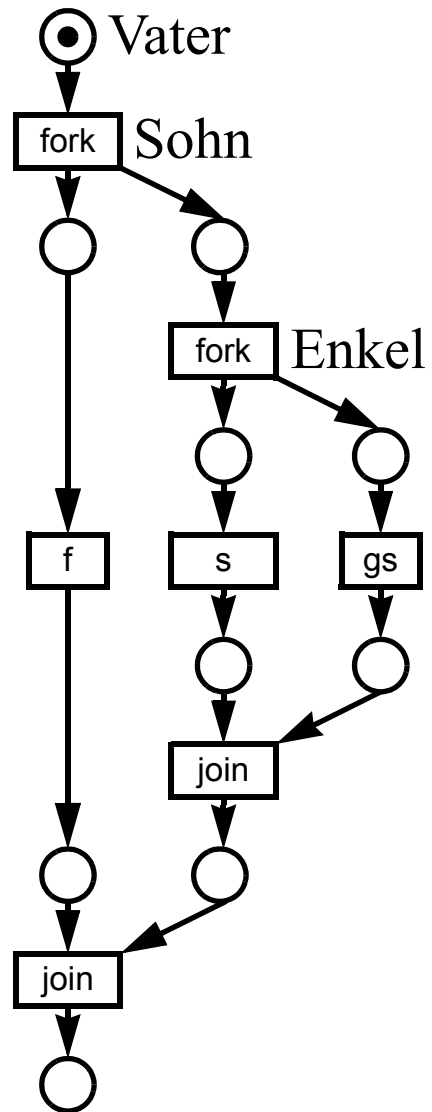
2.6.1.5 Geschachteltes Verzweigen und Zusammenführen

Bild 2.11
Geschach-
teltes
Fork & Join



Äquivalentes Petri-Netz, anders interpretiert

Bild 2.12
Geschach-
teltes
Fork & Join



2.6.2 Nebenläufigkeitsanweisungen

2.6.2.1 Symmetrisches Verzweigen und Zusammenführen

conc $a_1 \parallel a_2 \parallel \dots \parallel a_{n-1} \parallel a_n$ **end**

2.6.2.2 Geschachteltes Verzweigen und Zusammenführen

```
conc
   $a_1$ 
  || conc
     $a_2$ 
    ||  $\dots$ 
    ||  $a_{n-1}$ 
    ||  $a_n$ 
  end
end
```

Semantik äquivalent zu 2.6.2.1

2.6.3 Unix

2.6.3.1 Systemaufrufebene

Vater- und Sohn-Prozess

int pid, status

Fork

if (pid = fork())

{

/ Vater-Zweig */*

do_fathers_job;

pid = wait(&status);

}

else

{

/ Sohn-Zweig */*

do_sons_job;

Join

exit(status);

}

2.6.3.2 Shell-Ebene

p x & q y & wait

Fork

- Kommandos *p*, *q* werden wegen „&“ als Hintergrundprozesse ausgeführt.

Join

- *wait* ist ein Kommando, das
 - als Vordergrundprozess ausgeführt wird &
 - als solcher wartet, bis alle Hintergrundprozesse terminiert haben.

2.6.4 Java

2.6.4.1 Vater und Sohn

Code 2.12

```
public class FatherSonThread extends Thread {  
    public FatherSonThread (String name) {super(name);}  
    public void run () {  
        System.out.println(getName() + " started & dying");  
    }  
  
    public static void main (String[] args) {  
        FatherSonThread son = new FatherSonThread("Son");  
        son.start();  
        if (son.isAlive()) {...} else {...}  
        try {  
            son.join();  
        } catch (InterruptedException e) {  
            System.out.println(e.toString() + " ItrExc caught");  
        }  
    }  
}
```

Java:
Father-
Son-
Thread

Fork

Join

2.6.4.2 Vater und zwei Söhne

Code 2.13

```
public class Father2SonsThread extends Thread {
    public Father2SonsThread (String name) {super(name);}
    public void run () {
        System.out.println(getName() + " started & dying");
    }

    public static void main (String[] args) {
        Father2SonsThread s1 = new Father2SonsThread("Son 1");
        Father2SonsThread s2 = new Father2SonsThread("Son 2");
        s1.start();
        s2.start();
        ...
        try {s1.join();} catch(...) {...}
        ...
        try {s2.join();} catch(...) {...}
        ...
    }
}
```

Java: public Father2SonsThread (String name) {super(name);}
Father- public void run () {
2Sons- System.out.println(getName() + " started & dying");
Thread }

Fork s1.start();
Fork s2.start();

Join ...
 try {s1.join();} catch(...) {...}

Join ...
 try {s2.join();} catch(...) {...}

 ...
 }

2.6.4.3 Vater, Sohn und Enkel

Code 2.14

```
public class FatherSonGrandsonThread extends Thread {
    public FatherSonGrandsonThread (String name) {
        super(name);
    }
    public void run () {
        if (getName().equals("Son")) {
            FatherSonGrandsonThread grandson =
                new FatherSonGrandsonThread("Grandson");
            grandson.start();
            ...
            try {
                grandson.join();
            } catch (InterruptedException e) {
                System.out.println(e.toString() + " ItrExc caught");
            }
            ...
        }
    }
}
```

Java:
Father-
Son-
Grandson-
Thread

Fork

Join

```

    public static void main (String[] args) {
        FatherSonGrandsonThread son =
            new FatherSonGrandsonThread("Son");
Fork    son.start();
        ...
Join    try {
            son.join();
        } catch (InterruptedException e) {
            System.out.println(e.toString() + " ItrExc caught");
        }
        ...
    }
}

```

2.6.5 BlackBox

Einfacher Dienst zum Ausführen von Kommandos im Hintergrund

Code 2.15 DEFINITION Services;

CP: TYPE
Services Action = POINTER TO ABSTRACT RECORD
(a: Action) Do-, NEW, ABSTRACT
END;

PROCEDURE DoLater (a: Action; notBefore: LONGINT);
END Services.

- Erweitern der abstrakten Klasse Services.Action zu MyAction
(analog zum Implementieren der Runnable-Schnittstelle in Java)
- Redefinieren der Prozedur MyAction.Do
(analog zu MyRunnable.run)
- Registrieren eines Objekts obj von MyAction durch
Services.DoLater (obj, Services.now)
(analog zu thread-Objekt erzeugen & thread.start aufrufen)

Bemerkungen

- ☺ einfache Probleme schnell & leicht zu lösen
- ☺ beliebige Wiederstarts möglich
- ☺ oft keine Synchronisationsprobleme,
da Kommandos unteilbar ausgeführt

- ☹ kein Zusammenführen

Ersatzlösungen

- wiederholtes Abfragen (*polling*)
- Zurückrufen (*callback*)

- ☹ keine Synchronisationsmechanismen

2.6.6 Beispiel: Sortiert-Prüfung

Gegeben: Reihung vergleichbarer Elemente

Gefragt: Ist die Reihung sortiert?

Indextyp
Elementtyp

2.6.6.1 Rekursive Spezifikation durch Vertrag

Code 2.16

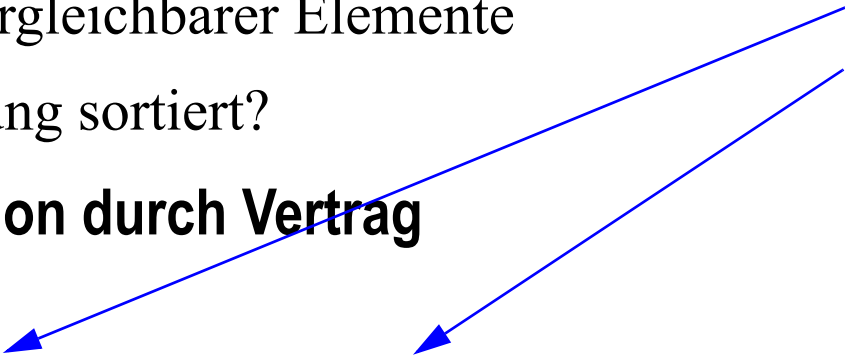
Sortiert
spezifiziert

```
sorted
  (in a : ARRAY [INDEX -> INTEGER, ITEM -> COMPARABLE];
   in lower, upper : INTEGER) : BOOLEAN is

  require
    a.lower <= lower
    lower <= upper
    upper <= a.upper

  ensure
    result = (lower + 1 >= upper and a [lower] <= a [upper]) or
              (lower + 1 < upper and
               sorted (a, lower, (lower + upper) div 2) and
               sorted (a, (lower + upper) div 2, upper))

  end
```



