Nebenläufige Programmierung in Java

Karl Friedrich Gebhardt

Auflage vom 10. Oktober 2019

Prof. Dr. K. F. Gebhardt

Tel: 0711-667345-11(16)(15)(12)
Fax: 0711-667345-10
email: kfg@lehre.dhbw-stuttgart.de

Inhaltsverzeichnis

Kapitel 1

Nebenläufigkeit

Dieses Kapitel behandelt eine Abstraktion der Probleme bei Betriebs- und Realzeit-Systemen. Reale (technische) Systeme bestehen aus nebenläufig oder parallel ablaufenden Prozessen. Um für solche Systeme einen Ausschnitt der realen Welt bequem modellieren zu können, muss das Software-Entwicklungs-System nebenläufige Prozesse zur Verfügung stellen.

Bei Realzeit-Systemen spricht man von **Nebenläufigkeit, *Concurrency*** (Gehani): *A concurrent program specifies two or more sequential programs that may be executed concurrently as parallel process.*

Oft wird in diesem Zusammenhang der Begriff "Parallelität" bzw. "parallel" synonym für "Nebenläufigleit" bzw. "nebenläufig" verwendet.

Wir machen hier folgende Unterscheidung: Nebenläufige Prozesse sind dann parallele Prozesse, wenn sie wirklich gleichzeitig, also etwa auf zwei oder mehreren Prozessoren ablaufen. "Nebenläufigkeit" ist der allgemeinere Begriff und umfasst auch die "Quasi-Parallelität", die uns ein Multitasking-Betriebssystem auf einem oder wenigen Prozessoren vorspiegelt.

Das Gegenteil von "nebenläufig" ist "sequentiell" oder "seriell".

Abhängig vom Betriebssystem oder der Programmiersprache spricht man von "Prozessen", "Threads" oder "Tasks". Wir werden hier als übergeordneten Begriff **"Task"** verwenden.

Betrachten wir als Beispiel ein Realzeit-System, das einen Druck überwacht, verschiedene andere Signale (1 bis 4) verarbeitet und schließlich eine Berechnung durchführt. Das könnte man folgendermaßen programmieren, wobei der Code zyklisch durchlaufen wird:

```
Druck
Signal1
Signal2
Signal3
Signal4
Berechnung
```

Wenn der Druck sehr wichtig ist, dann könnte das folgendermaßen programmiert werden:

```
Druck
Signal1
Druck
Signal2
Druck
Signal3
Druck
Signal4
Druck
Berechnung
```

Wenn nun `Signal3` eine mittlere Wichtigkeit hätte, wie würde dies dann aussehen? Es würde sehr unübersichtlich werden. Solch ein System ist also sehr schwer wart- und erweiterbar, etwa um weitere Signale. Daher wünscht man sich hier ein Betriebssystem, das die nebenläufige Kontrolle von Automatisierungsaufgaben und die Vergabe von Prioritäten erlaubt. Jeder Signal- oder Ereignisquelle wird eine **Task** zugeordnet. Jede Task bekommt eine bestimmte Priorität. Dann hätten wir ein aus mehreren nebenläufigen Programmen (Tasks) bestehendes Programm. Alle Programme werden nebenläufig mit unterschiedlicher Priorität als Task zyklisch durchgeführt:

```
Task D Prio 1     Task 1 Prio 2     Task 2 Prio 2     ...   Task B Prio 3

   Druck             Signal1           Signal2        ...      Berechnung
```

Solch ein System ist relativ leicht zu warten und zu erweitern.

Die Parallelität ist bei einem Einprozessorsystem nur eine **Quasiparallelität** oder **Pseudoparallelität**, d.h. der Prozessor bearbeitet zu einem Zeitpunkt immer nur genau eine Task. Der **Scheduler** ist die Komponente eines **Multitasking-**Betriebssystems, die den Tasks die Kontrolle über die CPU zuteilt. Er lässt das System nach außen hin parallel erscheinen.

Bemerkungen:

1. Aus Gründen der Performanz und/oder Einfachheit des Systems werden Systeme *ohne* Nebenläufigkeit bei Embedded Systemen und in der Mikroprozessorprogrammierung durchaus noch verwendet.

2. Normalerweise ist ein gut entworfenes nebenläufges Programm trotz des Taskwechsel-Overheads performanter als ein rein sequentielles Programm.

3. Die Verwendung von Mehrprozessor-Systemen ***(multi core)*** wird zur Zeit sehr diskutiert. Teilweise werden sie aus Gründen der Performanz schon eingesetzt, teilweise sind sie aus Gründen der Sicherheit (noch?) regelrecht verboten.

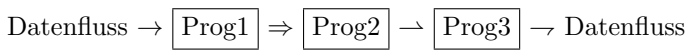
1.1 Arten, Vorteile, Notwendigkeit von Nebenläufigkeit

1.1.1 Parallele Algorithmen

Bei einem parallelen Algorithmus kann eine Berechnungsaufgabe so in Teilaufgaben zerlegt werden, dass die Teilaufgaben parallel abgearbeitet werden können. Beispiele sind Quicksort oder Fast-Fourier-Transformation. Eigentlich sind das sogenannte *binäre* Algorithmen. Sie werden normalerweise sequentiell durchgeführt, bringen aber einen deutlichen Zeitgewinn aufgrund des parallelen bzw. binären Algorithmus.

1.1.2 Konzeptionelle Vereinfachung

Betriebssystem-, Realzeit-System- und Datenbankprogramme werden durch Nebenläufigkeit konzeptionell vereinfacht, indem die Datenverarbeitung als ein (unendlicher) Datenfluss aufgefasst wird, der durch verschiedene *nebenläufig laufende* Unterprogramme bearbeitet und verändert wird:



Durch eine solche Aufteilung kann ein sehr verschachtelter Code in eine sequentielle Form gebracht werden, die leichter zu schreiben, zu lesen und zu warten ist.

Berühmt ist die Aufgabe von Conway: Lies einen Textfile mit Zeilen der Länge 40. Ersetze jede Folge von n = 1...10 Nullen durch eine Null gefolgt von n mod 10. Gib aus als Textfile mit Zeilen der Länge 75.

1. Erstellung **eines** ("monolithischen") Programms `leprschr`:

   ```
   $ leprschr < inFile > outFile
   ```

2. Erstellung von **drei** Programmen `lese`, `presse` und `schreibe`, die dann unter Verwendung des Pipe-Mechanismus nebenläufig ausgeführt werden:

   ```
   $ lese < inFile | presse | schreibe > outFile
   ```

Es zeigt sich, dass Aufgabe 2.) wegen der Aufteilung in drei einfache Aufgaben wesentlich einfacher als Aufgabe 1.) ist.

1.1.3 Multicomputer-Architekturen

Durch Parallelität können Multicomputer-Architekturen effizient ausgenutzt werden. Wir unterscheiden da:

- Transputer (Parallelität auf einem Chip)
- Multiprozessor-Architektur (Parallelität auf einem Board)
- Vernetzte Computer ***(Grid-Computing, Cloud-Computing)***

1.1.4 Ausnützen von Wartezeiten

Bei Uniprozessoren werden durch Nebenläufigkeit Wartezeiten effizient ausgenützt, um andere Benutzer zu bedienen. (Das dürfte die ursprüngliche Motivation zur Entwicklung von Multitasking-Betriebssystemen gewesen sein.) Ferner kann der Prozessor allen Benutzern gerecht zugeteilt werden.

1.1.5 Nebenläufigkeit als Modell

Die Systementwicklungsaufgabe erfordert die Nebenläufigkeit als Modell. Die zu modellierende reale Welt besteht aus parallel ablaufenden Prozessen. Die Überwachung von mehreren Alarmen oder die Steuerung eines Roboters mit mehreren Achsen würde sequentiell zu stark verzahnten oder verschachtelten Programmen führen.

Diese Art von Nebenläufigkeit ist das Thema der Vorlesung.

1.2 Konkurrenz und Kooperation

Die wichtigste Aufgabe bei der nebenläufigen Programmierung ist die **Synchronisation** und der **Datenaustausch** zwischen den Tasks. Die Tasks konkurrieren ***(Competition)*** um Betriebsmittel, müssen aber in einem technisch interessanten Programm auch zusammenarbeiten ***(Cooperation)***.

Für die konzeptionelle und programmiertechnische Vereinfachung durch den Einsatz von nebenläufigen Programmen muss ein Preis entrichtet werden: Die Anzahl der möglichen Verschachtelungen von nebenläufigen Tasks ist für praktische Zwecke unendlich. Außerdem ist es selten möglich, eine gewünschte Verschachtelung "einzustellen".

Nebenläufige Programme sind daher nicht erschöpfend zu testen.

Sie laufen i.A. **nicht deterministisch** ab – oft wegen Input von außen (Reaktion auf Signale) zu **unbekannten** Zeiten. (Per definitionem: Interrupts passieren zu unvorhersehbaren Zeitpunkten.) Das Resultat ist eine willkürliche **Verschachtelung** der Programme. Wenn ein Programm "mal richtig, mal falsch" läuft, ist das meist ein Zeitproblem.

Schauen wir uns als Beispiel folgenden Pseudocode in Pascal-S an:

```
program increment;
   const m = 8;
   var n :integer;

   procedure incr;
      var i :integer;
      begin
         for i:= 1 to m do n := n + 1;
      end;

   begin (* Hauptprogramm *)
```

```
   n := 0;
   cobegin
      incr;   incr;
   coend;
   writeln ('Die Summe ist : ', n);
end.
```

Was ist `n` ?

Bemerkungen:

1. Jede Verschachtelung ist gleich wahrscheinlich.
2. Aber wieviele Verschachtelungen ein bestimmtes Resultat liefern, ist sehr unterschiedlich. Das Resultat 10 ist z.B. sehr wahrscheinlich, während 7 schon relativ selten ist. Das Resultat 2 wird man wohl nicht erleben.

Da es praktisch unendlich viele Möglichkeiten der Verschachtelung gibt, können Fehler durch Testen normalerweise nicht ausgeschlossen werden. Fehler zeigen sich bei nebenläufigen Programmen nicht vorhersehbar. Wenn sie schließlich in Erscheinung treten, dann geschieht das oft im ungünstigsten Moment: Das System ist in Produktion und läuft unter hoher Last.

Da man nicht erschöpfend testen kann, sind eigentlich theoretische Beweise notwendig. Diese sind aber häufig zu schwierig oder zu aufwendig. In der Praxis wird man daher nach extremen oder schwierigen **Szenarien** suchen, mit denen ein Programm "theoretisch" getestet wird. Dabei dürfen keine Annahmen über das Zeitverhalten gemacht werden. Man muss **asnychron** denken.

Durch die Quasiparallelität von Einprozessor-Systemen scheinen falsche Programme oft richtig zu laufen, weil es nicht oder nur extrem selten zu einer wirklich willkürlichen Verschachtelung kommt. Man muss aber immer damit rechnen, dass ein Mehrprozessorsystem verwendet wird und es dadurch zu einer **echten Parallelität** kommt! Daher muss jede beliebige Verschachtelung in Betracht gezogen werden!

Die verschiedenen Szenarien haben normalerweise ganz unterschiedliche Wahrscheinlichkeiten für ihr Auftreten. Extreme Szenarien haben oft um viele Zehnerpotenzen kleinere Wahrscheinlichkeiten als "normale" Szenarien und sind daher experimentell kaum sichtbar zu machen.

Wir fassen zusammen:

- Fehler treten auf wegen besonderer zeitlicher Verschachtelungen.
- Die Fehler passieren sehr selten.
- Die Fehler sind schwer zu reproduzieren und zu finden.

Was bedeutet **Korrektheit**? Bei sequentiellen Programmen genügt meistens die richtige Antwort in Grenzfällen und bekannten Fällen.

Bei nebenläufigen Programmen muss darüberhinaus noch die **Sicherheit** und die **Lebendigkeit** gegeben sein.

- **Sicherheit** ***(safety)***: **Kritische Abschnitte** (oft Zugriff auf eine Ressource) ***(critical sections, "read-modify-write")*** müssen sich gegenseitig ausschließen **(Problem des gegenseitigen Ausschlusses,** ***mutual exclusion, "nothing bad ever happens to an object"[12])***. Wenn der Zustand eines Objekts verändert wird, dann muss dafür gesorgt werden, dass das Objekt immer von einem **konsistenten** Zustand in einen anderen konsistenten Zustand überführt wird. Dabei können durchaus **inkonsistente** Zustände **vorübergehend** (transient) auftreten. Das kann allerdings zu Problemen führen, wenn mehrere Tasks das Objekt gleichzeitig verändern.

  Da das Ergebnis von der Reihenfolge abhängen kann, in welcher die Tasks zum Zuge kommen – "den Wettlauf um die Ressource gewinnen", spricht man hier auch von sogenannten ***race conditions***.

  Daher müssen solche Zustandsänderungen unter gegenseitigem Ausschluss durchgeführt werden.

- **Lebendigkeit** ***(liveliness or liveness[12], "something good eventually happens")***: Jede Aktivität muss irgendwie "vorankommen". Lebendigkeit wird durch folgende Szenarien gefährdet:

  - **Verklemmung** ***(deadlock, deadly embrace)***: Das System steht, weil die Tasks gegenseitig aufeinander warten.
  - **Aussperrung** ***(starvation)***: Einzelne Tasks kommen nicht mehr zum Zug.
  - **Verschwörung** ***(conspiracy, livelock)***: Unwahrscheinliche Synchronisations-Szenarien machen die Leistung des Systems unvorhersehbar. D.h. die Tasks verschwören sich zu einem – normalerweise unwahrscheinlichen – blockierenden Zeitverhalten.

  Lebendigkeit kann man auch graduell verstehen. D.h. ein System kann mehr oder weniger performant oder reaktionsfähig oder nebenläufig sein. Sicherheit und Lebendigkeit sind oft gegenläufig. Eine erhöhte Sicherheit kann die Lebendigkeit einschränken und umgekehrt.

Um Lebendigkeit zu gewährleisten gibt es verschiedene Strategien:

- **Entdeckung und Behebung** von Verklemmungen bzw. Aussperrungen ***(detection and recovery)***
- **Verhinderung** ***(prevention)***: Es gibt keine Zustände, die die Lebendigkeit beeinträchtigen.
- **Vermeidung** ***(avoidance)***: Zustände, die die Lebendigkeit beeinträchtigen, werden vermieden.