2.6.6.2 Implementation mit Nebenläufigkeitsanweisung

Code 2.17

Sortiert
implemen-
tiert

```
sorted
  (in a : ARRAY [INDEX -> INTEGER, ITEM -> COMPARABLE];
   in lower, upper : INTEGER) : BOOLEAN is
  require
    a.lower <= lower and lower <= upper and upper <= a.upper
  local
    mid      : INTEGER := (lower + upper) div 2
    left, right : BOOLEAN
  do
    if lower + 1 >= upper then
      result := a [lower] <= a [upper]
    else
      conc
        left := sorted (a, lower, mid)
        || right := sorted (a, mid, upper)
      end
      result := left and right
    end
  end
```

Fork

Join

2.6.6.3 Implementationen in Java und BlackBox

Siehe

<ftp://studinf.reutlingen-university.de/MKI/Hug/Lehrveranstaltungsmaterial/I3/Java/Threads/Source/SortedChecker.java>

&

<ftp://studinf.reutlingen-university.de/MKI/Hug/Lehrveranstaltungsmaterial/I1/BlackBox-Informatik-Hug-08-09-WS/I3/Mod/SortedCheckersPolling.odc>

Übung 2.2 Laden Sie obige Beispiele von

Verzweigte
& zusam-
mengeführ-
te Threads

<ftp://studinf.reutlingen-university.de/MKI/Hug/Lehrveranstaltungsmaterial/I3/Java/Threads/Source/>

herunter & experimentieren Sie damit!

3.1 Zwecke der Synchronisation

- **Kooperationssynchronisation**

Prozesse P , Q bearbeiten Teilaufgaben einer Aufgabe & Prozess Q muss warten, bis Prozess P eine Teilaufgabe erledigt hat, deren Ergebnis Q benötigt.

Beispiel

$P_1, \dots, P_n$ kompilieren Übersetzungseinheiten,
 Q bindet die erzeugten Objektcodes zusammen.

- **Konkurrenzsynchronisation**

Prozesse P , Q benötigen ein exklusiv benutzbares Betriebsmittel & Prozess Q muss warten, bis Prozess P das Betriebsmittel freigibt.

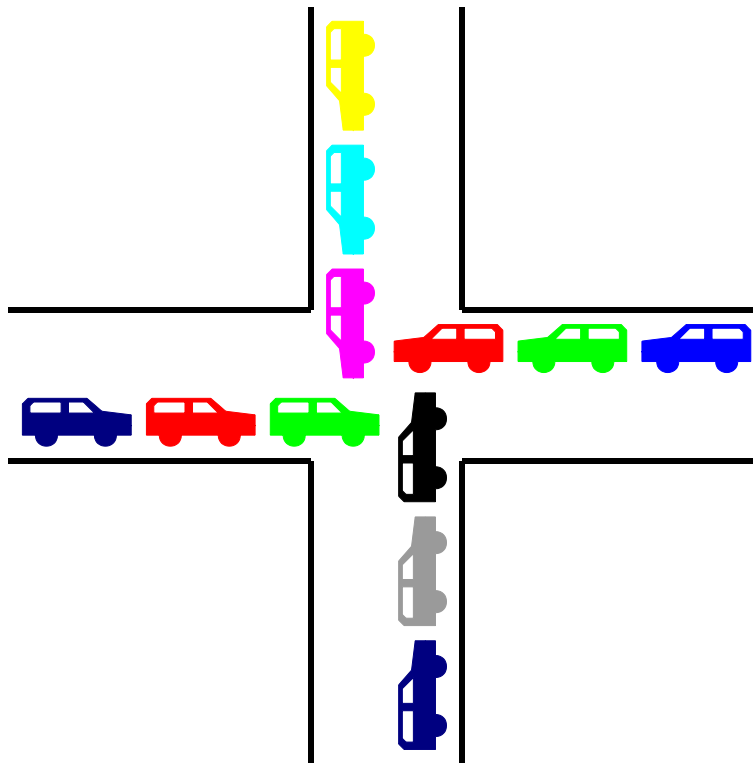
Beispiel

P belegt Drucker, Q fordert Drucker an.

3.2 Probleme bei Synchronisation

- Verklemmung (*deadlock*)

Bild 3.1
Verklemmung im
Straßenverkehr

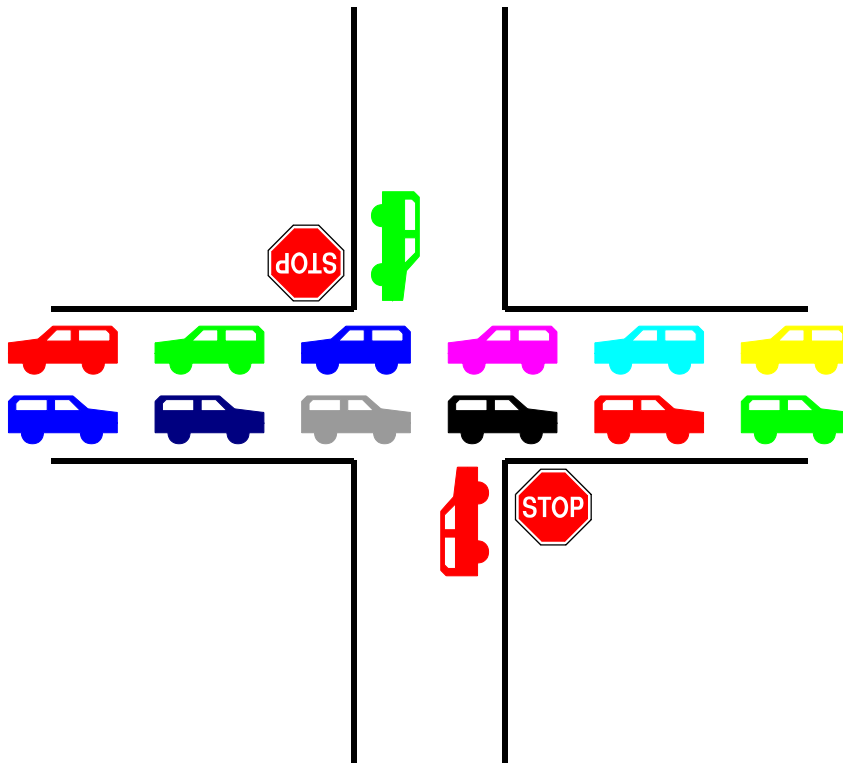


Zirkulares Warten: Jeder wartet auf eine Bedingung, die ein anderer herstellen soll, aber nicht kann, solange er selbst warten muss.

- **Aushungerung** (*starvation*)