Bei der Behandlung dieser Probleme durch Synchronisationsmechanismen spielen folgende Begriffe eine wichtige Rolle:

- ***Lock:*** Ein Lock ist ein Konstrukt, um den Zugang zu einem kritischen Bereich zu kontrollieren. Nur *eine* Task kann ein Lock für einen kritischen Bereich bekommen. Der Zugang ist exklusiv.

- ***Permit:*** Ein Permit ist ebenfalls ein Konstrukt, der den Zugang zu einem kritischen Bereich kontrolliert. Allerdings können *mehrere* Tasks ein Permit bekommen (bis zu einer anwendungsbedingten, festgelegten Obergrenze).
- Lock und Permit schließen sich gegenseitig aus. D.h. wenn eine Task ein Lock auf ein Objekt hat, kann keine andere Task ein Permit oder Lock auf dasselben Objekt bekommen. Wenn eine Task ein Permit auf ein Objekt hat, können zwar andere Tasks ebenfalls ein Permit auf dasselbe Objekt bekommen, aber keine andere Task kann ein Lock auf dieses Objekt bekommen.
- ***Block:*** Die genannten Konstrukte werden so eingesetzt, dass jede Task vor Betreten eines kritischen Bereichs versucht, ein Lock oder ein Permit zu bekommen. Ein Block zeigt an, dass die Task auf die Zuteilung eines Locks oder eines Permits wartet. (Es gibt aber auch die Möglichkeit, dass die Task nicht wartet, sondern ohne Betreten des kritischen Bereichs anderen (unkritischen) Code durchführt.)

Nach verlassen des kritischen Bereichs wird das Lock bzw. das Permit wieder freigegeben.

1.3 Übungen

1.3.1 Anzahl Szenarien

Wie groß ist die Anzahl der Szenarien (Verschachtelungen) v bei t Tasks $i = 1 \dots t$, mit jeweils n_i atomaren Anweisungen?

Kapitel 2

Synchronisationsmechanismen

Ein Echtzeit-System (Automatisierungssystem, AS) muss sich mit den Zuständen eines (externen technischen) Prozesses (P) synchronsieren, d.h. muss versuchen, gewisse *interne* Zustände zu erreichen, bevor oder nachdem gewisse externe Prozess-Zustände erreicht werden bzw. wurden. Es wäre kein Problem, einen externen Prozess zu steuern, wenn dessen Fortschreiten von Zustand zu Zustand beliebig durch das Automatisierungssystem verlangsamt oder beschleunigt werden könnte. Aber i.A. gehen die meisten realen Prozesse relativ unkontrollierbar von Zustand zu Zustand.

In diesem Kapitel werden wir uns mit folgenden Synchronisationsmechanismen beschäftigen:

- Zeit
- **Semaphor**
- Bolt-Variable
- **Monitor**
- Rendezvous
- **Channels (Kommunikationskanäle)**

Semaphor und Monitor als die am häufigsten eingesetzten Mechanismen werden wir eingehender behandeln.

2.1 Zeit

Die Verwendung der Zeit als Synchronisations-Mittel bedeutet, dass man Zeit- oder Stundenpläne für die einzelnen Tasks aufstellt. Ein Zeitplan gibt an, wann eine Task was wielange tun soll. Solche Zeitpläne wären ein nützliches Synchronisations-Mittel, wenn reale Prozesse deterministisch ablaufen würden. Das ist aber nicht der Fall. Daher ist die Zeit kein geeignetes Synchronisationsmittel. Im Gegenteil, sie stellt in der Form von einzuhaltenden Zeitplänen eine wichtige Aufgabe der Automatisierung dar, die es zu lösen gilt.

2.2 Semaphore

Für die Synchronisation von Tasks und die Kommunikation zwischen Tasks über Shared Memory hat Dijkstra **Semaphore** (Zeichenträger) eingeführt.
(Der Begriff "Semaphor" ist Neutrum oder Masculinum, also "das" oder "der" Semaphor.)

Semaphore sind ein Synchronisationsmittel, mit dem sogenannte **kritische Abschnitte**, Bereiche oder Gebiete ***(critical sections)*** geschützt werden können. Das sind Codestücke, die nur unter gegenseitigem Ausschluss oder nur in gewisser Reihenfolge betreten werden dürfen.

Semaphore sind folgendermaßen definiert:

- Es gibt eine ganzzahlige Semaphorvariable `s`, die nur **einmal** initialisiert werden kann und für die die folgenden beiden **atomaren (unteilbaren)** und **sich gegenseitig ausschließenden** Systemaufrufe zur Verfügung stehen:
- `p (s) :`
  aufrufende Task wartet bis `s > 0`
  `s = s - 1`
- `v (s) :`
  `s = s + 1`

Bemerkungen:

1. Eine Task wird (eventuell) durch ein Semaphor "gesperrt".
2. Durch ein `v (s)` wird maximal *eine* Task "freigegeben".
3. Das Abfragen des Wertes eines Semaphors ist nicht erlaubt.
4. Ein Semaphor verwaltet typischerweise ein Betriebsmittel, das gesperrt oder freigegeben wird. Es löst das Problem des gegenseitigen Ausschlusses, ohne dass es zu Aussperrung und Verklemmung kommt. Als Beispiel betrachten wir einen Drucker, der nur exklusiv benutzt werden kann.

   `init s = 1`

Task A	Task B	Task C
↓	↓	↓
`p (s)`	`p (s)`	`p (s)`
`drucken`	`drucken`	`drucken`
`v (s)`	`v (s)`	`v (s)`
↓	↓	↓

5. Bezeichnungen: Betriebssysteme und Echtzeit-Sprachen verwenden unterschiedliche Namen für die Semaphor-Operationen. Im Folgenden sind ein paar Beispiele aufgeführt:

`p (s)`	`v (s)`	Dijkstra
passeren	verlaten	Dijkstra
sperren	freigeben	
`s.p ()`	`s.v ()`	objekt-orientiert
`request (s)`	`release (s)`	
`request (s)`	`free (s)`	
`wait (s)`	`signal (s)`	
`take (s)`	`give (s)`	
`semTake (s)`	`semGive (s)`	VxWorks
`down (s)`	`up (s)`	
`pend (s)`	`post (s)`	
`s.acquire ()`	`s.release ()`	Java
Ereignis abwarten	Ereignis senden	Ereignissicht

6. Wesentlich ist, dass die Semaphoroperationen **atomar** ***(atomic, primitive,*** **unteilbar)** sind. Sie laufen im nicht unterbrechbaren Teil eines Betriebssystems.

7. Ohne Semaphore heißt die Lösung "Dekkers Algorithmus" mit den folgenden Nachteilen:

   - Aktives Warten der Tasks.
   - Bei mehr als zwei Tasks wird die Programmierung sehr unübersichtlich.

8. **Binäre Semaphore:** Bisher konnte die Semaphorvariable jeden Wert ≥ 0 annehmen. Daher heißen diese Semaphore auch **allgemeine** oder ***counting*** **Semaphore**. Wenn wir nur die Werte 0 und 1 zulassen, dann sprechen wir von einem binären Semaphor. Im Algorithmus für `v (s)` muss dann `s = s + 1` durch

   ```
   s = 1
   ```

   ersetzt werden.

9. **Allgemeine Semaphore:** Wenn die Semaphor-Variable mit k initialisiert wird, dann kann das entprechende Betriebsmittel von bis zu k Tasks gleichzeitig benutzt werden. Als Beispiel könnte man sich k Drucker vorstellen, die als *ein* Betriebsmittel gesehen werden.

10. Außer der Initialisierung sind keine Zuweisungen an die Semaphorvariable erlaubt.

11. Für die Warteschlange wird keine FIFO-Strategie gefordert, aber auch nicht verboten. Jedenfalls dürfen Programme nicht abhängig von einer Warteschlangen-Strategie sein.

12. Es gibt zahllose Varianten. Zum Beispiel:

    - Atomares Sperren von mehreren Semaphoren, also etwa `p (s1, s2, s3)`.
    - Manche Betriebssyteme oder Sprachen bieten Semaphoroperationen mit Zeitschranken *(timeout)* an.
    - **Additive Semaphore:** Mit den Semaphoroperationen kann das Semaphor um ein Delta verändert werden. Wenn das Semaphor dadurch kleiner Null würde, wird die Task gesperrt.
    - **Semaphorgruppen:** Mit einer Semaphoroperation können für eine Anzahl von Semaphoren Delta-Werte übergeben werden. Falls dabei ein Semaphor kleiner Null würde, wird die Task gesperrt.

Es lohnt sich hier nicht, auf diese Varianten im Detail einzugehen. Das wird erst wichtig, wenn man mit einem konkreten Echtzeit-Betriebssystem oder -Sprache arbeitet. Da allerdings sollte man die Dokumentation sehr genau lesen.

13. **Semaphorproblematik:** Das Semaphor-Konzept bietet Lösungen für zwei Problembereiche, nämlich das

    Problem des **gegenseitigen Ausschlusses** *(competition)*

    und das

    Synchronisationsproblem **"Warten auf ein Ereignis"** und **"Senden eines Ereignisses"** (Ereignissicht, *cooperation)*.

    Beide Problembereiche werden vermischt und das führt zu konzeptionellen Verständnisproblemen. Die bisher definierten Semaphore sind eigentlich nur für die Ereignissicht vernünftig einsetzbar. Für den gegenseitigen Ausschluss sollten die unten definierten MUTEX-Semaphore verwendet werden. Manche Betriebssysteme (z.B. OSEK) kennen daher keine Semaphore, sondern nur MUTEXe (Ressourcen) und Ereignisse (Botschaften).

2.2.1 Synchronisationsbeispiel Schweiß-Roboter

Übung.

2.2.2 MUTEX-Semaphore

Mutual-Exclusion-Semaphore sind eine Erweiterung der binären Semaphore zur Realisierung des gegenseitigen Ausschlusses.

- Verschachtelter Zugriff durch dieselbe Task auf kritische Bereiche ist möglich ***(re-entrant)***. Eine Task, die ein MUTEX-Semaphor belegt, wird **Inhaber** ***(owner)*** des Semaphors. Ein Inhaber eines Semaphors kann das Semaphor **mehrfach** belegen ***(reentrancy, concept of ownership)***. Damit können Programme wesentlich modularer gestaltet werden. Ein "verschachtelter" Zugriff auf kritische Ressourcen ist möglich. Also bei folgendem Code wird sich eine Task nicht selbst aufhängen:

  ```
  init MUTEX-Semaphor  m = freigegeben;

  p (m);
     tu was Kritisches;
     p (m);
        tu noch mehr Kritisches;
     v (m);
  v (m);
  ```

  Bemerkung: Unter Go sind MUTEXe **nicht** re-entrant!

  – MUTEX-Semaphore werden automatisch mit `1` initialisiert.

  - Der Inhaber eines MUTEX-Semphors muss den Semaphor so oft wieder freigeben, wie er ihn belegt hat, damit der Semaphor für andere Tasks freigegeben ist.
  - Nur der Inhaber eines MUTEX-Semaphors kann den Semaphor freigeben.
  - Eine Task, die einen MUTEX-Semaphor besitzt, kann nicht durch eine andere Task gelöscht werden.
  - Im Allgemeinen kann ein MUTEX-Semaphor nicht von einer Interruptserviceroutine belegt oder freigegeben werden.

- **Prioritätsumkehr** wird verhindert.

  Prioritäts-Inversion entsteht, wenn eine hochpriore Task C ein Betriebsmittel benötigt, das gerade eine niedrigpriore Task A besitzt, und eine mittelpriore Task B verhindert, dass Task A läuft, um das Betriebsmittel für C freizugeben.

  In dem folgenden Time-Line-Diagramm ist ein Szenario für das Phänomen der Prioritäts-Inversion angegeben.

  (Bild)

  Es gibt zwei Möglichkeiten, das Problem der Prioritäts-Inversion zu lösen:

  Vererbung von Prioritäten: Wenn z.B. Task C `p(s)` macht, dann erbt Task A die Priorität von Task C, falls diese höher ist, bis Task A `v(s)` macht. Dieser Mechanismus ist häufig an spezielle Semaphore für den gegenseitigen Ausschluss gekoppelt (MUTEX–Semaphore, `synchronized` in Java).

  Priority Ceiling Protocol: Jedes Betriebsmittel, das unter gegenseitigem Ausschluss benutzt wird, bekommt eine "höchste" Priorität ***(ceiling priority)*** (P_c). Diese Priorität P_c sollte mindestens so hoch sein wie die höchste Priorität (P_1) der Tasks, die das Betriebsmittel verwenden. Sie sollte aber kleiner sein als die kleinste Priorität (P_2) der Tasks, die das Betriebsmittel (BM) *nicht* benutzen und die höhere Priorität haben als P_1:

  $P_1 = \max(P)$ über alle Tasks, die BM benutzen
  $P_2 = \min(P)$ über alle Tasks mit $P > P_1$, die BM nicht benutzen
  $P_1 \leq P_c < P_2$ (logisch)

  Wenn eine Task ein Betriebsmittel benutzt, dann wird ihre Priorität während dieser Zeit auf die P_c des Betriebsmittels gesetzt, falls ihre eigene Priorität kleiner war.

  Dieses Schema hat den Vorteil, dass eine niedrigpriore Task bei Benutzung des Betriebsmittels sofort mit höherer Priorität läuft und dadurch das BM früher wieder freigibt. Es hat den Nachteil, dass es weniger dynamisch ist.

- Die Verwaltung der MUTEX-Semaphore verursacht einen größeren internen Zeitaufwand als die Verwaltung binärer Semaphore.

Verklemmung: Wenn mehr als ein Betriebsmittel exklusiv beantragt wird, dann kann es zu Verklemmungen ***(deadlock)*** kommen.

Wir nehmen an, dass wir einen Drucker und einen Plotter haben, die gemeinsam exklusiv benutzt und daher durch die MUTEX-Semaphore `md` und `mp` geschützt werden sollen:

```
init md = 1
init mp = 1
```

Task A	Task B
↓	↓
`p (md)`	`p (mp)`
`p (mp)`	`p (md)`
`drucken`	`drucken`
`plotten`	`plotten`
`v (md)`	`v (mp)`
`v (mp)`	`v (md)`
↓	↓

Wenn Task A nach seinem erfolgreichen `p (md)` suspendiert wird, und dann Task B ein erfolgreiches `p (mp)` machen kann, werden beide durch die nächste Semaphoroperation gesperrt.