Bild 3.2

Aushungerung im
Straßen-
verkehr



Unendlich langes Warten abhängig vom Verhalten der Anderen

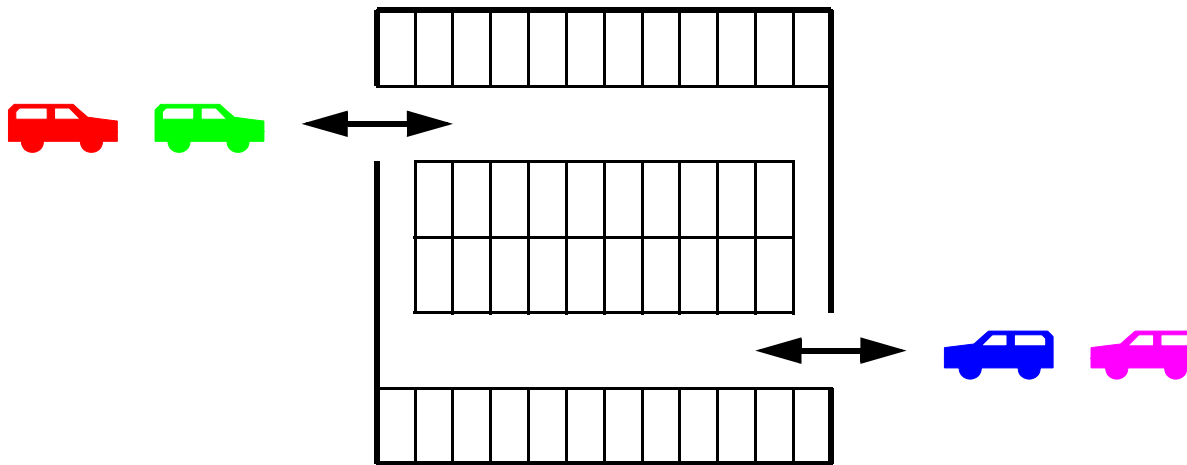
fair := aushungerungsfrei

3.3 Beispiel: Ein Parkhaus-Problem

Gegeben

Parkhaus, mehrere Zufahrten, Kapazität 100 Pkw

Bild 3.3
Parkhaus

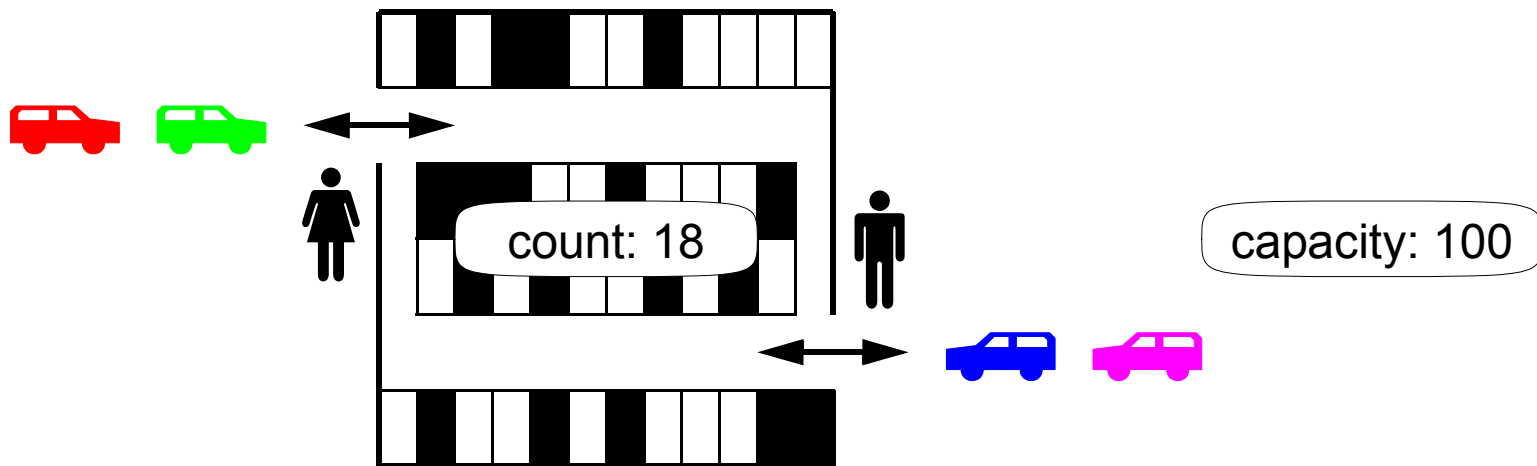


Gesucht

Einfahrtregelung so, dass Kapazität nicht überschritten

Lösungskonzept 1

Bild 3.4
Parkhaus
mit Zähler
&
Aufpassern



- Zähler für parkende Pkw im Parkhaus
- an jedem Tor ein „Aufpasser“, der Zähler aktualisiert
- Invarianten:
 $0 \leq \text{count} \leq \text{capacity}$
 $\text{count} = \text{tatsächliche Anzahl der Pkw im Parkhaus}$

3.3.1 Lösungsansatz 1.1

Code 3.1
Parkhaus
1.1

```
capacity : NATURAL is const 100          prozessglobale Vereinbarungen
count    : NATURAL is init 0              Anzahl parkender Pkw

watchdog (self : 1 .. number_of_gates) is process
do
  (2) loop
  (1)   if car enters car park then
  (3)     count := count + 1
  (1)   elseif car leaves car park then
  (3)     count := count - 1
        end
      end
    end -- watchdog
```

Probleme

- (1) Überlastschutz fehlt!
- (2) **Aktives Warten** (*busy waiting*) ist ineffizient!
- (3) Was passiert beim Zugriff auf den Zähler?

3.3.2 Lösungsansatz 1.2 mit Überlastschutz

Code 3.2
Parkhaus
1.2

```
watchdog (self : 1 .. number_of_gates) is process
do
  loop
    if car enters car park then
      count := count + 1
      if count = capacity then
        traffic_lights.set (red)          gates.close
      end
    elseif car leaves car park then
      count := count - 1
      if count = capacity - 1 then
        traffic_lights.set (green)       gates.open
      end
    end
  end
end -- watchdog
```

☺ Kooperationssynchronisation: Aufpasser signalisieren Pkw-Fahrern die Auslastung mit Ampeln oder Schranken.

3.3.3 Lösungsansatz 1.3 ohne aktives Warten

Code 3.3
Parkhaus
1.3

```
watchdog (self : 1 .. number_of_gates) is process
do
  loop
    if car enters car park then
      count := count + 1
      ...
    elseif car leaves car park then
      count := count - 1
      ...
    else
      wait
    end
  end
end -- watchdog
```

- Wie wird das Warten realisiert? Aufpasser schlafen.
- Kooperationssynchronisation: Ankommende Pkw-Fahrer müssen mit Aufpassern kommunizieren, z.B. sie durch Hupen aufwecken.

3.3.4 Lösungsansatz 1.1 auf Maschinenbefehlebene

Code 3.4
Parkhaus
1.1MBE

watchdog (self : 1 .. number_of_gates) is process

local

datareg : NATURAL

do

loop

if *car enters car park* **then**

LOAD count, datareg

++count

ADD datareg, 1

STORE datareg, count

elseif *car leaves car park* **then**

LOAD count, datareg

--count

SUB datareg, 1

STORE datareg, count

end

end

end -- watchdog

☹ Auch Dreiadressbefehle erfordern zwei Speicherzugriffe!

ADD count, 1, count

3.3.5 Exemplarische Abläufe

Im Parkhaus sind 18 Pkw, West: Pkw kommt an, Ost: Pkw geht ab.

Tabelle 3.1 Nebenläufige Ablaufverfolgung

watchdog west		watchdog east		count
instructions	datareg	instructions	datareg	
	don't care		don't care	18
LOAD count, datareg				
		LOAD count, datareg		
ADD datareg, 1				
		SUB datareg, 1		
STORE datareg, count				
		STORE datareg, count		

Ist das Ergebnis korrekt?

Tabelle 3.2 Nebenläufige Ablaufverfolgung

watchdog west		watchdog east		count
instructions	datareg	instructions	datareg	
	don't care		don't care	18
		LOAD count, datareg		
LOAD count, datareg				
		SUB datareg, 1		
ADD datareg, 1				
		STORE datareg, count		
STORE datareg, count				

Ist das Ergebnis korrekt?

3.3.6 Zwischenbilanz

- Problem (3) „Was passiert beim Zugriff auf den Zähler?“ ist, die **Konsistenz** des Zählers unter nebenläufigen Zugriffen zu erhalten.
- **Konsistenzbedingung** = Invariante
- **Invariante** des Zählers:
$$\text{count} = \text{tatsächliche Anzahl der Pkw im Parkhaus}$$
- **Verallgemeinerung**: Problem ist, die **Konsistenz** prozessglobaler Daten unter nebenläufigen Zugriffen zu erhalten.
- **Forderung**: Wir brauchen mindestens einen geeigneten Mechanismus zur Konkurrenzsynchonisierung!

3.4 Kritische Abschnitte

Kritischer Abschnitt (*critical section*, kA) :=

Codestück, das auf prozessglobale Daten oder Betriebsmittel zugreift.

- ++count
- count
- Zu einer Menge prozessglobaler Daten kann es mehrere kAe geben, die auf diese Daten zugreifen.
- Solche kAe dürfen nicht in beliebiger zeitlicher Überlappung, sondern nur unter gewissen Synchronisationsbedingungen ausgeführt werden.

Synchronisationsbedingung :=

Bedingung für die Ausführung einer Menge kAe als Ganzes.

- count
- count = #Pkw
- Eigentlich sind nicht die Codeabschnitte, sondern die prozessglobalen Daten kritisch: **kritische Daten**.
- Problem ist, die Konsistenz prozessglobaler Daten unter nebenläufigen Zugriffen zu erhalten.

3.4.1 Unteilbarkeit

Eine Menge von kAen heißt **unteilbar** (atomar, *indivisible*, *atomic*), wenn es zu jeder nebenläufigen Ausführung von kAen dieser Menge eine sequenzielle Ausführung dieser kAe gibt, die dieselbe Wirkung wie die nebenläufige Ausführung hervorbringt.

- Unteilbare kAe stören sich nicht gegenseitig, auch wenn sie von mehreren Prozessen nebenläufig ausgeführt werden.

Beispiele

- Einzelne Speicherzugriffe sind unteilbar.
- Kommandoausführungen in BlackBox sind unteilbar.
- Normale Maschinenbefehle sind ununterbrechbar, aber nicht notwendig unteilbar.
(Ein Befehl kann mehrere Speicherzugriffe hervorrufen.)
- Threads in *Java* sind nicht notwendig unteilbar.

3.4.2 Wichtige Synchronisationsbedingungen

- Einseitiger & gegenseitiger Ausschluss
⇒ **Ausschlusssynchronisation**
meist benötigt zur Konkurrenzsynchronisation
- Reihenfolgebeziehungen
⇒ **Reihenfolgesynchronisation**
oft benötigt zur Kooperationssynchronisation
- Warten auf das Eintreten von Ereignissen
⇒ **Ereignissynchronisation**
meist benötigt zur Kooperationssynchronisation

3.4.3 Zustände eines kritischen Abschnitts

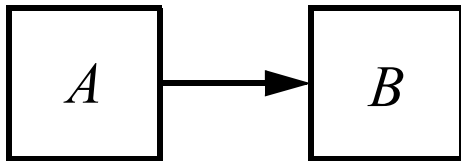
- nicht in Ausführung, nicht aktiviert
- aufgerufen, aber noch nicht betreten, noch nicht aktiviert
- in Ausführung, nach Aufruf betreten, aktiviert
- möglich: einfach oder mehrfach aktiviert

3.4.4 Ausschluss und Ausschlussgraph

Seien $A, B \in \text{Ae}$.

Bild 3.5

Ausschlussgraph für einseitigen Ausschluss

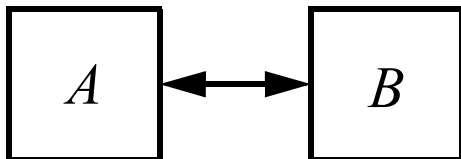


A **schließt** (*excludes*) B **aus** : $\Leftrightarrow$

wenn sich A in Ausführung befindet, darf B nicht betreten werden.

Bild 3.6

Ausschlussgraph für gegenseitigen Ausschluss



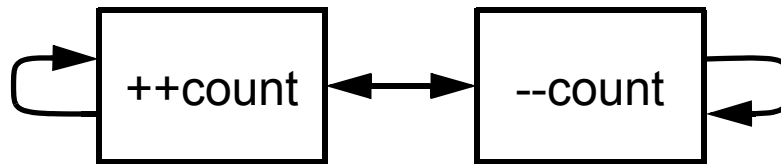
A & B **schließen sich gegenseitig aus** (*mutually exclude each other*) : $\Leftrightarrow$

A schließt B aus & B schließt A aus.

3.4.5 Beispiel: Parkhaus-Problem

Bild 3.7

Ausschluss
graph für
Parkhaus-
zähler-
aktualisie-
rung



3.5 Gegenseitiger Ausschluss

Eine Menge kAe steht unter der Synchronisationsbedingung **gegenseitiger Ausschluss** (*mutual exclusion*), wenn sich zu jedem Zeitpunkt höchstens einer der kAe in einfacher Ausführung befinden darf.

- KAe unter gegenseitigem Ausschluss dürfen/können/werden nebenläufig aufgerufen, aber nicht nebenläufig ausgeführt.
- KAe unter gegenseitigem Ausschluss sind unteilbar.

Fragen

- Wie erreicht man gegenseitigen Ausschluss von kAen?
 - Mit geeigneten Operationen,
 - die selbst wieder kAe sind!
- Wie erreicht man Unteilbarkeit von Operationen zur Implementation kAe?

3.5.1 Implementationen mit Synchronisationsprimitiven

Zu einer **Sperre** (*lock*) gehört ein Paar von Operationen, genannt **Synchronisationsprimitive**, die die Sperre

- setzen &
- aufheben.

Implementationen

klassisch
modern

prozedural
objektorientiert

Code 3.5
Abstrakte
Sperre

abstract class LOCK

feature

enter is atomic abstract

Eintrittsprotokoll, stellt Ausschluss her

exit is atomic abstract

Austrittsprotokoll, hebt Ausschluss auf

invariant

#called (exit) <= **#completed** (enter)

#completed (enter) <= **#completed** (exit) + 1

end -- LOCK

Erläuterungen

- atomic ● enter & exit unteilbar, dargestellt durch **atomic**
- #called ● **#called** (op) = Anzahl begonnener Aufrufe der Operation op
- #completed ● **#completed** (op) = Anzahl beendeter Aufrufe der Operation op
- $\forall op: \mathbf{\#completed} (op) \leq \mathbf{\#called} (op)$

Benutzung

- zu jeder Menge kAe , die sich gegenseitig ausschließen sollen, ein Objekt einer Sperren-Klasse vereinbaren
- jeden kA mit Aufrufen der Synchronisationsprimitive klammern

Ungeschützte Situation ohne Ausschluss

critical data D

process P:

critical section A accessing D



process Q:

critical section B accessing D



Geschützte Situation mit gegenseitigem Ausschluss

mutex : LOCK is init ...

critical data D

mutex.enter

critical section A accessing D

mutex.exit



mutex.enter

critical section B accessing D

mutex.exit



3.5.1.1 Ist Synchronisation mit einfachen Maschinenbefehlen möglich?

Übung 3.1 Warum versagt der folgende Lösungsversuch Code 3.6 für das Fehlschlag Ausschlussproblem?

Code 3.6 Undichte Sperre

```
class LEAKY_LOCK extend LOCK
feature {NONE}
  busy : BOOLEAN is init false

feature
  enter is
    do
      until not busy loop end
      busy := true
    end

  exit is
    do
      busy := false
    end

end -- LEAKY_LOCK
```

3.5.1.2 Dekkers Algorithmus

Idee: Statt ein Bit drei Bits

Code 3.7

Dekkers
Sperre

```
class DEKKERS_LOCK
```

```
feature {NONE}
```

```
  need  : ARRAY [BOOLEAN, BOOLEAN] is init (false, false)
```

```
  turn  : BOOLEAN
```

```
feature
```

```
  enter (self : BOOLEAN) is
```

```
    do
```

```
      from need [self] := true
```

```
        turn := not self
```

```
      until not need [not self] or turn = self loop end
```

```
    end
```

```
  exit (self : BOOLEAN) is
```

```
    do
```

```
      need [self] := false
```

```
    end
```

```
end -- DEKKERS_LOCK
```

Indextyp

Elementtyp

Benutzung von Dekkers Sperre

Code 3.8

Zwillinge

```
mutex : DEKKERS_LOCK
sibling (self : BOOLEAN) is process
  -- Two processes named false and true.
do
  loop
    ...
    mutex.enter (self)
    critical section (self)
    mutex.exit (self)
    ...
  end
end -- sibling
```

Bemerkungen zu Dekkers Lösung

- von Dekker gefunden, von Dijkstra 1965 erstmals veröffentlicht, von Peterson 1981 verbessert

Vorteil ☺ Synchronisation prinzipiell ohne spezielle Maschinenbefehle möglich

Nachteile ☹

- ☹ relativ kompliziert, damit fehleranfällig
- ☹ Anzahl (2) & Identität der Prozesse (false, true) festgelegt
- ☹ nicht leicht auf $n > 2$ Prozesse verallgemeinerbar (s. 3.5.1.3)
- ☹ realisiert durch **aktives Warten** (*busy waiting*). Ständiges Testen der Variablen kostet Prozessorzeit & Speicherzugriffe.

- praktisch irrelevant, aber sehr lehrreich

3.5.1.3 Lamports Bäckerei-Algorithmus

Ideen

- gegenseitiger Ausschluss von n Prozessen
- nur mit einfachen Lade- und Speichere-Befehlen
- ein Bit pro Prozess

Benutzung der Bäckerei-Sperre

Code 3.9

n Prozesse

```
mutex : BAKERY_LOCK
client (self : 1 .. n) is process
do
  loop
    ...
    mutex.enter (self)
    critical section (self)
    mutex.exit (self)
    ...
  end
end -- client
```

-- n processes named 1 to n .

Code 3.10 **class** BAKERY_LOCK

Bäckerei-
Sperr

```
(1)      ticket : ARRAY [1 .. n, NATURAL] is init each 0

feature
  enter (self : 1 .. n) is
    do
(2)      ticket [self] := 1
(3)      ticket [self] := max (ticket [1],..., ticket [n]) + 1      -- = ticket.max + 1
(4)      foreach other : 1 .. n loop
(5)        until ticket [other] = 0 or
(6)          ticket [self] < ticket [other] or
(7)          (ticket [self] = ticket [other] and self <= other)
(8)      loop end
      end
end
```

```
(9)      exit (self : 1 .. n) is
          do
            ticket [self] := 0
          end
        end -- BAKERY_LOCK
```

Übung 3.2 Erläutern Sie die Idee und die Funktionsweise von Code 3.10 und begründen Sie, warum die Bäckerei-Sperre gegenseitigen Ausschluss garantiert! Beziehen Sie sich auf die angegebenen Zeilennummern!