Das kann nur vermieden werden, wenn man fordert, dass immer in der gleichen Reihenfolge gesperrt wird:

```
init md = 1
init mp = 1
```

Task A	Task B
↓	↓
`p (md)`	`p (md)`
`p (mp)`	`p (mp)`
`drucken`	`drucken`
`plotten`	`plotten`
`v (md)`	`v (mp)`
`v (mp)`	`v (md)`
↓	↓

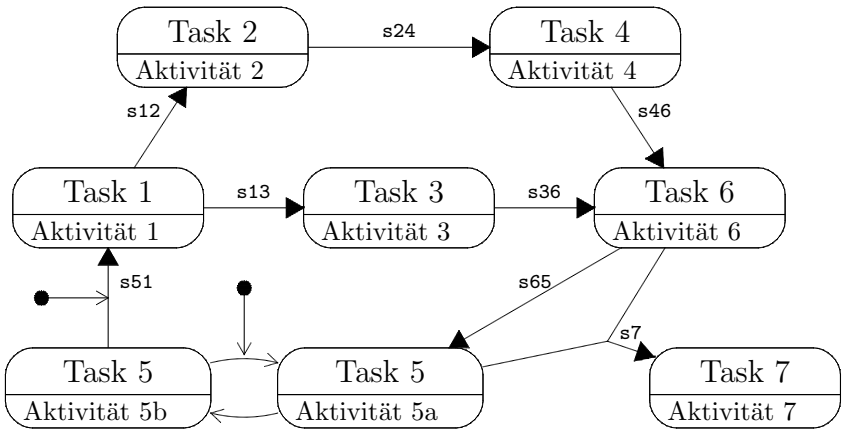
Deswegen bieten manche Systeme das atomare Sperren von mehr als einem Semaphor an (`p (md, mp)`).

2.2.3 Herstellung von Ausführungsreihenfolgen

Für die Ereignis-Sicht (Warten-auf-und-Senden-von-Ereignissen) werden die normalen Semaphoren verwendet. Damit werden meistens Abläufe gesteuert.

Wenn Aktivitäten von verschiedenen Tasks durchgeführt werden, dann muss man eventuell die Reihenfolge dieser Tasks festlegen, um sogenannte ***race conditions*** zu vermeiden. Eine Race-Condition liegt vor, wenn das Ergebnis davon abhängt, welche Task das Rennen bei nebenläufigen Aktivitäten gewinnt (z.B. beim Zugriff auf gemeinsamen Speicher).

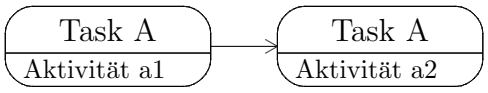
Im Folgenden **Ablauf-Diagramm** wird der Ablauf durch Ovale und Pfeile definiert.



Grafische Notation:

- Die Ovale enthalten einen Tasknamen und eine Aktivität.
- Das Ablauf-Diagramm ist so zu verstehen, dass alle Tasks gleichzeitig loslaufen, aber einige eventuell durch Synchronisationsmechanismen gesperrt werden.

  Typischerweise laufen die Tasks in einer Schleife, ohne dass das notwendigerweise im Diagramm angezeigt wird.
- Die Ovale werden durch einen oder mehrere Pfeile verbunden.
- Wenn ein Pfeil Ovale **ein- und derselben** Task verbindet (Beispiel Task 5), dann wird er mit einfacher Spitze notiert.



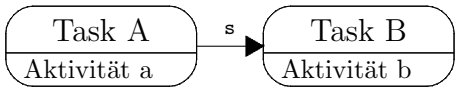
  Im Code der Task A werden die beiden Aktivitäten in der durch den Pfeil angedeuteten Reihenfolge kodiert:

```
Task A:
// ...
Aktivität a1
Aktivität a2
// ...
```

- Wenn ein Pfeil zwei Ovale **unterschiedlicher** Tasks verbindet, wird ein Pfeil mit "gefüllter" Pfeilspitze verwendet. Er wird durch einen

  Warte-Auf-Sende-Ereignis-Mechanismus

implementiert. Dazu können wir Semaphore oder Monitore verwenden. Eine Bezeichnung für das Synchronisationsmittel kann auf dem Pfeil angebracht werden.



Diese Notation bedeutet, dass die Task B ihre `Aktivität b` erst nach Beendigung der `Aktivität a` in Task A beginnen darf.

Im Folgenden nehmen wir allgemeine Semaphore als Synchronisationsmittel:

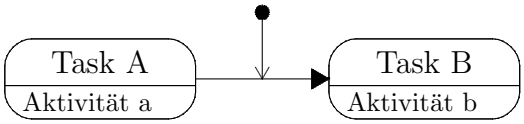
> Jeder Pfeil zwischen unterschiedlichen Tasks wird mit einem allgemeinen Semaphor implementiert.

Die Pfeilspitze bedeutet eine `p`-Operation ("warte auf Ereignis") am Anfang der Aktivität der Zieltask. Die Aktivität der Ausgangstask eines Pfeiles macht am Ende eine `v`-Operation ("sende Ereignis").

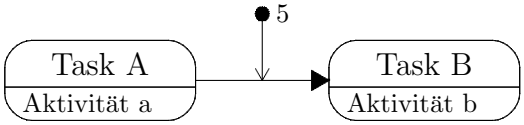
```
Task A:              Task B:
// ...               // ...
Aktivität a          p (s)
v (s)                Aktivität b
// ...               // ...
```

Die Semaphore werden normalerweise mit 0 initialisiert. Das bedeutet, dass Task B vor Aktivität b gesperrt wird.

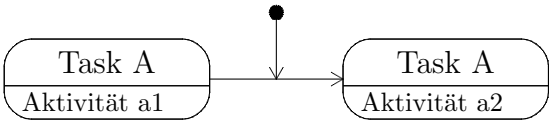
- Damit es überhaupt "losgeht", müssen eventuell eine oder mehrere Start-Aktivitäten definiert werden. Im Beispiel oben ist das "Aktivität 1" der Task 1. Das bedeutet, dass der oder die Semaphore, die den Eintritt in die Start-Aktivität kontrollieren, mit 1 initialisiert werden. Grafisch wird das mit einem "Pseudozustand" angegeben:



Wenn ein Semaphor mit `>1` initialisiert wird, dann schreiben wir den Wert an den Pseudozustand:

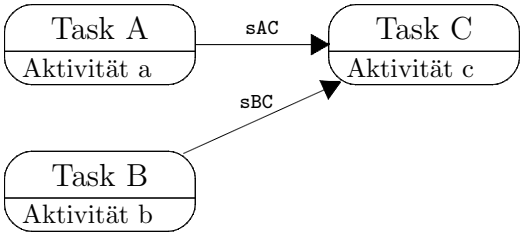


Diese Notation wird auch verwendet, um anzugeben, mit welcher Aktivität innerhalb *einer* Task begonnen wird (Beispiel Task 5).



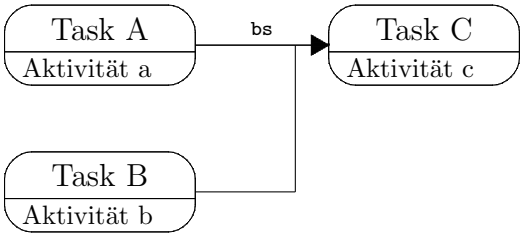
Hier wäre mit `Aktivität a2` zu beginnen.

- Wenn mehrere Pfeile in ein Oval gehen, dann bedeutet das eine "UND"-Verknüpfung. Im oben gezeigten Beispiel kann die `Aktivität 6` der Task 6 erst beginnen, wenn sowohl `Aktivität 4` als auch `Aktivität 3` fertig sind.




```
Task A:              Task B:              Task C:
// ...               // ...               // ...
Aktivität a          Aktivität b          p (sAC)
v (sAC)              v (sBC)              p (sBC)
// ...               // ...               Aktivität c
// ...               // ...               // ...
```

- Bei eine "ODER"-Verknüpfung vereinigen wir die beiden Pfeile vorher (Beispiel Task 7) und realisieren diese Pfeile – im einfachsten Fall! – durch einen gemeinsamen Binären Semaphor.




```
Task A:              Task B:              Task C:
// ...               // ...               // ...
Aktivität a          Aktivität b          p (bs)
v (bs)               v (bs)               Aktivität c
// ...               // ...               // ...
```

Die "ODER"-Verknüpfung ist eher untypisch. Die oft sehr komplexe Semantik lässt sich grafisch nicht vernünftig darstellen. Abhängig vom gewünschten Verhalten der beteiligten Tasks kann die Realisierung sehr kompliziert werden, d.h. eine **Erzeuger-Verbraucher**-Semantik (siehe folgenden Abschnitt) muss definiert und implementiert werden. Eventuell muss man zusätzliche Tasks definieren, um das gewünschte Ergebnis zu erhalten.

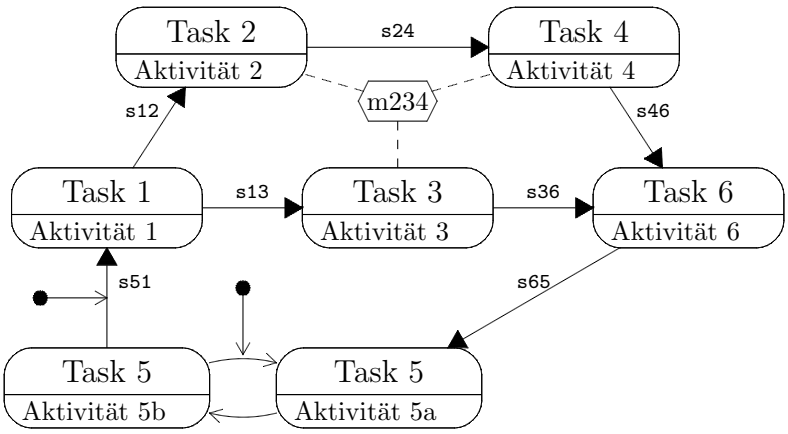
Im oben angegebenen Beispiel würden die Semaphore `s12`, `s13`, `s24`, `s36`, `s46`, `s65`, `s51` und `s7` definiert werden. Der Semaphor `s51` müsste mit 1, alle andern mit 0 initialisiert werden.

Die Tasks müssten dann folgendermaßen programmiert werden:

```
Task 1:        Task 2:        Task 3:        Task 4:        Task 5:
// ...         // ...         // ...         // ...         // ...
p (s51)        p (s12)        p (s13)        p (s24)        p (s65)
Aktivität 1    Aktivität 2    Aktivität 3    Aktivität 4    Aktivität 5a
v (s12)        v (s24)        v (s36)        v (s46)        v (s7)
v (s13)        // ...         // ...         // ...         Aktivität 5b
// ...         // ...         // ...         // ...         v (s51)
// ...         // ...         // ...         // ...         // ...
```

```
Task 6:        Task 7:
// ...         // ...
p (s36)        p (s7)
p (s46)        Aktivität 7
Aktivität 6    // ...
v (s65)        // ...
v (s7)         // ...
// ...         // ...
```

Bei den Ablauf-Diagrammen werden Bereiche des gegenseitigen Ausschlusses normalerweise nicht berücksichtigt. Wenn man das ebenfalls graphisch darstellen möchte, dann kann man die sich gegenseitig ausschließenden Bereiche gestrichelt mit der betreffenden Kontrollinstanz (etwa `m234`) verbinden. Das kann ein (MUTEX-)Semaphor oder ein Monitor oder sonst ein Lock sein. Im folgenden Beispiel würden sich die Aktivitäten 2, 3 und 4 gegenseitig ausschließen.



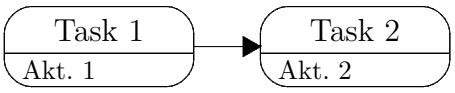
Dabei würden die drei Aktivitäten 2, 3 und 4 jeweils in MUTEX-Operationen verpackt werden:

```
p (m234)
Aktivität x
v (m234)
```

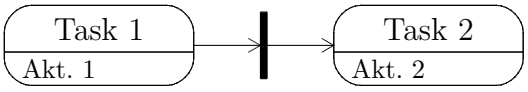
Bemerkungen:

Task-Rahmen: Beispielprogramme zum Testen solcher Abläufe lassen sich relativ einfach in Java schreiben unter Verwendung eines Task-Rahmens (`www.dhbw-stuttgart.de/~kfg/pdv/taskRahmen.zip`).