Korrektheit zeigen
(Prüfung SS 2006, 15 Punkte)

Bemerkungen zu Lamports Bäckerei-Sperre

Vorteil ☺ s. Vorteil von Dekkers Lösung

☺ es geht mit $n > 2$ Prozessen

Nachteile ☹ s. Nachteile von Dekkers Lösung außer Anzahl der Prozesse

3.5.1.4 Unterbrechungssperre

Idee

Unterbrechungen während der Ausführung eines kAs verbieten

Code 3.11 **class** INTERRUPT_LOCK **extend** LOCK

Sperre mit
maskierten
Unterbre-
chungen

feature

enter **is**

do

interrupts.disable

end

exit **is**

do

interrupts.enable

end

end -- INTERRUPT_LOCK

Bemerkungen zu dieser Lösung

- erfordert Aufruf einer BS-Operation (Trap-Befehl), um privilegierten Befehl auszuführen

Nachteile

- ☹ $\Rightarrow$ nur im BS nutzbar, nicht für Anwendungen
- ☹ nur bei Einprozessorsystemen geeignet
- ☹ Reaktionsfähigkeit auf externe Unterbrechungen herabgesetzt, besonders bei langen kAen
- ☹ $\Rightarrow$ nur für kurze kAe geeignet

- Ununterbrechbarkeit garantiert i.A. nicht Unteilbarkeit.
- Ein unteilbarer kA muss nicht notwendig ununterbrechbar sein.

3.5.1.5 Teste-und-Setze-Befehl

Idee

Spezieller unteilbarer Maschinenbefehl, dargestellt durch **atomic**

Code 3.12	TEST_&_SET (inout busy, loc : BOOLEAN) is atomic	
Teste-&-	do	
Setze-	loc := busy	(busy, loc) := (true, busy)
Befehl	busy := true	kollektive Zuweisung
	end	

Beim Aufruf ist

- busy prozessglobal (Speichervariable),
- loc prozesslokal (Condition Code, Register).

Code 3.13 **class** SPINLOCK **extend** LOCK

Kreisel-
Sperre

```
feature {NONE}
  busy : BOOLEAN is init false

feature
  enter is
    local
      loc : BOOLEAN
    do
      from TEST_&_SET (busy, loc)
      until not loc
    loop
      TEST_&_SET (busy, loc)
    end
  end

  exit is
    do
      busy := false
    end

end -- SPINLOCK
```

Bemerkungen zu dieser Lösung

Nachteile ☹ aktives Warten

☹ $\Rightarrow$ nur für kurze kAe geeignet

☹ Verklemmungsgefahr bei Einprozessorsystemen mit Prioritäten

☹ nicht fair, nicht aushungerungsfrei

3.5.1.6 Unterbrechungssperre und Teste-und-Setze-Befehl

Idee

Kombiniere 3.5.1.4 & 3.5.1.5

Übung 3.3 Warum scheitert der folgende Lösungsversuch Code 3.14 für das
Missgriff Ausschlussproblem?

Code 3.14 **class** DANGEROUS_LOCK **extend** LOCK

Gefährliche
Sperr

```
feature {NONE}
  busy : BOOLEAN is init false

feature
  enter is
    local
      loc : BOOLEAN
    do
      loop
        TEST_&_SET (busy, loc)
      until not loc
      interrupts.disable
    end

  exit is
    do
      interrupts.enable
      busy := false
    end

end -- DANGEROUS_LOCK
```

Code 3.15 **class** RIGHT_LOCK **extend** LOCK

Korrekte

Sperre

feature {NONE}

 busy : BOOLEAN **is** **init** false

feature

 enter **is**

local

 loc : BOOLEAN

do

 interrupts.disable

loop

 TEST_&_SET (busy, loc)

until not loc

end

 exit **is**

do

 busy := false

 interrupts.enable

end

end -- RIGHT_LOCK

Bemerkungen zu dieser Lösung

- erfordert Aufruf einer BS-Operation

Nachteile ☹ $\Rightarrow$ nur im BS nutzbar, nicht für Anwendungen

☹ eingeschränkte Reaktionsfähigkeit & aktives Warten

☹ $\Rightarrow$ nur für kurze kAe geeignet

☹ nicht fair

3.5.2 Fazit zu Synchronisationsprimitiven

Vorteil ☺ für kurze kAe innerhalb von BS-Operationen nutzbar

Nachteile ☹ Nachteile von Code 3.15

- KAe müssen mit enter & exit geklammert sein.
- ☹ Vergisst der Programmierer ein exit, so bleibt der kA gesperrt.
- ☹ Wird ein Prozess in einem kA abgebrochen (per Kommando oder Ausnahme), so bleibt der kA gesperrt.
- ⇒ höhere Synchronisationsmechanismen erwünscht

3.6 Anforderungen an höhere Synchronisationsmechanismen

- wichtig
- (1) für Anwendungen nutzbar
 - (2) für Konkurrenz-/Ausschlusssynchronisation geeignet
 - (3) für Kooperations-/Ereignissynchronisation geeignet
 - (4) Reaktionsfähigkeit des Rechners auf externe Unterbrechungen erhaltend
 - (5) aktives Warten vermeidend
 - (6) für lange kAe geeignet
 - (7) Klammerungsprobleme vermeidend
 - (8) trotz Fehlern in kAen funktionsfähig
- unwichtig
- (9) fair

3.7 Semaphore

Edsger W. Dijkstra: *Cooperating Sequential Processes*
Technological University Eindhoven (1965)

Ein **Semaphor** (*semaphore*) ist ein Synchronisationsmechanismus mit zwei Operationen,

- P (von holländisch „passerem“) &
- V (von holländisch „vrygeven“).

Intern gehören zu einem Semaphor

- ein Zähler &
- ein Wartezustand für Prozesse.

Tabelle 3.3
Namen für
Semaphor-
operationen

Dijkstra	<i>Algol 68</i>	<i>PEARL</i>	hier
P	down	REQUEST	await
V	up	RELEASE	signalize

Semaphor als Klasse

Code 3.16 Semaphore

```
class SEMAPHORE (init_count : NATURAL)
feature {NONE}
    count : INTEGER is init init_count

feature
    await is atomic
        do
            count := count - 1
            if count < 0 then keep waiting until signaled end
        end

    signalize is atomic
        do
            count := count + 1
            if count <= 0 then let some waiting process continue end
        end

invariant
    #completed (await) <= #completed (signalize) + init_count

end -- SEMAPHORE
```

Semantik

- `await` & `signalize` unteilbar, dargestellt durch **atomic**
- Bedeutung des Zählers `count`:
 - `count > 0` : Anzahl verfügbarer Betriebsmittel
 - `count < 0` : Anzahl wartender Prozesse
 - `count = 0` : kein Betriebsmittel verfügbar, kein Prozess wartet
- Initialwert für `count` ist ≥ 0 ,
da anfangs keine Prozesse warten
- Invariante:
 - angefangener `await`-Aufruf ggf. verzögert,
bis entsprechend viele `signalize`-Aufrufe beendet
 - wichtig für Korrektheitsbeweise von Problemlösungen
- `await` & `signalize` sind **kommutativ**, Signal wird gemerkt:
`sem.await; sem.signalize` äquivalent zu `sem.signalize; sem.await`

3.7.1 Gegenseitiger Ausschluss

- Betriebsmittel sind die unter gegenseitigem Ausschluss stehenden kAe.
- Zu jedem Zeitpunkt ist höchstens ein kA verfügbar.

Code 3.17 **class** SEMAPHORE_LOCK **extend** LOCK

Semaphor-
Sperre

feature {NONE}

mutex : SEMAPHORE **is** **init** (1)

feature

enter **is**

do

mutex.await

end

exit **is**

do

mutex.signalize

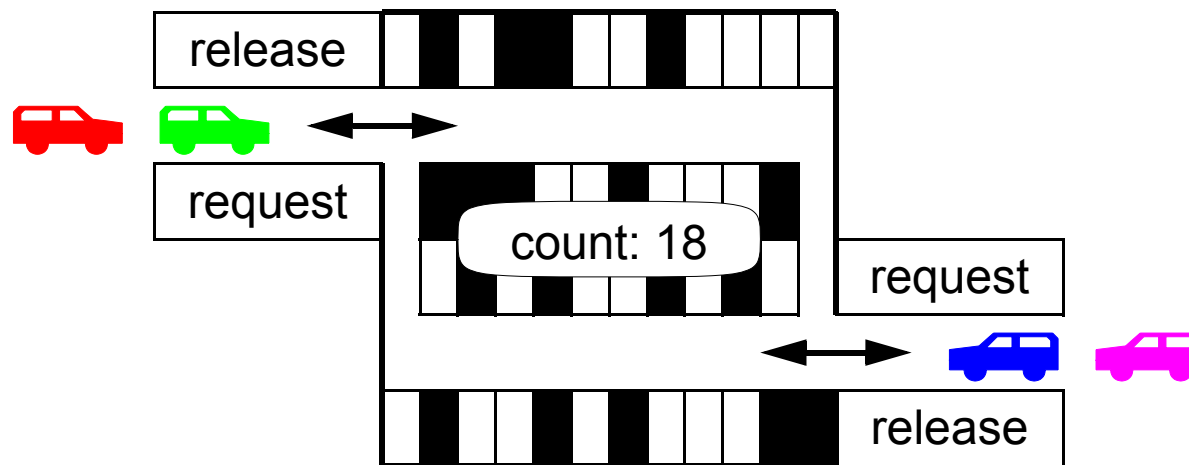
end

end -- SEMAPHORE_LOCK

3.7.2 Beispiel: Parkhaus-Problem

Lösungskonzept 2

Bild 3.8
Parkhaus
mit Zähler
& Tickets



- Zähler für parkende Pkw im Parkhaus
 - Aufpasser entlassen, dafür übernehmen Pkw
 - Anfordern &
 - Freigeben
- von Parkplätzen