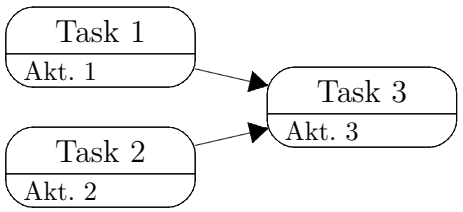
UML-Notation: Als UML-Notation müsste das Aktivitäts-Diagramm verwendet werden:



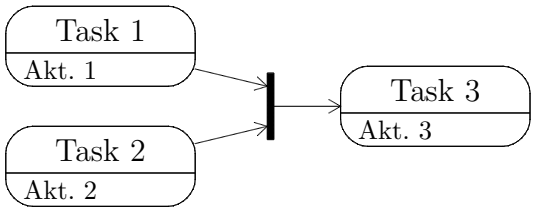
UML-Aktivitäts-Diagramm:



Oder:



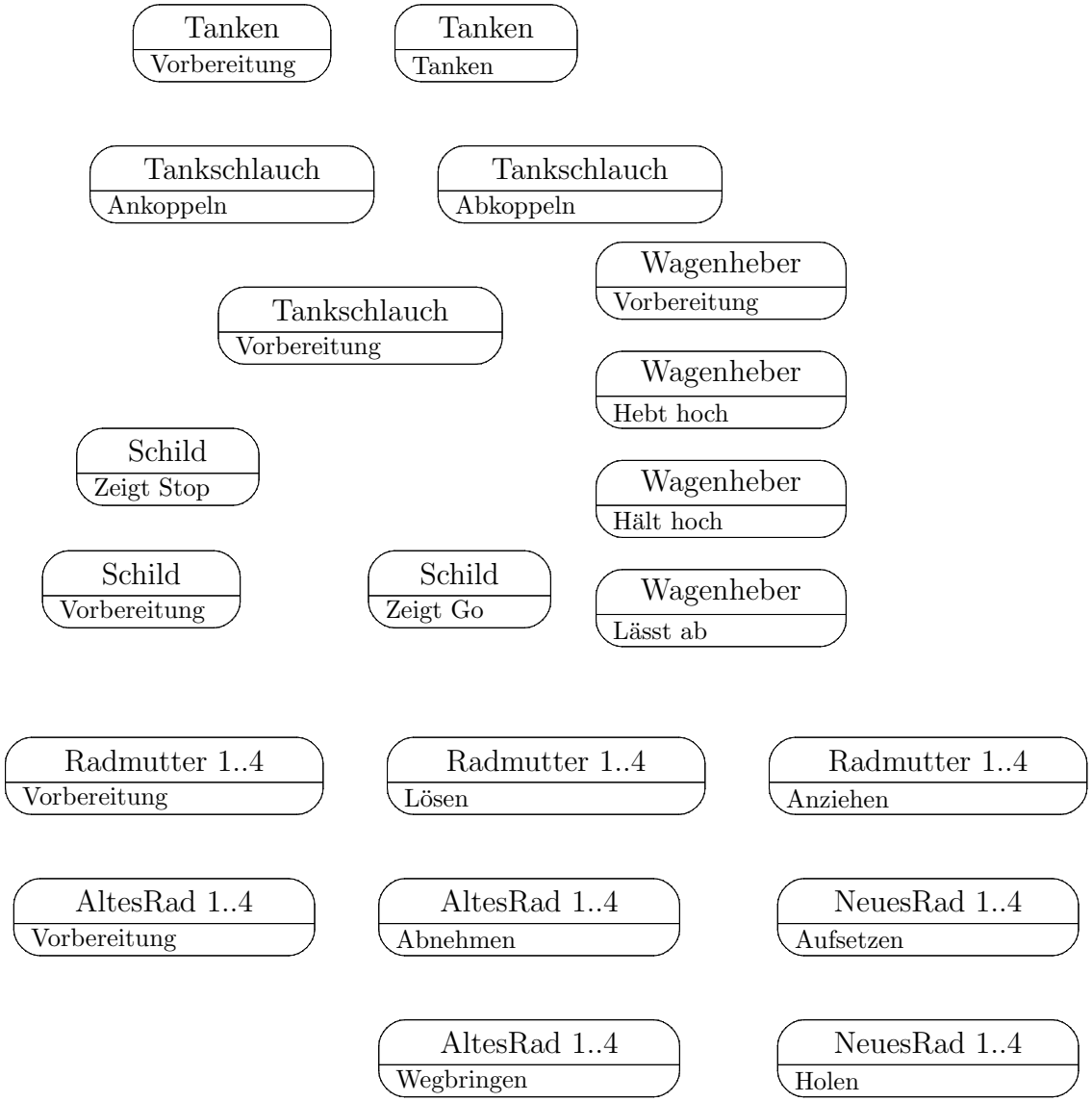
UML–Aktivitäts–Diagramm:



Allerdings werden durch die UML-Notation manche Diagramme ziemlich überladen.

Übung Boxenstop

Vervollständigen Sie das folgende Ablauf-Diagramm durch Pfeile.



Übung Holzhacken

Roboter hacken Holz. Folgende Aktivitäten kann man die Roboter ausführen lassen:

`Holen` (Roboter holt ein zu zerhackendes Holzstück.)
`Vorlegen` (Roboter legt das Holzstück auf dem Hackblock vor.)

`Ausholen` (Roboter holt mit der Axt aus.)
`Hacken` (Roboter schlägt mit der Axt zu.)
`Freimachen` (Roboter macht den Hackblock frei von Holzscheiden.)
`Schichten` (Roboter schichtet Holzscheide auf.)

In jedem Fall gibt es nur **einen** Hackblock!

Entwerfen Sie Ablaufdiagramme für

einen
zwei
drei

Roboter, d.h. Tasks.

2.2.4 Erzeuger-Verbraucher-Problem

Ein wichtiges Element der Kooperation zwischen Tasks ist der Austausch von Daten. Zur Verdeutlichung der Problematik betrachten wir folgendes Beispiel:

Die Task `E` schickt der Task `V` die Koordinatenwerte (x, y, z) der Zielposition eines Roboterarms, indem drei gemeinsame Speicherplätze verwendet werden. Ein Szenario ist:

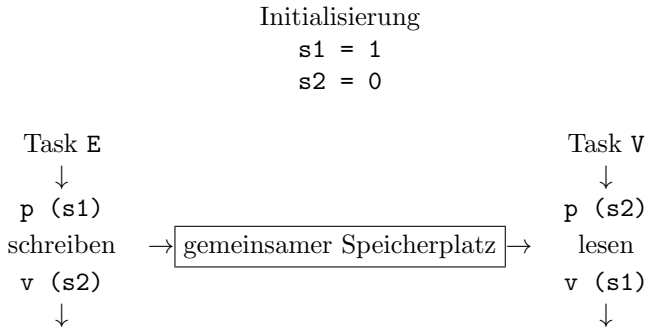
`E` schreibt x_1, y_1, z_1.
`V` liest x_1, y_1.
`V` wird suspendiert.
`E` schreibt neue Werte x_2, y_2, z_2.
`V` liest weiter z_2. $\Longrightarrow$ `V` hat inkonsistenten Datensatz x_1, y_1, z_2.

Vor einer Lösung des Problems muss die Art des Datenaustauschs geklärt werden:

- a) Soll jeder geschriebene (erzeugte) Datensatz auch gelesen (verbraucht) werden?
- b) Sollen dabei Schreiben und Lesen zeitlich entkoppelt werden?
- c) Soll nur der jeweils aktuelle Datensatz (eventuell öfter) gelesen werden?
- d) Soll der jeweils aktuelle Datensatz nur gelesen werden, wenn geschrieben wurde?
- e) Soll das Reader-Writer-Problem gelöst werden (siehe unten)?

Lösungen mit Semaphoren:

- zu a) Jeder geschriebene (erzeugte) Datensatz soll auch gelesen (verbraucht) werden.

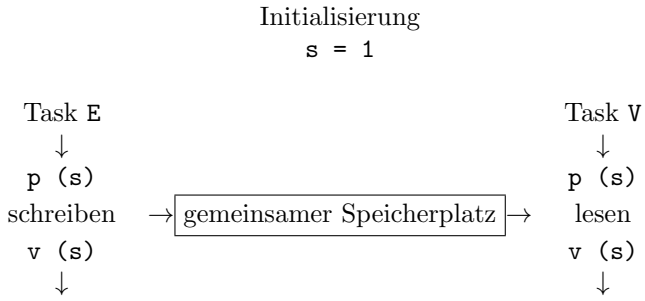


Die beiden Tasks sind zeitlich eng gekoppelt.

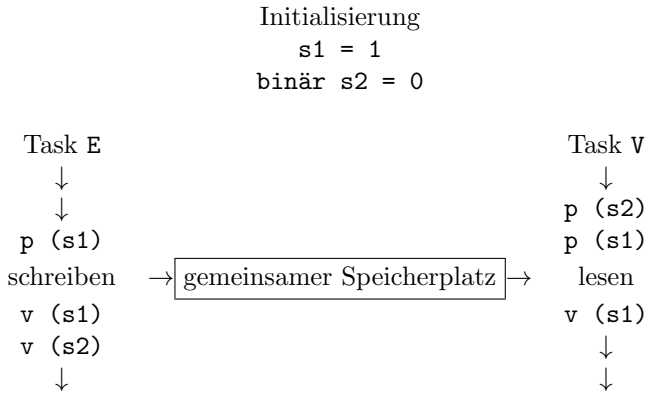
- zu b) Schreiben und Lesen sollen zeitlich entkoppelt werden. Zeitliche Entkopplung ist nur möglich über einen Ringpuffer:

Dazu mehr in einem späteren Abschnitt.

- zu c) Nur der jeweils aktuelle Datensatz soll (eventuell öfter) gelesen werden.



- zu d) Nur der jeweils aktuelle Datensatz soll gelesen werden, wenn geschrieben wurde.

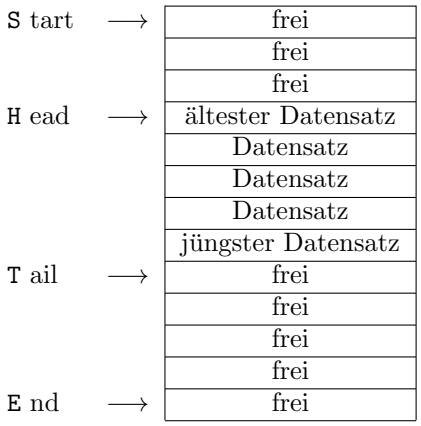


- zu e) Das Reader-Writer-Problem wird in einem späteren Abschnitt behandelt.

Ringpuffer

Ein Ringpuffer (Zirkularpuffer, Pipe, Kanal) ist ein Speicherbereich mit `L` Speicherplätzen `0 ... L-1` für `L` Datensätze und zwei Methoden: Eine Methode `enqueue` (lege ab, schreiben), um Datensätze in den Ringpuffer zu schreiben, und eine Methode `dequeue` (entnehme, lesen), um Datensätze aus dem Ringpuffer zu entnehmen. Die Entnahme-Strategie ist FIFO. Der Zustand des Ringpuffers wird durch folgende Größen beschrieben:

`L` : Anzahl der Speicherplätze `0 ... L-1`
`N` : Anzahl der belegten Speicherplätze
`T` : Nummer des ersten freien Speicherplatzes, auf den ein Datensatz abgelegt werden kann ***(Tail***, Schwanz).
`H` : Nummer des ersten belegten Speicherplatzes, aus dem ein Datensatz entnommen werden kann ***(Head***, Kopf).
`S` : (`= 0`) Anfang (Start) des Ringpuffers
`E` : (`= L-1`) Ende des Ringpuffers



Initialisierung des Ringpuffers:

```
H = T = S
N = 0
```

Methode `enqueue (D)` (`D` sei ein Datensatz.):

```
void  enqueue (D)
   {
   Warte bis N < L;
   Schreibe nach T;
   if T == E then T = S;
   else T = T ⊕ 1;
```

```
   N = N + 1;
   }
```

(Mit "$\oplus$" meinen wir eben "zur nächsten Adresse".)

Methode `dequeue ()`:

```
D  dequeue ()
   {
   Warte bis N > 0;
   Lies von H;
   if H == E then H = S;
   else H = H ⊕ 1;
   N = N - 1;
   }
```

Nun wollen wir versuchen, das "Warten" mit Semaphoren zu realisieren. Außerdem muss erreicht werden, dass die Verwaltung von `N` nicht durcheinander kommt, wenn wir nebenläufige Schreiber und Leser haben.

1. Versuch

```
init  binärer Semaphor s = 1;

void  enqueue (D)
   {
   while N == L do nothing;
   Schreibe nach T;
   if T == E then T = S;
   else T = T ⊕ 1;
   p (s);
   N = N + 1;
   v (s);
   }

D  dequeue ()
   {
   while N == 0 do nothing;
   Lies von H;
   if H == E then H = S;
   else H = H ⊕ 1;
   p (s);
   N = N - 1;
   v (s);
```

```
   }
```

`N` ist zwar geschützt, aber 1.) wird aktiv gewartet und 2.) käme die Verwaltung von `H` und `T` durcheinander, wenn wir mehrere Schreiber bzw. Leser hätten.

2. Versuch

```
init  binärer Semaphor s = 1;

void  enqueue (D)
   {
   while N == L do nothing;
   p (s);
   Schreibe nach T;
   if T == E then T = S;
   else T = T ⊕ 1;
   N = N + 1;
   v (s);
   }

D  dequeue ()
   {
   while N == 0 do nothing;
   p (s);
   Lies von H;
   if H == E then H = S;
   else H = H ⊕ 1;
   N = N - 1;
   v (s);
   }
```

Auch hier wird aktiv gewartet. Ferner könnte es sein, dass z.B. bei `N == 1` zwei Leser über die Warte-Anweisung kommen und dann versuchen zwei Datensätze zu entnehmen, obwohl nur einer vorhanden ist. (Das Problem gibt es natürlich auch schon bei Versuch 1.)

3. Versuch

```
init  binärer Semaphor s = 1;

void  enqueue (D)
   {
   p (s);
   while N == L do nothing;
   Schreibe nach T;
   if T == E then T = S;
   else T = T ⊕ 1;
   N = N + 1;
   v (s);
   }
```

```
D  dequeue ()
   {
   p (s);
   while N == 0 do nothing;
   Lies von H;
   if H == E then H = S;
   else H = H ⊕ 1;
   N = N - 1;
   v (s);
   }
```

Das löst zwar alle Probleme des gegenseitigen Ausschlusses. Aber das System ist tot. Denn wenn eine Task mal in einer Warteschleife ist, dann wird jede andere Task durch `p (s)` gesperrt. Wir müssen also immer irgendwie mal eine andere Task in den kritischen Bereich lassen.

4. Versuch

```
init  binärer Semaphor s = 1;

void  enqueue (D)
   {
   p (s);
   while N == L { v (s); do nothing; p (s); }
   Schreibe nach T;
   if T == E then T = S;
   else T = T ⊕ 1;
   N = N + 1;
   v (s);
   }

D  dequeue ()
   {
   p (s);
   while N == 0 { v (s); do nothing; p (s); }
   Lies von H;
   if H == E then H = S;
   else H = H ⊕ 1;
   N = N - 1;
   v (s);
   }
```

Das müsste einigermaßen funktionieren. Allerdings haben wir immer noch eine aktive Warteschleife. Um das zu lösen, führen wir noch zwei weitere binäre Semaphore ein:

5. Versuch

```
init  binärer Semaphor s = 1;
```

```
init  binärer Semaphor s1 = 0;
init  binärer Semaphor s2 = 0;

void  enqueue (D)
   {
   p (s);
   while N == L { v (s); p (s1); p (s); }
   Schreibe nach T;
   if T == E then T = S;
   else T = T ⊕ 1;
   N = N + 1;
   v (s2);
   v (s);
   }

D  dequeue ()
   {
   p (s);
   while N == 0 { v (s); p (s2); p (s); }
   Lies von H;
   if H == E then H = S;
   else H = H ⊕ 1;
   N = N - 1;
   v (s1);
   v (s);
   }
```

Es ist sehr schwer und umständlich zu beweisen, dass dieser Algorithmus tatsächlich funktioniert.