3.7.2.1 Abstrakter Lösungsansatz

Code 3.18 **abstract class** CAR_PARK (init_capacity : NATURAL)
Abstraktes **feature** {car}
Parkhaus capacity : NATURAL **is init** init_capacity
 request **is atomic abstract**
 release **is atomic abstract**

 invariant
 capacity > 0
 #called (release) <= **#completed** (request)
 #completed (request) <= **#completed** (release) + capacity

 end -- CAR_PARK

Code 3.19 **car_park : CAR_PARK is init (100)**
Pkw **car (self : 1 .. 100 * car_park.capacity) is process**
 do
 until crashed
 loop
 drive_around (random_time, random_places)
 car_park.request
 stand_still (random_time)
 car_park.release
 end
 end -- car

3.7.2.2 Konkurrenzsynchronisation

Code 3.20 **class** UNBOUNDED_CAR_PARK **extend** CAR_PARK

Ideales

Parkhaus

feature {NONE}

mutex : SEMAPHORE **is init** (1)

feature {car}

count : NATURAL **is init** 0

Anzahl parkender Pkw

request **is**

do

mutex.await

count := count + 1

mutex.signalize

end

release **is**

do

mutex.await

count := count - 1

mutex.signalize

end

end -- UNBOUNDED_CAR_PARK

Bemerkungen zu diesem Entwurf

- Ausschlusssynchronisation beim Aktualisieren des Zählers count durch Semaphore mutex
- count wächst beliebig
⇒ Parkhaus muss unbeschränkte Kapazität haben
- ☹ Überlastschutz fehlt, capacity ungenutzt

3.7.2.3 Kooperationssynchronisation

Code 3.21 Einfaches Parkhaus

```
class SIMPLE_CAR_PARK extend CAR_PARK
feature {NONE}
    free_places : SEMAPHORE is init (capacity)
feature {car}
    request is
        do
            free_places.await
        end
    release is
        do
            free_places.signalize
        end
end -- SIMPLE_CAR_PARK
```

Bemerkungen zu dieser Lösung

- Ereignissynchronisation zur Kooperation Pkw - Parkhaus durch Semaphor `free_places`
- Betriebsmittel sind die freien Parkplätze.
- $\Rightarrow$ `capacity` genutzt zum Initialisieren von `free_places`
- Zähler `count` nicht notwendig, da `free_places` die freien Plätze zählt

- Bei diesem Problem genügt Kooperationssynchronisation mit Ereignissen
 - „Parkhaus belegt“,
 - „Stellplatz freigegeben“.
- Bei anderen Problemen treten Kooperations- & Konkurrenzsynchrisation kombiniert auf.
- $\Rightarrow$ 3.7.2.4 kombiniert 3.7.2.2 & 3.7.2.3

3.7.2.4 Kooperations- und Konkurrenzsynchonisierung

Code 3.22 **class** LUXURY_CAR_PARK **extend** CAR_PARK

Luxus-
Parkhaus

feature {NONE}

free_places : SEMAPHORE **is init** (capacity)

mutex : SEMAPHORE **is init** (1)

feature {car}

count : NATURAL **is init** 0

request is

do

free_places.await

mutex.await

count := count + 1

mutex.signalize

end

invariant

count <= capacity

end -- LUXURY_CAR_PARK

release is

do

mutex.await

count := count - 1

mutex.signalize

free_places.signalize

end

Bemerkungen zu dieser Lösung

- Ereignissynchronisation zur Kooperation Pkw - Parkhaus durch Semaphor `free_places`
- `capacity` genutzt zum Initialisieren von `free_places`
- Ausschlusssynchronisation beim Aktualisieren des Zählers `count` durch Semaphor `mutex`
- `count` intern redundant, aber für Pkw sichtbar

☺ beweisbar korrekte Lösung

☹ Anfordernder Pkw muss ggf. zweimal warten:

1. auf freien Platz,
2. auf Erhöhen des Zählers.

Übung 3.4 Warum misslingt der folgende Lösungsversuch Code 3.23 für das
Irrtum Parkhaus-Problem?

Code 3.23 Tödliches Parkhaus

```
class DEADLY_CAR_PARK extend CAR_PARK
    Vereinbarungen & Invarianten wie bei Code 3.22

    request is
        do
            mutex.await
            free_places.await
            count := count + 1
            mutex.signalize
        end

    release is
        do
            mutex.await
            count := count - 1
            free_places.signalize
            mutex.signalize
        end

end -- DEADLY_CAR_PARK
```

3.7.3 Fazit zu Semaphoren

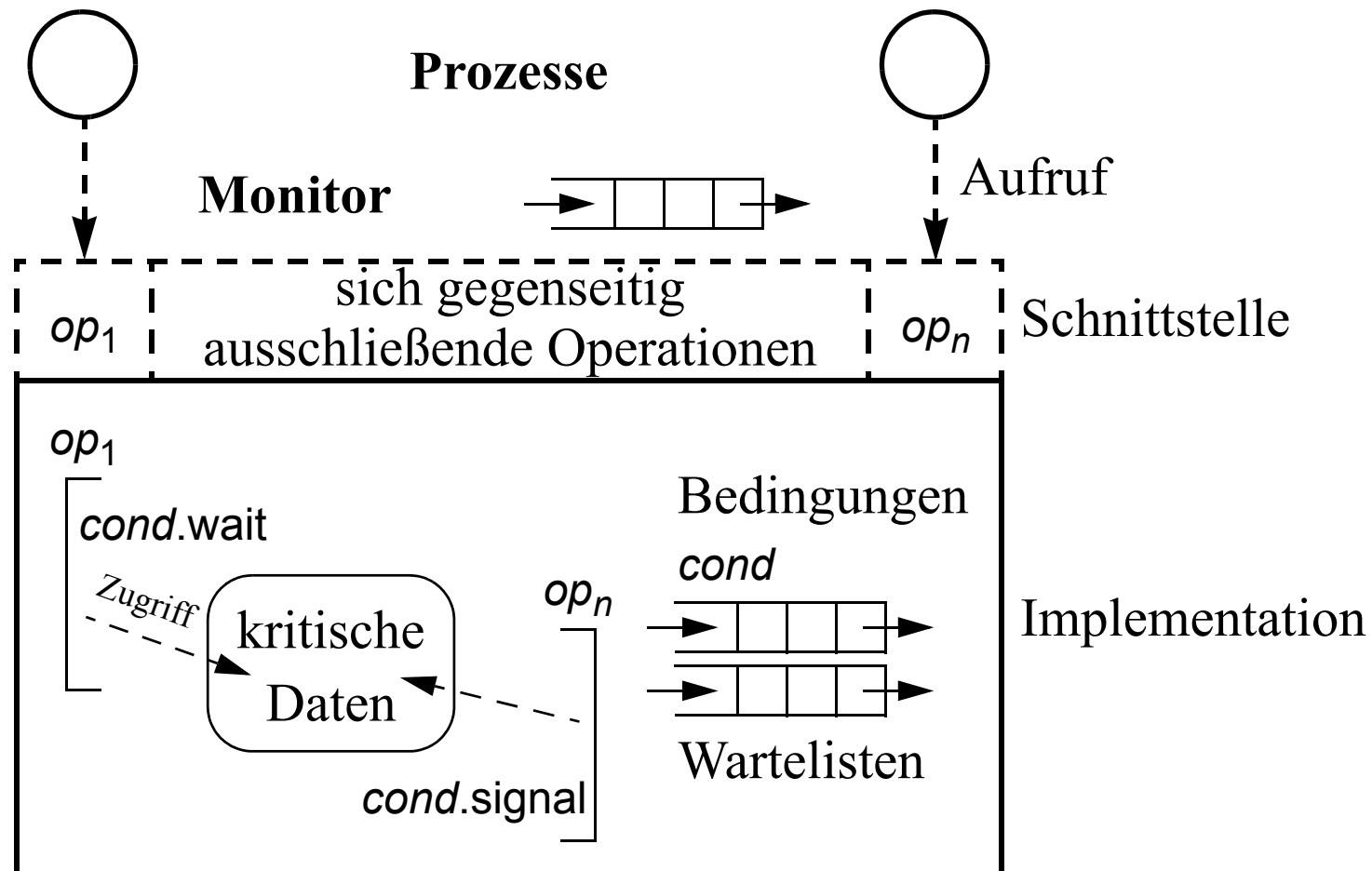
- Vorteile
- ☺ erfüllen die Anforderungen 3.6 (1) bis (6)
 - ☺ flexibles Synchronisationskonzept, das viele Synchronisationsprobleme lösen kann
 - ☺ Korrektheit von Lösungen beweisbar
- Nachteile
- ☹ Anforderungen 3.6 (7) bis (9) nicht erfüllt
 - ☹ await & signalize paarweise passend auszuführen
⇒ vergessener Aufruf führt zu Fehlsynchronisation
 - ☹ await- & signalize-Aufrufe über Quelltext verstreut
⇒ Synchronisation schwer zu überblicken & zu verstehen
 - ☹ an await- & signalize-Aufrufen allein nicht erkennbar, ob sie Kooperations- oder Konkurrenzsynchronisation dienen