Derartige Probleme können einfacher und sicherer mit dem Monitor-Konzept programmiert werden.

Übung: Welcher Semaphor kann/sollte in den oben genannten Beispielen ein MUTEX sein?

6. Versuch

Wir versuchen hier nochmal eine elegante Lösung mit zwei allgemeinen Semaphoren, die die Verwaltung von `N` übernehmen, und zwei binären Semaphoren, die die Ringpuffer-Zeiger-Verwaltung schützen:

```
init  allgemeiner Semaphor sl = L;
init  allgemeiner Semaphor sn = 0;
init  Binärer Semaphor sh = 1;
init  Binärer Semaphor st = 1;

void  enqueue (D)
   {
   p (sl);
   p (sh);
   Schreibe nach H;
```

```
    if H == E then H = S;
    else H = H ⊕ 1;
    v (sn);
    v (sh);
    }

D   dequeue ()
    {
    p (sn);
    p (st);
    Lies von T;
    if T == E then T = S;
    else T = T ⊕ 1;
    v (sl);
    v (st);
    }
```

2.2.5 Emulation von Semaphoren in Java

Siehe Skriptum Java bzw. Kapitel Threads.

2.3 Bolt-Variable — Reader-Writer-Problem

Reader-Writer-Problem: Es gibt einen gemeinsamen Speicherbereich, für den folgende Regeln gelten sollen:

1. Schreiber haben exklusiven Zugriff. D.h. höchstens ein Schreiber darf schreiben. Wenn ein Schreiber schreibt, darf kein Leser lesen.
2. Viele Leser können gleichzeitig lesen. Wenn Leser lesen, darf kein Schreiber schreiben.
3. Solange Leser lesen, werden neue Leser zum Lesen zugelassen.

Dieses Problem ist mit Semaphoren sehr umständlich zu lösen. Die Sprache PEARL hat deswegen die Bolt-Variable erfunden. Datenbanksyteme kennen X- und S-Locks für die Lösung dieses Problems.

Eine Bolt-Variable `B` hat drei Zustände

```
reserved    (gesperrt)
free        (Sperre möglich)
entered     (Sperre nicht möglich)
```

und einen Zähler `Z` für die gerade mitbenutzenden Tasks und vier **atomare** und **sich gegenseitig ausschließende** Anweisungen:

```
reserve (B):   aufrufende Task wartet bis B == free
               B = reserved

free (B):      B = free

enter (B):     aufrufende Task wartet bis B != reserved
               B = entered
               Z = Z + 1

leave (B):     Z = Z - 1
               if (Z == 0) B = free
```

Schreiber schützen ihre Schreib-Operation mit der Kombination:

```
reserve (B);
   Schreiben;
free (B);
```

Leser schützen ihre Lese-Operation mit der Kombination:

```
enter (B);
   Lesen;
leave (B);
```

2.3.1 Implementierung mit dem Java-Monitor

Übung Bolt-Variable:

```
import java.io.*;
public interface  BoltVariable
   {
   void  reserve ();
   void  free ();
   void  enter ();
   void  leave ();
   }
```

Realisieren Sie diese Schnittstelle in einer Klasse `BoltVariableI`.

2.3.2 Reader-Writer-Problem mit Writer-Vorrang

Eine meist sinnvolle Variante des Reader-Writer-Problems ist es, den Schreibern Vorrang zu geben. D.h., wenn ein Schreiber schreiben will, wird kein weiterer Leser zugelassen.

Die Lösung nur mit Semaphoren wird hier schon recht kompliziert.

Wenn man allerdings die Bolt-Variable verwendet, dann ist man auch noch auf Semaphore angewiesen und es wird eigentlich zu komplex.

Insgesamt sind Bolt-Variable kein nützliches Konzept und wurden hier eigentlich nur behandelt, um das Reader-Writer-Problem vorzustellen, das ein interessantes Beispiel für den Einsatz von Semaphoren, Monitoren und anderen Synchronisationsmechanismen ist.

2.4 Monitore

Da Semaphoroperationen entfernte Wirkungen haben können, widerspricht dies dem Konzept der Strukturierung. Räumlich und zeitlich im Entwicklungsprozess eines Programms voneinander entfernte Prozeduren können Semaphoroperationen benützen. Das Semaphor gilt als das "Goto" der nebenläufigen Programmierung.

Der Monitor dagegen ist ein *strukturiertes* Synchronisationswerkzeug, mit dem ein Programmbereich definiert wird, der zu einer Zeit nur von *einer* Task oder *einem* Prozess betreten bzw. benutzt werden darf. Damit kann z. B. gegenseitiger Ausschluss implementiert werden.

Bei einem *monolithischen* Monitor werden alle kritischen Phasen in *einem* Monitor zentralisiert. Bei *dezentralisierten* Monitoren hat jeder Monitor eine spezielle Aufgabe mit geschützten Daten und Funktionen.

Ein Monitor hat folgende Struktur:

```
Monitor
   Folge von Datenvariablen
   Folge von Prozeduren
   Körper zur Initialisierung
```

Auf die im Monitor definierten Daten kann nur über die Prozeduren des Monitors zugegriffen werden. Der Körper wird bei Start des Programms einmal durchlaufen. Jedem Monitor sind in Pascal-S ein oder mehrere Bedingungsvariablen `c` zugeordnet, auf die mit den Proceduren `wait (c)`, `signal (c)` und `nonempty (c)` zugegriffen werden kann:

`wait (c)` : Der aufrufende Prozess wird blockiert und in eine Warteschlange (FIFO-Strategie) von Prozessen eingeordnet, die ebenfalls wegen `c` blockiert sind. Die Ausführung von `wait (c)` gibt den Eintritt in den Monitor wieder frei.

`signal (c)` : Falls die Warteschlange von `c` nicht leer ist, wird der erste Prozess in der Warteschlange von `c` erweckt. Danach muss der Monitor sofort verlassen werden. "Ich signalisiere, dass `c` eingetreten ist."

`nonempty (c)`: Ist `true`, falls die Warteschlange nicht leer ist.

Da Monitore das Synchronisationskonzept in Java sind, findet sich eine weitergehende Diskussion mit Beispielen im Java-Skriptum bzw. im Kapitel Threads.

2.5 Rendezvous

Das Rendezvous basiert auf Ideen von Hoare[8] und ist der Synchronisationsmechanismus, den die Echtzeitsprache **Ada** zur Verfügung stellt.

Ada entstand als ein Abkömmling von Pascal im Auftrag des Department of Defense der USA zwischen 1975 und 1980 in Konkurrenz zwischen mehreren Firmen. Durchgesetzt hat sich die Gruppe um Jean Ichbiah von CII Honeywell Bull in Frankreich. Der erste Standard war ANSI MIL-STD-1815. Benannt ist die Sprache nach Augusta Ada Byron, Countess of Lovelace, die als die welterste ProgrammiererIn gilt und mit Charles Babbage zusammenarbeitete.

Das Rendezvous-Konzept wird unter der Bezeichnung ***Transaction*** auch in der von Gehani entwickelten nebenläufigen Erweiterung von C/C++ ConcurrentC/C++ (CCC) als Synchronisationsmechanismus verwendet. Wir verwenden einen Pseudocode, der an die Syntax und Semantik von CCC angelehnt ist, um die grundlegenden Mechanismen des Rendezvous vorzustellen.

Das Rendezvous ist entwickelt worden für eine typische Client/Server-Situation (im generischen Sinn). Dazu ein Beispiel:

Ein Friseur bietet den Dienst `Haarschneiden` an. Er ist allein und kann daher nur einen Kunden gleichzeitig bedienen. Wenn es mehrere Kunden gibt, dann müssen diese warten. Andererseits, wenn es keine Kunden gibt, dann muss er warten. Mit Rendezvous bzw. Transaction würden wir das folgendermaßen programmieren:

```
process body Friseur ()
   {
   while (true)
      {
      // Bereitet sich auf den nächsten Kunden vor

      accept Haarschneiden (Wunsch)
         {
         // Versucht den Wunsch zu erfüllen.
         // Überlegt sich den Preis, den er verlangen kann.
         Preis = "14,50";
         treturn Preis;
         }  // end accept

      // Kehrt die Haare zusammen.
      }  // end while
   }  // end process

process body Kunde (Friseur friseur)
   {
   char* Wunsch;
   char* Preis;
   // Überlegt sich, welche Haartracht er wünscht.
   Wunsch = "top modern";

   Preis = friseur.Haarschneiden (Wunsch);
```

```
   // Verarbeitet den Preis und Haarschnitt.
   }
```

`Friseur` und `Kunde` sind beide nebenläufig laufende Tasks (in CCC `process` genannt). Das Rendezvous ist ein Mechanismus, bei dem die beiden Tasks an der sogenannten `accept`-Prozedur – hier Haarschneiden – aufeinander warten. Charakteristisch ist die **Unsymmetrie**. Der Kunde muss bei `Haarschneiden` warten, bis der Friseur an das `accept Haarschneiden` gekommen ist. Umgekehrt muss der Friseur am `accept Haarschneiden` warten, bis der Kunde `Haarschneiden` aufruft. Für den Kunden verhält sich das Rendezvous ähnlich wie ein Funktionsaufruf. Er wird allerdings währenddessen (d.h. während er auf die Durchführung der Prozedur warten muss und während der Durchführung selbst) suspendiert. Der Friseur macht die eigentliche Arbeit in seiner Umgebung und in seinem Task-Kontext.

Die `accept`-Prozedur kann Argumente und einen Rückgabewert haben.

Mit dem einfachen Rendezvous kann schon das Problem des gegenseitigen Ausschlusses realisiert werden. Typischerweise gibt es viele Kunden, für die das Haarschneiden unter gegenseitigem Ausschluss durchgeführt wird.

Ein Server kann durchaus mehrere `accept`-Routinen anbieten:

```
process body Friseur ()
   {
   while (true)
      {
      // Bereitet sich auf den nächsten Kunden vor

      accept Haarschneiden (Wunsch)
         {
         treturn Preis;
         }  // end accept

      // Kehrt die Haare zusammen.

      accept Rasieren ()
         {
         treturn Preis;
         }  // end accept

      // Reinigt das Waschbecken.
      }  // end while
   }  // end process
```

Man kann damit gewisse Reihenfolgen von Operationen, die unter gegenseitigem Ausschluss durchzuführen sind, fest vorgeben. In unserem Beispiel ist das nicht besonders sinnvoll, da der Friseur unbedingt zwischen einem Haarschneide-Kunden und einem Rasier-Kunden abwechseln müsste. Aber in dem Beispiel des Schweiss-Roboters wäre das durchaus sinnvoll, wenn man

das Positionieren und Schweissen in einer Server-Task als `accept`-Routinen unterbringen würde. Dann wäre damit die Reihenfolge und der gegenseitige Ausschluss garantiert.

Aber unser Friseur hätte gern die Möglichkeit, unter den `accept`s auswählen zu können. Dafür gibt es den `select`-Mechanismus:

```
process body Friseur ()
   {
   while (true)
      {
      // Bereitet sich auf den nächsten Kunden vor

      select
         {
         accept Haarschneiden (Wunsch) {treturn Preis;}
         or accept Rasieren () {treturn Preis;}
         or accept Bartschneiden () {treturn Preis;}
         or delay 1 Minute ; LiestKurzZeitung ();
         }

      // Räumt auf.
      }  // end while
   }  // end process
```

Der Friseur hat jetzt seinen Synchronisationspunkt am `select`. Dort schaut er, ob es einen Kunden in der Warteschlange gibt, der einen Dienst aufgerufen hat. Wenn es einen solchen Kunden gibt, dann wird er diesen Kunden bedienen. Oder er wartet höchstens eine Minute, ob innerhalb dieser Minute ein Kunde kommt. Wenn kein Kunde kommt, geht er in die `delay`-Alternative und liest etwas Zeitung, ehe er dann aufräumt und sich wieder auf den nächsten Kunden vorbereitet, was beides ja dann recht kurz sein kann oder entfällt.

Das Rendezvous in CCC bietet noch mehr Möglichkeiten, die ohne weiteren Kommentar im Folgenden zusammengefasst sind.

Accept-Statement :

`accept` Transaktionsname`(`Argumente$_{opt}$`)`
`suchthat(`Bedingungsausdruck`)`$_{opt}$
`by (`Prioritätsausdruck`)`$_{opt}$
Block (Körper des Accept)

Ohne `suchthat` und `by` wird eine FIFO-Warteschlange der wartenden Transaktionen gebildet.

Mit `by` werden die Prioritätsausdrücke für alle wartenden Transaktionen ausgewertet. Die Transaktion mit dem *Minimum* wird angenommen.

Mit `suchthat` werden nur solche Transaktionen berücksichtigt, deren Bedingungsausdruck ungleich Null ergibt.

Select-Statement :

```
select
   {
      (Bedingung-1):opt     Alternative-1
   or (Bedingung-2):opt     Alternative-2
   ...
   or (Bedingung-n):opt     Alternative-n
   }
```

Die Alternativen sind eine Folge von Statements. Das erste Statement einer Alternative bestimmt die Art der Alternative. Es werden vier Arten von Alternativen unterschieden:

- `accept`-Alternative: `accept`-Statement eventuell gefolgt von anderen Statements.
- `delay`-Alternative: `delay`-Statement eventuell gefolgt von anderen Statements.
- `terminate`-Alternative: `terminate;`
- `immediate`-Alternative: alle anderen Statements

Die Bedingungen entscheiden, ob eine Alternative offen oder geschlossen ist. Mindestens eine Bedingung muss offen sein. Die Reihenfolge der Alternativen spielt keine Rolle.

Select-Mechanismus:

1: *if* es eine offene `accept`-Alternative gibt *and* ein Transaktionsaufruf dafür vorliegt *and* eine mögliche `suchthat`-Bedingung erfüllt ist, *then* nimm diese Alternative.

2: *else if* es eine offene `immediate`-Alternative gibt, *then* nimm diese.

3: *else if* es keine offenen `delay`- oder `terminate`-Alternativen gibt, *then* warte auf Transaktionsaufrufe für eine der offenen `accept`-Alternativen.

4: *else if* es eine offene `delay`-Alternative (z.B. `delay d`), aber keine offene `terminate`-Alternative gibt, *then* wenn eine akzeptable Transaktion innerhalb von `d` Sekunden kommt, nimm die entsprechende `accept`-Alternative, sonst nimm `delay`-Alternative. (Bei mehereren `delay`-Alternativen nimm die mit minimalem `d`.)