3.8 Monitore

C.A.R. Hoare: *Monitors: An Operating System Structuring Concept*
CACM 17, 10 (1974)

Bild 3.9

Monitor als
synchroni-
sierter ADT



Ideen

- höheres Synchronisationskonzept als Semaphore
- kritische Daten in ADT verbergen
- Konkurrenzsynchronisation, indem sich alle Schnittstellenoperationen gegenseitig ausschließen
- Kooperationssynchronisation durch Bedingungsmechanismus (*condition*)

3.8.1 Parkhaus-Problem mit Konkurrenzsynchronisation

Code 3.24 **monitor class** UNBOUNDED_CAR_PARK **extend** CAR_PARK

Ideales
Parkhaus

```
feature {car}  
  count : NATURAL is init 0
```

```
  request is implizit unteilbar
```

```
    do
```

```
      count := count + 1
```

```
    end
```

```
  release is implizit unteilbar
```

```
    do
```

```
      count := count - 1
```

```
    end
```

```
end -- UNBOUNDED_CAR_PARK
```

☺ request & release schließen sich implizit wegen des Schlüsselworts **monitor** aus.

☹ Überlastschutz fehlt, capacity ungenutzt

3.8.2 Kooperations-/Ereignissynchronisation

- Synchronisationsmechanismus „**Bedingung**“ (*condition*) als ADT

Code 3.25

Bedingung

```
class CONDITION
feature {monitor}
  wait is atomic
  do
    keep waiting until signalled
  end

  signal is atomic
  do
    let some waiting process continue, if any
  end
end -- CONDITION
```

- nur innerhalb eines Monitors vereinbar & benutzbar
- wait & signal sind **nicht** kommutativ, Signal kann verloren gehen:
cond.wait; cond.signal **nicht** äquivalent zu cond.signal; cond.wait

3.8.2.1 Konflikt zwischen signalisierendem und signalisiertem Prozess

- Höchstens ein Prozess darf die Kontrolle über den Monitor besitzen.
- Welcher Prozess erhält bei signal die Monitorkontrolle?
Alternativen:
 - (a) der signalisierte Prozess
 - (b) der signalisierende Prozess
- Wann erhält der andere Prozess die Monitorkontrolle zurück?
- In welchem Zustand ist dann der Monitor?

3.8.2.2 Hoares Signal-Semantik

- Der signalisierte Prozess erhält die Monitorkontrolle vom signalisierenden Prozess.
- Der signalisierende Prozess kann dafür sorgen, dass die Bedingung, auf die der signalisierte Prozess gewartet hat, erfüllt ist.
- $\Rightarrow$ Der signalisierte Prozess muss die Bedingung nicht erneut prüfen.
- Der signalisierende Prozess
 - blockiert sich &
 - erhält die Monitorkontrolle bevorzugt vor Erstanforderern zurück (*urgent queue*).
- Bei vielen Problemen steht signal am dynamischen Ende von Monitoroperationen, sodass der signalisierende Prozess die Monitorkontrolle gar nicht mehr benötigt.

3.8.3 Parkhaus-Problem mit Kooperationssynchronisation

3.8.3.1 Signalisierter Prozess erhält Monitorkontrolle

Code 3.26 monitor class LUXURY_CAR_PARK extend CAR_PARK

Luxus-
Parkhaus

feature {NONE}

free_places : CONDITION

feature {car}

count : NATURAL is init 0

request is

do

if count >= capacity then

free_places.wait

end

assert count < capacity **end**

count := count + 1

end

invariant

count <= capacity

end -- LUXURY_CAR_PARK

release is

do

count := count - 1

assert count < capacity **end**

free_places.signal

end

3.8.3.2 Signalisierender Prozess behält Monitorkontrolle und läuft weiter

Code 3.27

Schlichtes
Parkhaus

```
monitor class MODEST_CAR_PARK extend CAR_PARK
```

```
feature {NONE}
```

```
    free_places : CONDITION
```

```
feature {car}
```

```
    count : NATURAL is init 0
```

```
request is
```

```
    do
```

```
        until count < capacity
```

```
        loop
```

```
            free_places.wait
```

```
        end
```

```
        assert count < capacity end
```

```
        count := count + 1
```

```
    end
```

```
invariant
```

```
    count <= capacity
```

```
end -- MODEST_CAR_PARK
```

```
release is
```

```
    do
```

```
        count := count - 1
```

```
        assert count < capacity end
```

```
        free_places.signal
```

```
    end
```

Bemerkungen zu beiden Lösungen

- ☹ Der signalisierende Prozess weiß nicht, ob ein Prozess wartet.
- ☹ $\Rightarrow$ unnötige, weil effektlose Systemaufrufe

Bemerkungen zu Code 3.27

- ☹ Der signalisierte Prozess weiß nicht, ob seine Fortsetzungsbedingung erfüllt ist, da andere Prozesse sich vorgedrängt haben können.
- ☹ $\Rightarrow$ Er muss die Bedingung in einer Schleife prüfen.
- ☹ $\Rightarrow$ partiell aktives Warten

Im Rest von 3.8 gilt Hoares Signal-Semantik!

3.8.4 Gegenseitiger Ausschluss

Code 3.28 **monitor class** MONITOR_LOCK **extend** LOCK

Monitor-
Sperre

feature {NONE}

 busy : BOOLEAN **is init** false

 nonbusy : CONDITION

feature

 enter **is**

do

if busy **then** nonbusy.wait **end**

assert not busy end

 busy := true

assert busy end

end

 exit **is**

do

assert busy end

 busy := false

assert not busy end

 nonbusy.signal

end

end -- MONITOR_LOCK

- Sperren durch Monitore implementierbar (ohne aktives Warten)
- ☺ ⇒ alle mit Sperren lösbaren Synchronisationsprobleme auch mit Monitoren lösbar

3.8.5 Semaphore

Code 3.29 **monitor class** MONITOR_SEMAPHORE (init_count : NATURAL)

Monitor-

extend SEMAPHORE

Semaphor

feature {NONE}

count : INTEGER **is** **init** init_count

geerbt von SEMAPHORE

cond : CONDITION

feature

await **is**

do

count := count - 1

if count < 0 **then** cond.wait **end**

end

signalize **is**

do

count := count + 1

if count <= 0 **then** cond.signal **end**

end

end -- MONITOR_SEMAPHORE

Bemerkungen zu dieser Lösung

- Semaphore durch Monitore implementierbar
- ☺ $\Rightarrow$ alle mit Semaphoren lösbaren Synchronisationsprobleme auch mit Monitoren lösbar

Übung 3.5
Implementa-
tion von
Monitoren
mit Sema-
phoren

Monitore lassen sich mit Semaphoren implementieren.

Geben Sie eine solche Implementation an!

- ☺ $\Rightarrow$ alle mit Monitoren lösbaren Synchronisationsprobleme auch mit Semaphoren lösbar

Fazit

- ☺ $\Rightarrow$ Semaphore & Monitore haben gleiche Problemlösungsmächtigkeit, da sie dieselben Synchronisationsprobleme lösen können.
- Semaphor- & Monitorlösungen von Synchronisationsproblemen können unterschiedlich effizient sein.