5: *else if* es eine offene `terminate`-Alternative, aber keine offene `delay`-Alternative gibt, *then* warte auf das erste der folgenden Ereignisse:

- **5a:** akzeptable Transaktion. Nimm dann entsprechende `accept`-Alternative.
- **5b:** Das Programm gerät in einen Zustand, wo alle Prozesse entweder *completed* sind oder an einem Select-Statement mit offener `terminate`-Alternative warten. In diesem Fall beende das ganze Programm.

6: *else* (es gibt offene `delay`- und offene `terminate`-Alternativen.) sind zwei Implementierungsstrategien möglich:

- **6a:** ignoriere offenes `terminate` und verfahre wie unter **4**.
- **6b:** Warte auf das erste der folgenden Ereignisse:
  - **6b1:** akzeptable Transaktion. Nimm dann entsprechende `accept`-Alternative.
  - **6b2:** Es passiert **5b**. Verfahre dann wie unter **5b**.
  - **6b3:** Zeitlimit läuft aus. Nimm `delay`-Alternative.

2.6 Channels

Kommunikationskanäle – ***Channels*** – können unter den meisten Sprachen und Betriebssystemen als Kommunikations-Mittel zwischen Tasks eingesetzt werden, da die meisten Sprachen oder Betriebssyteme Queues anbieten. Auch das Rendezvous basiert letzten Endes auf Channels.

Die Sprache Go bietet Channels als schlechthin **die** (einzige) Synchronisations-Möglichkeit an, und die Mechanismen sind besonders bequem und expressiv.

Das Ada-Rendezvous basiert ja auf dem Konzept des Austauschs von Botschaften (Hoare[8]). In Go kann dieses Konzept mit Channels wesentlich "leichtgewichtiger" und klarer realisiert werden.

Die Zugriffe (Schreiben, Lesen) auf Channels sind immer **thread-sicher**. Es kann nicht zu Race-Conditions kommen.

In Channel Schreiben: `x ->kanal`
Aus Channel Lesen: `x <-kanal`

Ist ein Channel voll, muss das Schreiben warten. Ist ein Channel leer muss das Lesen warten.

Entscheidend ist `select`-Mechanismus, der als "Mehrweg-Nebenläufigkeits-Schalter" (***multi way concurrent control switch)*** verwendet werden kann.

```
select {
   case y ->kanal0: // mach was
   case <-kanal1: // mach was
   case x <- kanal2: // mach was mit x
   case <-time.After(3.5*time.Second): // mach was
   default: // mach was
}
```

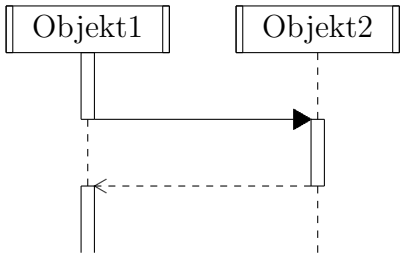
Die Programm-Ausführung wartet am `select` bis einer der Fälle zutrifft. Wenn der Fall `default` dabei ist, trifft dieser immer zu und es wird bei `select` nicht gewartet. Allerdings haben alle anderen Fälle Priorität, d.h. wenn ein `case` zutrifft, also ein Channel beschreibbar oder lesbar oder eine Zeitdauer abgelaufen ist, dann wird dieser Fall ausgeführt.

`case <-kanal`-Alternativen können geschlossen werden, indem die Channel-Variable `kanal` auf `nil` gesetzt wird.

2.7 Message Passing

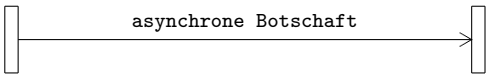
Das Rendezvous in Ada oder die Transaction in Concurrent C beruhen auf einem ***Message Passing*** Konzept. Daher werden bei dieser Gelegenheit im Folgenden die beiden Message Passing Konzepte **Synchrones** und **Asynchrones** Message Passing (sync-MP und async-MP) diskutiert.

Synchrone (blocking) Message Passing :



Objekt 1 schickt eine Botschaft an Objekt 2 und wartet auf eine Antwort.

Asynchrone (non-blocking) Message Passing :



Objekt 1 schickt eine Botschaft an Objekt 2 und wartet **nicht** auf eine Antwort, sondern macht sofort weiter. Eine eventuelle Antwort wäre dann wieder eine Asynchrone Botschaft von Objekt 2 an Objekt 1.

Synchrone (blocking) Message Passing : (Vorteile, Merkmale)

- Bidirektionaler Botschaftenaustausch.
- Verständlicher als async-MP. Async-MP verhält sich oft tückisch wegen unübersichtlicher Zeitabhängigkeiten. Sync-MP verhält sich dagegen wie ein Funktionsaufruf.
- Keine implementationsabhängige Semantik, d.h. synchron ist immer synchron, während asynchron auch manchmal synchron wird, wenn etwa die Botschaften-Queue voll ist.
- Fehlermeldungen sind ganz natürlich.
- Timeouts sind leicht implementierbar.
- Benutzbar als Synchronisationsmechanismus. Bei async-MP ist Polling nötig.
- Programmierfehler sind leichter zu finden. Bei async-MP treten Fehler häufig nur unter Stress auf.

Asynchrone (non-blocking) Message Passing **oder** *Message Queue* : (Vorteile, Merkmale)

- Maximiert Nebenläufigkeit (Concurrency).
- Botschaftenkanal ***(Message Queue)*** mit der Möglichkeit der Auswahl von Botschaften zwecks Optimierung. Oft ist es möglich, den Botschaften Prioritäten ("NORMAL" und "URGENT") zuzuweisen und damit das einfache FIFO-Verhalten zu unterlaufen.
- Expressiveness: z.B. Server kann erst andere Klienten bedienen, wenn für einen Klienten nicht alle benötigten Resourcen zur Verfügung stehen. Bei sync-MP gibt es die sogenannte "early reply"-Technik (Wäsche-Bon, laundry ticket).
- Das synchrone Protokoll kann leicht simuliert werden. Simulation von async-MP mit sync-MP ist umständlicher: man benötigt einen Buffer-Prozess (für die Queue).

- Eine Message Queue transportiert individuelle Botschaften. Der Begriff ***Pipe*** steht für einen Zeichenstrom zwischen zwei Tasks. Eine Pipe wird wie ein I/O-Device behandelt und stellt `read/write`-Methoden zur Verfügung. Für Pipes ist eine Select-Anweisung realisierbar. D.h. eine Task kann alternativ auf eine Menge von I/O-Devices warten.

Ungefähre Leistungsdaten:

Relative Zeiten	Uni-prozessor	Multi-prozessor
Synchron	1	100
Asynchron	10	30
Sim.Async	2	110

2.8 Verteilter gegenseitiger Ausschluss

Man unterscheidet drei Algorithmen:

- **Zentrale Verwaltung:** Ein Knoten übernimmt die Verwaltung eines Semaphors. Er heißt **Koordinator**. Bei ihm müssen alle Tasks, die einen kritischen Bereich betreten wollen, anfragen, ob das möglich ist.

  Konzeptionell sind zentrale Konzepte oder zentrale Koordination bei verteilten Systemen verpönt. Aber obwohl die nächsten beiden Konzepte als nicht "zentral" gelten, kommen sie ohne eine gewisse zentrale Koordination mindestens in der Initial-Phase nicht aus.

- **Token-Ring:** Bei der Initialisierung bekommt eine Task das Token, d.h. die Erlaubnis, einen kritischen Bereich zu betreten. Außerdem muss jede Task die im Token-Ring nachfolgende Task kennen.

  Wenn eine Task das Token hat, dann kann sie entweder den kritischen Bereich betreten oder sie muss das Token an die folgende Task weitergeben.

- **Eintrittskarte:** Hierbei sendet eine Task, die einen kritischen Bereich betreten will, an alle Konkurrenten einen Betretungswunsch und wartet dann, bis alle Konkurrenten diesen Wunsch erlauben. Dann kann sie den kritischen Bereich betreten. Nach Verlassen des kritischen Bereichs wird an alle Tasks, die ebenfalls warten, eine Erlaubnis geschickt. Algorithmus:

  1. Betretungswunsch an alle Konkurrenten senden.
  2. Warten bis alle Konkurrenten die Erlaubnis erteilen. (Wenn dies mit einer Task realisiert wird, dann heißt diese Task **Reply-Task**.) Währenddessen muss auf Betretungswünsche anderer Prozesse reagiert werden. Das wird mit einer weiteren Task **(Request-Task)** realisiert, die diese Prozesse als "verschoben" registriert, aber die sonst (d.h. wenn man selbst nicht wartet) diesen Prozessen eine Erlaubnis erteilt. D.h. die Request-Task muss immer laufen.
  3. Wenn alle anderen interessierten Tasks entweder eine Erlaubnis erteilt haben oder "verschoben" sind, dann kann der kritische Bereich betreten werden.
  4. Kritischen Bereich verlassen und dann alle "verschobenen" Konkurrenten informieren, d.h. eine Erlaubnis schicken.

Das Problem hier ist, dass die Intertask-Kommunikation sehr hoch ist. Außerdem muss zentral geregelt werden, wer zu den Konkurrenten gehört.

2.9 Intertask-Kommunikation

Zur Intertask- oder Interprozess-Kommunikation werden ***Message Queues*** und ***Pipes*** eingesetzt.

Message Queue: Das ist ein Ringpuffer, dessen Speicherplätze durch ganze Objekte, nämlich Botschaften belegt werden. Es werden typischerweise die Methoden

```
send (:Message)
```
und
```
receive () :Message
```

bzw. nach modernem Standard

```
enqueue (:Message)
```
und
```
dequeue () :Message
```

zur Verfügung gestellt.

Pipe: Das ist ein Ringpuffer, dessen Speicherplätze Bytes oder Chars sind. Eine Pipe repräsentiert einen Datenstrom. Es werden typischerweise die Methoden

```
write (anzBytes, :byte[])
```
und
```
read (anzBytes, :byte[]) :int
```

zur Verfügung gestellt. Die `read`-Methode gibt die tatsächlich gelesenen Bytes zurück. D.h. es können durchaus weniger als `anzBytes` gelesen werden. Die Methode blockiert nur, wenn der Puffer ganz leer ist.

Abhängig vom System werden zusätzliche Methoden oder unterschiedliches Verhalten angeboten.

2.10 Diskussion

Semaphore und Monitore sind elementare Konzepte. Semaphore sind weiter verbreitet.

Monitore sind strukturiert und damit wesentlich sicherer. Semaphore sind das "Goto" der nebenläufigen Programmierung. Entfernte Prozeduren können Semaphore benützen. Das widerspricht dem Konzept der Strukturierung.

Dijkstra: "Was ist Software-Engineering? How to program if you can't."

Kapitel 3

Petri-Netze

Obwohl wir Petri-Netze als nicht besonders nützlich für die Entwicklung von Echtzeitsystemen erachten, seien sie hier der Vollständigkeit halber vorgestellt (und auch weil sie ein nettes Spielzeug sind).

Carl Adam Petri hat 1962 in einer Veröffentlichung die dann nach ihm benannten Petri-Netze vorgestellt. Seither wurden verschiedene Varianten entwickelt, von denen hier eine vorgestellt wird[19].

Petri-Netze sind ein Mittel zur Modellierung paralleler Abläufe.

3.1 Netzregeln und Notation

Ein Petri-Netz besteht aus

- **Stellen** (Kreise, Ovale) und
- **Transitionen** (Rechtecke) und
- **Pfeilen**, die
  - entweder von einer Stelle zu einer Transition gehen
  - oder von einer Transition zu einer Stelle gehen.
- Stellen können **Marken** (fette Punkte) besitzen.
- Zustandsänderungen erfolgen durch das **Schalten** von Transitionen. Eine Transition t kann schalten, wenn sich in allen Stellen, von denen ein Pfeil nach t geht, mindestens eine Marke befindet. Durch das Schalten wird aus all diesen Stellen eine Marke entfernt und eine Marke den Stellen hinzugefügt, zu denen ein Pfeil von t aus geht. (Es kann immer nur eine Transition schalten.)
- Zwei Transitionen stehen im **Konflikt**, wenn sie mindestens eine eingehende Stelle teilen und beide schalten können.

- Es kann Transitionen geben, die keine **ein**gehenden Pfeile haben. Diese Transitionen können immer schalten. Sie sind sogenannte Markengeneratoren.

- Es kann Transitionen geben, die keine **aus**gehenden Pfeile haben. Wenn diese Transitionen schalten, dann vernichten sie Marken.

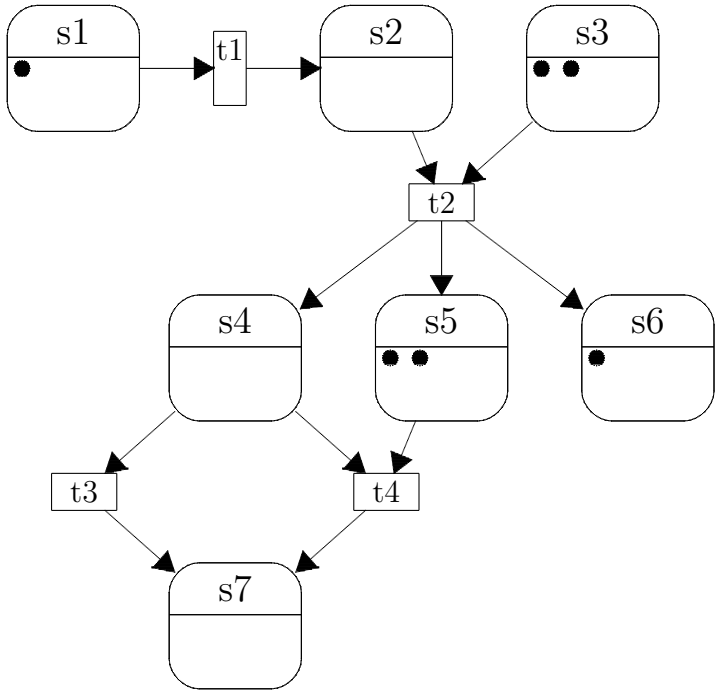
- Die Reihenfolge, in der Transitionen, die schalten können, schalten, ist nicht festgelegt. Damit ist ein Petri-Netz nicht deterministisch. D.h. es gibt eben verschiedene Szenarien.