3.8.6 Fazit zu Monitoren

- Vorteile
- ☺ erfüllen die Anforderungen 3.6 (1) bis (6) & teilweise (7)
 - ☺ flexibles Synchronisationskonzept, das viele Synchronisationsprobleme lösen kann
 - ☺ bei Hoares Signal-Semantik Korrektheit von Lösungen beweisbar
 - ☺ Ausschlusssynchronisation implizit, aus Anweisungsteil abstrahiert
- Nachteile
- ☹ Ereignissynchronisation explizit im Anweisungsteil
 - ☹ wait & signal passend auszuführen
⇒ vergessener Aufruf kann zu Fehlsynchronisation führen
 - ☹ wait- & signal-Aufrufe über Quelltext verstreut
⇒ Ereignissynchronisation schwer zu überblicken & zu verstehen
 - ☹ potenzielle Probleme bei geschachtelten Monitoraufrufen

3.9 Monitore in Java

3.9.1 Semantik verglichen mit Hoares Monitor

3.9.1.1 Ausschlusssynchronisation

- `synchronized`-Attribut bei Methoden
statt **monitor**-Attribut bei Klasse (ähnlich **atomic**)
 - `synchronized`-Methoden schließen sich gegenseitig aus.
☺ $\Rightarrow$ Nicht alle öffentlichen Methoden müssen sich gegenseitig ausschließen.
- `synchronized` auf Blöcke innerhalb von Methoden anwendbar
☺ $\Rightarrow$ Nicht ganze Methodenrumpfe müssen sich ausschließen.

Durch `synchronized`-Attribut

☺ mehr Flexibilität

☹ weniger Sicherheit

3.9.1.2 Ereignissynchronisation

- Methoden `wait`, `notify`, `notifyAll` der Klasse `Object` statt `CONDITION`-Typ
 - `wait` entspricht `wait`, `notify` entspricht `signal`.
 - `notifyAll` lässt alle wartenden Threads weiterlaufen.
 - `wait`, `notify`, `notifyAll` dürfen nur in `synchronized`-Blöcken desselben Objekts aufgerufen werden, sonst wird eine `IllegalMonitorStateException`-Ausnahme ausgelöst.
- ☹ ⇒ pro Monitor nur eine Warteliste für alle Threads zum Warten auf Ereignisse, keine Differenzierung zwischen Ereignissen
- ☹ ⇒ viele Synchronisationsprobleme nur umständlich & aufwändig zu lösen

3.9.1.3 Semantik von notify und notifyAll

- Der notifizierende Thread behält die Monitorkontrolle & läuft weiter.
- Jeder notifizierte Thread wird deblockiert & muss sich erneut um die Monitorkontrolle bewerben, um nach Erhalt der Monitorkontrolle nach dem `wait`-Aufruf weiter zu laufen.
- ☹ pro Monitor nur eine Warteliste für alle Threads zum Warten auf die Monitorkontrolle, keine Differenzierung zwischen Erstanforderung & Wiedereintritt nach `wait`
- ☹ $\Rightarrow$ Erstanforderer können notifizierten Threads zuvorkommen.
- ☹ $\Rightarrow$ Notifizierte Threads müssen ihre Wartebedingung erneut prüfen.
- ☹ $\Rightarrow$ `while`-Schleife statt `if`-Alternative

3.9.1.4 Semantik von wait

- Ein wartender Thread t kann unterbrochen werden, indem ein anderer Thread $t.interrupt()$ aufruft.
- Dadurch wird das Warten von t beendet & in t eine InterruptedException-Ausnahme ausgelöst, die der t -Thread
 - weiterreichen kann oder
 - vernünftig behandeln muss.

3.9.2 Gegenseitiger Ausschluss

Code 3.30

Java: Lock

```
public class Lock {  
    protected boolean busy = false;  
    public synchronized void enter ()  
                                throws InterruptedException  
    {  
        while (busy) {  
            wait();  
        }  
        busy = true;  
    }  
    public synchronized void exit ()  
    {  
        busy = false;  
        notify();  
    }  
}
```

Bemerkungen zu dieser Lösung

- ☺ `enter` & `exit` schließen sich explizit wegen des Schlüsselworts `synchronized` aus.
- `wait` & `notify` implizit von `Object` geerbt, daher unqualifiziert aufgerufen
- In `wait` kann prinzipiell eine `InterruptedException` ausgelöst werden, die in Code 3.30 weitergereicht wird.

Alternative mit Ausnahmebehandlung

Code 3.31

Java: Lock
with
Exception
Handling

```
public class LockwithXH {  
    public synchronized void enter ()  
                                throws InterruptedException {  
        while (busy) {  
            try {  
                wait();  
            } catch(InterruptedException e) {  
                throw new InterruptedException("enter wait failed");  
            }  
        }  
        busy = true;  
    }  
    ...  
}
```

- `wait()` in `try {...} catch(...) {...}`-Struktur einbetten
- als Ausnahmebehandlung neue Ausnahme auslösen oder Thread terminieren lassen

Schlechte Alternative mit leerer Ausnahmebehandlung

Code 3.32

Java: Bad
Lock

```
public class InterruptResistantLock {  
    public synchronized void enter () {  
        while (busy) {  
            try {  
                wait();  
            } catch (InterruptedException e) {}  
        }  
        busy = true;  
    }  
    ...  
}
```



- Beim Warten unterbrochener Thread prüft Wartebedingung & wartet ggf. erneut.
- ☹ Thread abbrechen, damit er an der Unterbrechungsstelle weiterläuft?
- ☹ Eine **leere Ausnahmebehandlung** im catch-Block ist zwar **praktisch unsinnig**, wird aber in manchen Lehrbüchern so propagiert!

3.9.3 Semaphore

Code 3.33 Java: Semaphore

```
public class Semaphore {
    protected int count;

    public Semaphore (int initCount) {
        assert initCount >= 0 : "Precondition: Initial value..";
        count = initCount;
    }

    public synchronized void await ()
        throws InterruptedException {
        --count;
         if (count < 0) wait();
    }

    public synchronized void signalize () {
        ++count;
        if (count <= 0) notify();
    }
}
```

3.9.4 Parkhaus-Problem

Code 3.34

```
public class CarPark {  
    protected int capacity;  
    protected int count = 0;  
  
    public CarPark (int initCapacity) {  
        assert initCapacity > 0 : "Precondition: Initial val..";  
        capacity = initCapacity;  
    }  
  
    public synchronized void request ()  
        throws InterruptedException {  
        while (count >= capacity) wait();  
        ++count;  
    }  
  
    public synchronized void release () {  
        assert count > 0 : "Precondition: #called(release) ...";  
        --count;  
        notify();  
    }  
}
```

3.10 Aufgaben

Laden Sie obige Klassen von

<ftp://studinf.reutlingen-university.de/MKI/Hug/Lehrveranstaltungsmaterial/I3/Java/Threads/Source/>

herunter & nutzen Sie sie in folgenden Beispielen!

Übung 3.6 Pkw- Parkhaus- Simulation

Implementieren Sie eine Pkw-Parkhaus-Simulation in *Java* mit Pkw gemäß Code 3.19 & einem Parkhaus gemäß Code 3.34!

Übung 3.7 Parkhaus mit Sema- phoren

Implementieren Sie eine Variante der Klasse `carPark`, die nicht das Attribut `synchronized` benutzt, sondern die Klasse `Semaphore` (Code 3.33) gemäß Code 3.22! Testen Sie die Pkw-Parkhaus-Simulation von Übung 3.6 damit!

A

Literaturverzeichnis

- | | |
|--------------------|---|
| Allgemeines | <p>[And91] Gregory R. Andrews: <i>Concurrent Programming. Principles and Practice</i>. The Benjamin/Cummings Publishing Company Inc., Redwood City, California (1991) 637 S.</p> <p>[Fre87] Bernhard Freisleben: <i>Mechanismen zur Synchronisation paralleler Prozesse</i>. Springer-Verlag, Berlin u.a. (1987) 357 S.</p> <p>[HeH89] Ralf Guido Herrtwich, Günter Hommel: <i>Kooperation und Konkurrenz. Nebenläufige, verteilte und echtzeitabhängige Programmsysteme</i>. Springer-Verlag, Berlin u.a. (1989) 463 S.</p> <p>[ZöH88] Dieter Zöbel, Horst Hogenkamp: <i>Konzepte der parallelen Programmierung</i>. B. G. Teubner, Stuttgart (1988) 235 S.</p> |
| Java | <p>[Lea97] Doug Lea: <i>Concurrent Programming in JavaTM. Design Principles and Patterns</i>. Addison-Wesley Longman Inc., Reading, Massachusetts (1997) 339 S.</p> <p>[Oec01] Rainer Oechsle: <i>Parallele Programmierung mit Java Threads</i>. Fachbuchverlag Leipzig im Carl Hanser Verlag, München/Wien (2001) 173 S.</p> |