  Konflikte müssen irgendwie behandelt werden. Z.B. kann man bei Konflikten Prioritäten vergeben.

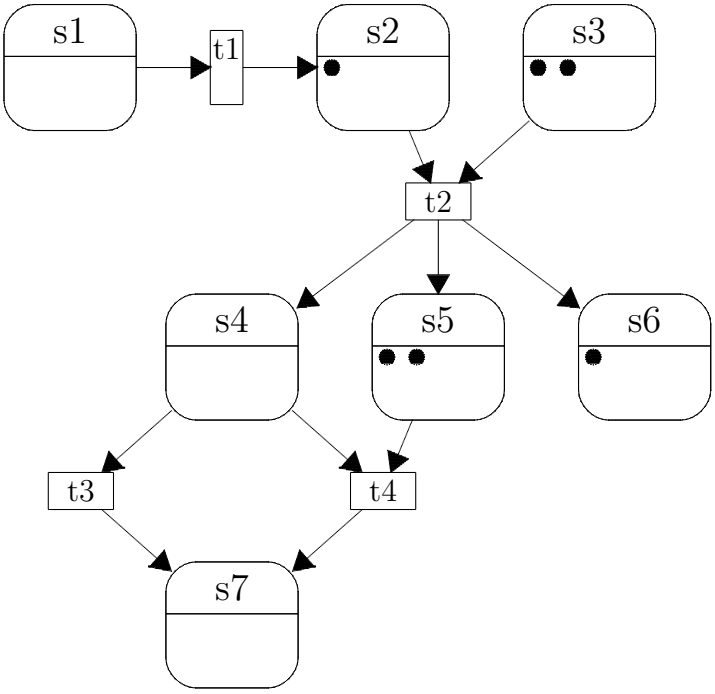
- Variante: Die Kapazität (maximale Anzahl von Marken) einer Stelle kann beschränkt sein. Eine Transition, die solch einer Stelle eine Marke zuführen würde, kann nicht schalten.

- Variante: Ein Pfeil kann eventuell mehr als eine Marke transportieren.

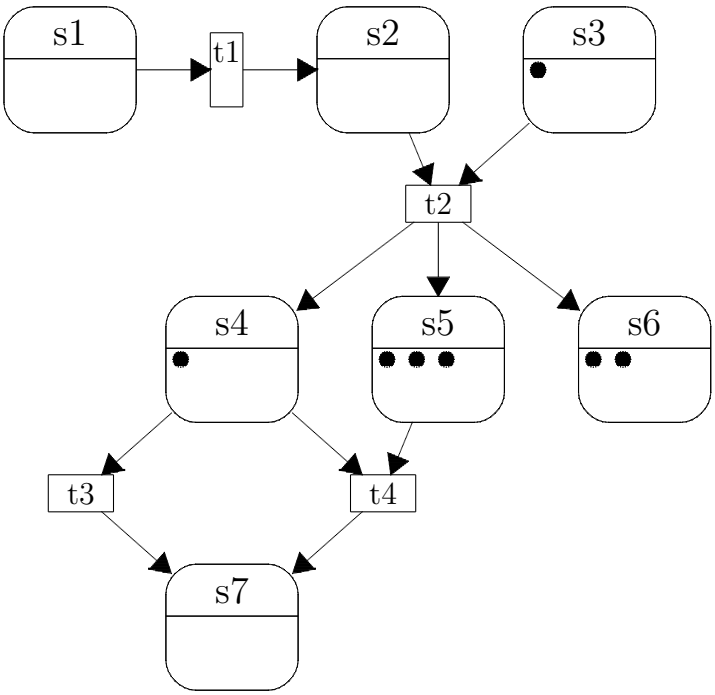
Als Beispiel betrachten wir folgendes Petri-Netz:



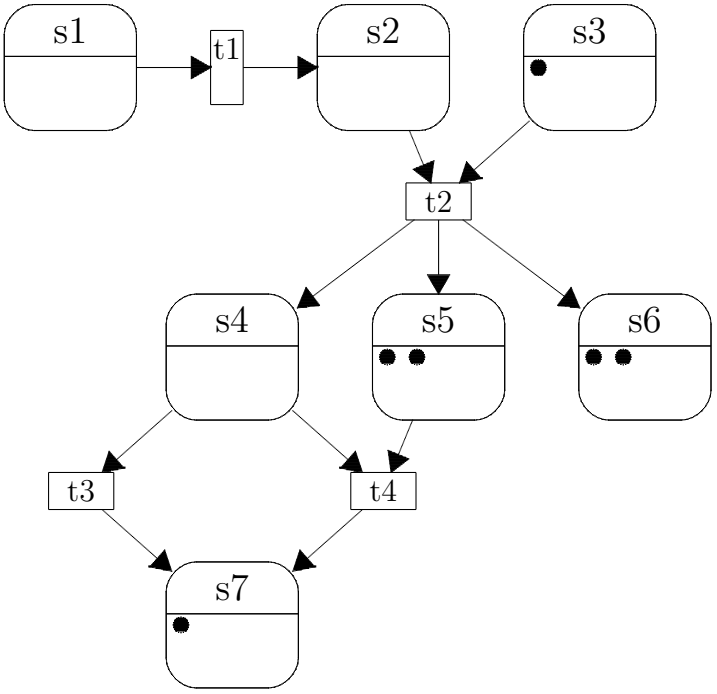
Als einzige Transition kann t_1 schalten. Das ergibt:



Jetzt kann Transition t_2 schalten. Das ergibt:



Jetzt können t_3 und t_4 schalten. Es kommt zum Konflikt. Wir entscheiden uns für t_4. Das ergibt:



Wir bilden Code folgendermaßen auf Stellen und Transitionen ab:

- Stellen sind Punkte vor Code-Stücken oder Prozedur-Aufrufen, die alle mit Marken belegt sein müssen, damit die Code-Stücke bzw. Prozeduren ausgeführt werden können. Wenn z.B. vor Betreten eines Code-Stücks ein oder mehrere Bedingungen erfüllt sein müssen, auf die gewartet werden muss, dann lässt sich das durch ein oder mehrere Stellen realisieren.

  Stellen sind Wartebedingungen. Es wird an der Stelle gewartet, bis die Stelle mindestens eine Marke hat. (Eventuell muss auf mehr als eine Stelle gewartet werden.)

- Transitionen sind Code-Stücke oder Prozeduren. Es werden Pfeile von den vor der Transition liegenden Stellen zu der betreffenden Transition gezogen.

- Wenn nach einer Transition eine Wartebedingung aufgehoben wird, dann wird ein Pfeil von dieser Transition zur entsprechenden Stelle gezogen.

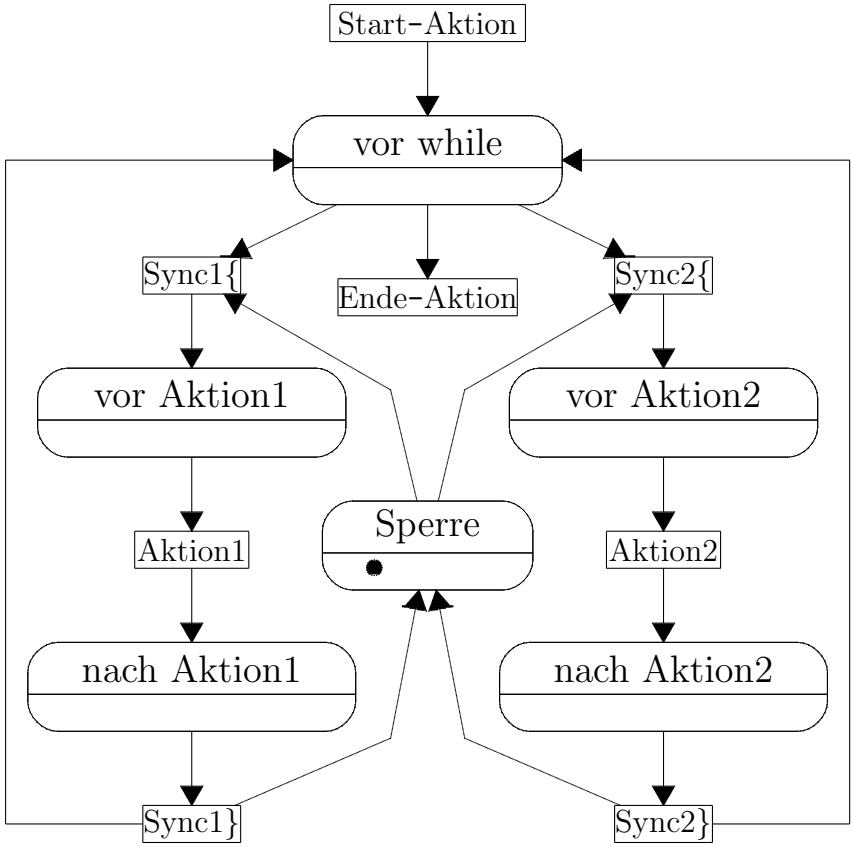
3.2 Beispiel Java-Monitor

Als Beispiel zeigen wir die Realisierung des gegenseitigen Ausschlusses mit dem Monitor-Konzept von Java. Der Java-Code hat folgende Form:

```
public class   Task extends Thread
   {
   private static Object   sync = new Object ();
      // Wird zum synchronisieren verwendet.
   private static java.util.Random  zufall = new java.util.Random ();
   public void run ()
```

```
    {
    System.out.println ("Start-Aktion");         // Transition: "Start-Aktion"
    boolean weiter = true;
    while (weiter)                               // Stelle: "vor while"
       {
       switch (zufall.nextInt (2))
          {
          case 0:
             synchronized (sync)                 // Transition: "Sync0{"
                {                                // Stelle: "vor Aktion0"
                System.out.println ("Aktion0");  // Transition: "Aktion0"
                weiter = false;                  // Stelle: "nach Aktion0"
                }                                // Transition: "Sync0}"
          case 1:
             synchronized (sync)                 // Transition: "Sync1{"
                {                                // Stelle: "vor Aktion1"
                System.out.println ("Aktion1");  // Transition: "Aktion1"
                                                 // Stelle: "nach Aktion1"
                }                                // Transition: "Sync1}"
          case 2:
             synchronized (sync)                 // Transition: "Sync2{"
                {                                // Stelle: "vor Aktion2"
                System.out.println ("Aktion2");  // Transition: "Aktion2"
                                                 // Stelle: "nach Aktion2"
                }                                // Transition: "Sync2}"
          }
       }
    System.out.println ("Ende-Aktion");          // Transition: "Ende-Aktion"
    }
 public static void   main (String[] arg)
    {
    for (int i = 0; i < 10; i++) new Task ().start ();
    }
 }
```

Dieses Programm kann mit folgendem Petri-Netz modelliert werden: (Der ”0”-Zweig wurde der Übersichtlichkeit halber weggelassen, sieht aber wie die anderen Zweige aus.)



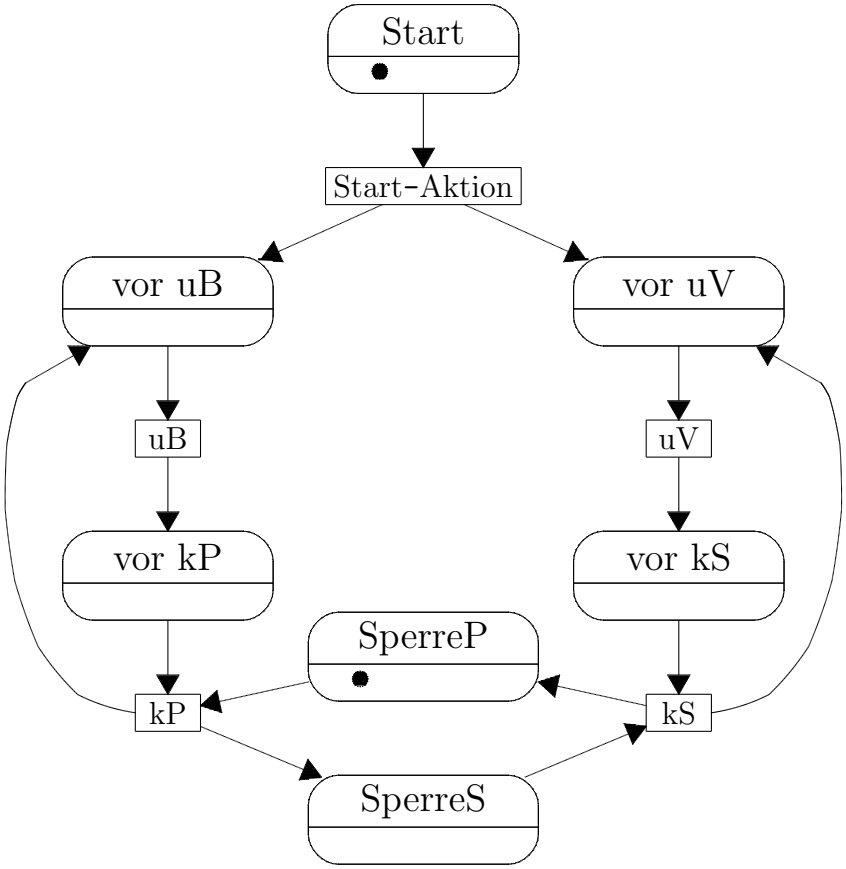
Die einzige Transition, die in diesem Zustand schalten kann, ist "Start-Aktion". Sie kann immer schalten. Dies entspricht einer Generierung von Threads. Sie liefert dabei Marken in die Stelle "vor while". Dadurch kann dann die Transition "Sync{" schalten, wenn die Stelle "Sperre" mit einer Marke belegt ist.

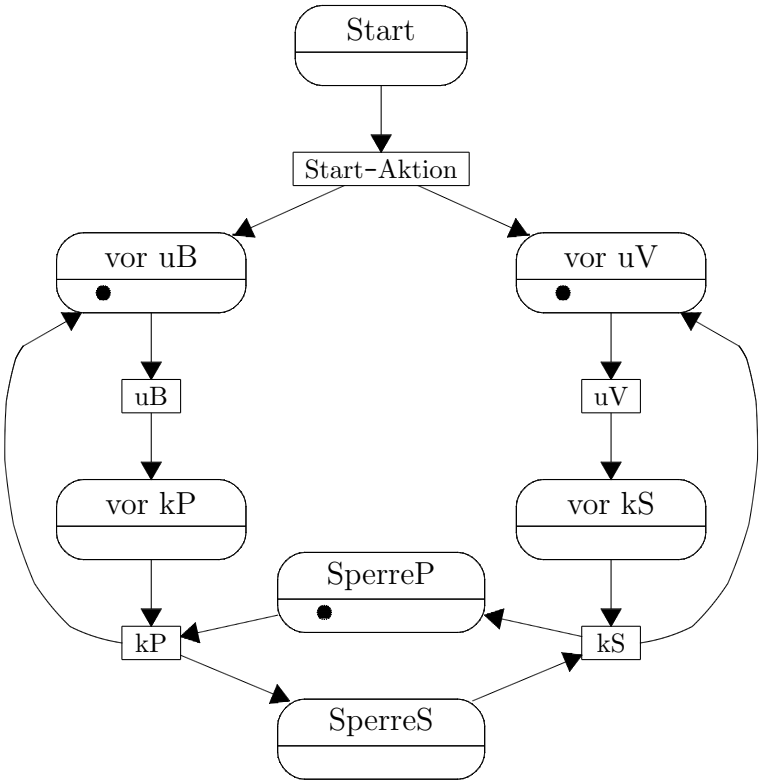
Dass die Sperre mit einer Marke initialisiert ist, bedeutet, dass die eine Task den geschützten Bereich betreten kann (Schalten von Transition "SyncX"). Dabei verliert die Sperre ihre Marke. Die nächste Task muss warten, bis eine Transition "SyncX" geschaltet hat und dadurch der Sperre wieder eine Marke gegeben hat.

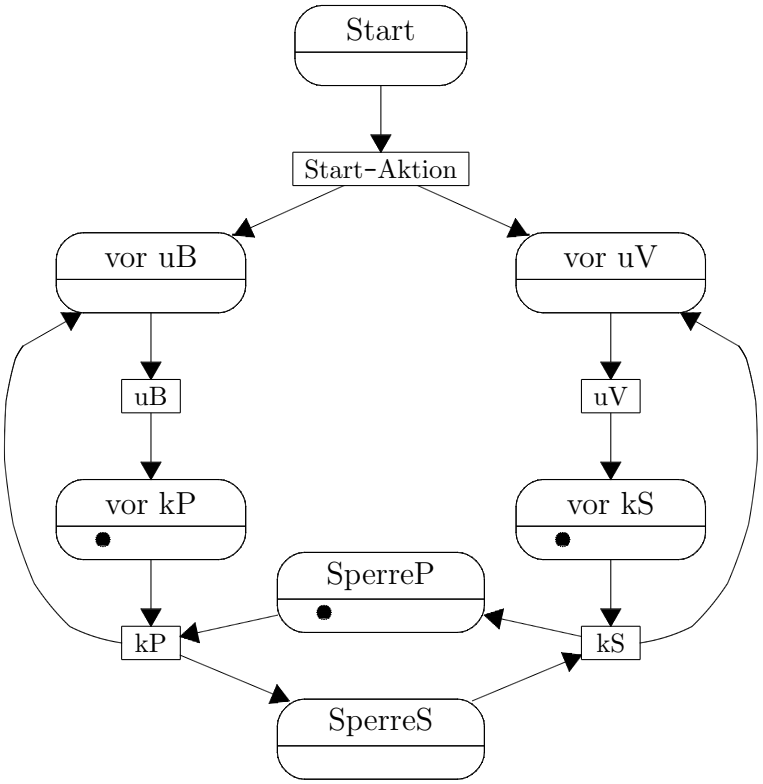
Es ist sehr umständlich, diese Petri-Netze von Hand durchzuschalten. Ein Simulator ist hier unerlässlich.

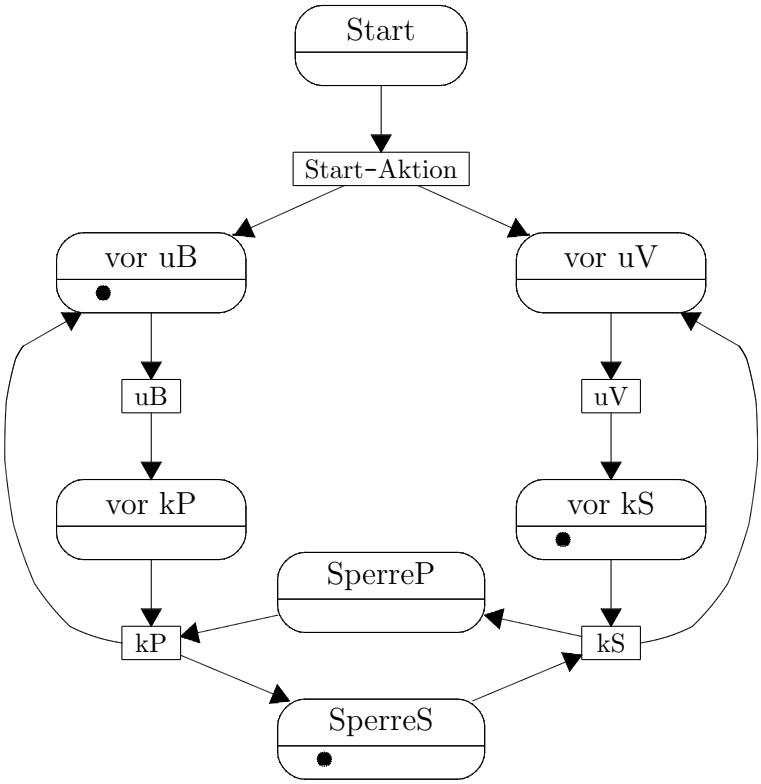
3.3 Beispiel Schweissroboter

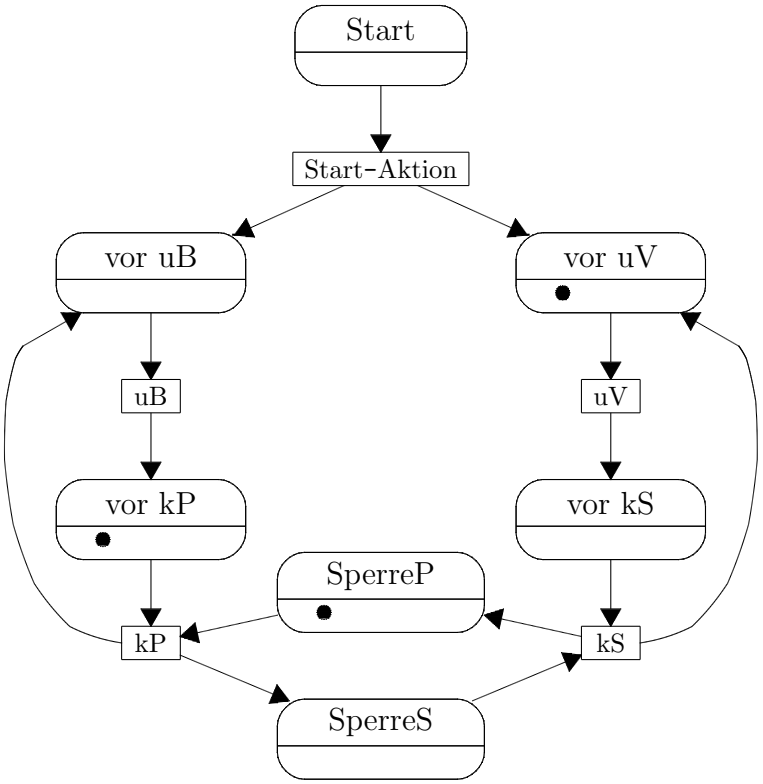
Als weiteres Beispiel stellen wir den Schweiss-Roboter aus dem Kapitel Synchronisationsmechanismen vor:


Start
Start-Aktion
vor uB
vor uV
uB
uV
vor kP
vor kS
SperreP
kP
kS
SperreS


Start
Start-Aktion
vor uB
vor uV
uB
uV
vor kP
vor kS
SperreP
kP
kS
SperreS


Start
Start-Aktion
vor uB
vor uV
uB
uV
vor kP
vor kS
SperreP
kP
kS
SperreS


Start
Start-Aktion
vor uB
vor uV
uB
uV
vor kP
vor kS
SperreP
kP
kS
SperreS



3.4 Übung

Erstellen Sie Regeln für die Implementation mit Semaphoren.

Kapitel 4

Schritthaltende Verarbeitung

Schritthaltende Verarbeitung, Realzeit-Betrieb, Echtzeit-Betrieb oder **Realtime-Betrieb** bedeutet, dass das einen **technischen Prozess (P)** steuernde **Automatisierungssystem (AS)** auf die Anforderungen des Prozesses innerhalb vorgegebener Zeiten reagiert. (Das gesamte System bezeichnen wir als ein **Prozess-Automatisierungs-System (PAS)**.)

Zusätzlich zur **Richtigkeit** eines Ergebnisses spielt bei der Realzeit noch die **Rechtzeitigkeit** des Ergebnisses eine wichtige Rolle.

4.1 Prozessbedingte Zeiten

4.1.1 Programm- oder Prozessaktivierung

Ein Programm des AS kann

- **zyklisch** (Alarme des technischen Prozesses und die Reaktion des Automatisierungssystems erfolgen in zeitlich konstantem Abstand.) oder
- **nicht zyklisch** (Alarme und Reaktion erfolgen zu willkürlichen Zeitpunkten.)

arbeiten. In jedem Fall kann das Programm des AS **ereignisgesteuert** durch den realen Prozess und/oder durch die **eigene festgelegte Programmausführung** aktiviert werden.

4.1.2 Spezifikation über Zeitdauern

Die prozessbedingte **maximal zulässige Reaktionszeit** oder **spezifizierte Reaktionszeit** bezeichnen wir mit t_z.

Die Ereignisse des zu steuernden Prozesses P *(events,* Alarme, Anforderungen), auf die das AS zu reagieren hat, treten i.A. zyklisch im zeitlichen Abstand t_p auf. t_p heißt **Prozesszeit (PZ)** oder im Zusammenhang mit Direct Digital Control (DDC) Abtastperiode oder Taktzeit. Die

Prozesszeit ist nicht zu verwechseln mit der **Prozess-Zeitkonstante** oder genauer **Prozess-Regelstrecken-Zeitkonstante (PRZ)**, die durch die Natur des Prozesses (δ-Sprungverhalten, Einschwingverhalten usw.) vorgegeben ist und nicht durch das AS beeinflusst werden kann. Die Prozesszeit ist eine Design-Größe in Abhängigkeit von der Prozess-Zeitkonstanten. Bei DDC ist es z.B. sinnvoll, die Prozesszeit um eine oder mehrere Größenordnungen kleiner zu wählen als die Prozess-Zeitkonstante.

Die **Prozesszeit** t_p ist entweder eine Größe, die der Prozessbetreiber unter Berücksichtigung der physikalisch-chemischen Eigenschaften des Prozesses festlegt. Dann sind das häufig konstante Zeitintervalle. Oder die Prozesszeit hängt von mehr oder weniger zufälligen Ereignissen oder Prozesszuständen ab. In dem Fall können die Prozesszeiten sehr unterschiedlich sein. Damit aber überhaupt ein Systementwurf möglich ist, muss auf jeden Fall für die Prozesszeit eine untere Grenze $t_{p_{min}}$ angebbar sein:

$$t_{p_{min}} \leq t_p$$

Alle folgenden Überlegungen bleiben auch für den Fall der zufällig verteilten Prozesszeiten richtig, wenn man als Prozesszeit t_p den Wert $t_{p_{min}}$ verwendet *(Worst Case Design)*.

Die **maximal zulässige Reaktionszeit** t_z muss so spezifiziert werden, dass

$$t_z \leq t_p$$

gilt. Wenn t_z nicht explizit angegeben wird, wird $t_z = t_p$ angenommen.

Damit Echtzeit-Betrieb gewährleistet ist, muss die **Reaktionszeit** t_r des Prozessautomatisierungssystems kleiner als die spezifizierte Reaktionszeit t_z sein, d.h. die **Echtzeitbedingung (EZB)** ***(real-time condition, RTC)***

$$t_r \leq t_z$$

muss gelten.

Durch welche Maßnahmen kann man nun die Erfüllung der Echtzeitbedingung erreichen?

Zunächst ist zu prüfen, ob die Erfassungs-, Rechen- und Ausgabekapazitäten überhaupt ausreichen, um die geforderten Aufgaben in den spezifizierten Zeitintervallen durchführen zu können. Darüberhinaus gibt es für den Programmierer eines AS nur noch zwei Entwurfs-Möglichkeiten: **Tasking** und **Vergabe von Prioritäten**.

Tasking bedeutet Aufteilung der ganzen Steuerungsaufgabe in Teilaufgaben, die (quasi-)parallel bearbeitet werden. Ein technischer Prozess kann i.A. so modelliert werden, dass er aus mehreren Teilprozessen i mit den zugehörigen Prozesszeiten t_{p_i} und maximal zulässigen Reaktionszeiten t_{z_i} besteht. Für alle Teilprozesse muss die Echtzeitbedingung erfüllt sein:

$$t_{r_i} \leq t_{z_i} \quad (\leq t_{p_i}) \qquad \text{für alle Teilprozesse} \quad i$$

- Jedem Teilprozess eine ihn steuernde Task zuzuordnen, ist ein bewährtes **Design-Prinzip** des AS.

4.1.3 Spezifikation über Zeitpunkte

Wir haben die Echtzeitbedingungen über Intervalle definiert. Man kann das auch über Zeitpunkte tun:

- R: frühester Startzeitpunkt *(release time)*
- D: spätester Endzeitpunkt *(deadline)*
- S: tatsächlicher Startzeitpunkt *(start time)*
- E: tatsächlicher Endzeitpunkt *(completion time)*

Es wird dann häufig noch die Laufzeit *(computation time)* spezifiziert:

$$C = E - S$$

Übung 1: Stellen Sie den Zusammenhang mit den Intervalldefinitionen her.

4.1.4 Übung 2: Motorsteuerung

Diskutieren Sie die Verhältnisse bei einer Motorsteuerung.

4.2 Task

Der Begriff **Task** bezieht sich auf die *Ausführung* eines Programms oder Programmstücks. Verschiedene Tasks können sich auf dasselbe Programmstück beziehen.

Anstatt Task werden auch die Begriffe **Prozess** (hier nicht als "zu steuernder Prozess", sondern als "Rechenprozess") oder **Thread** verwendet. Für uns ist Task der übergeordnete Begriff.

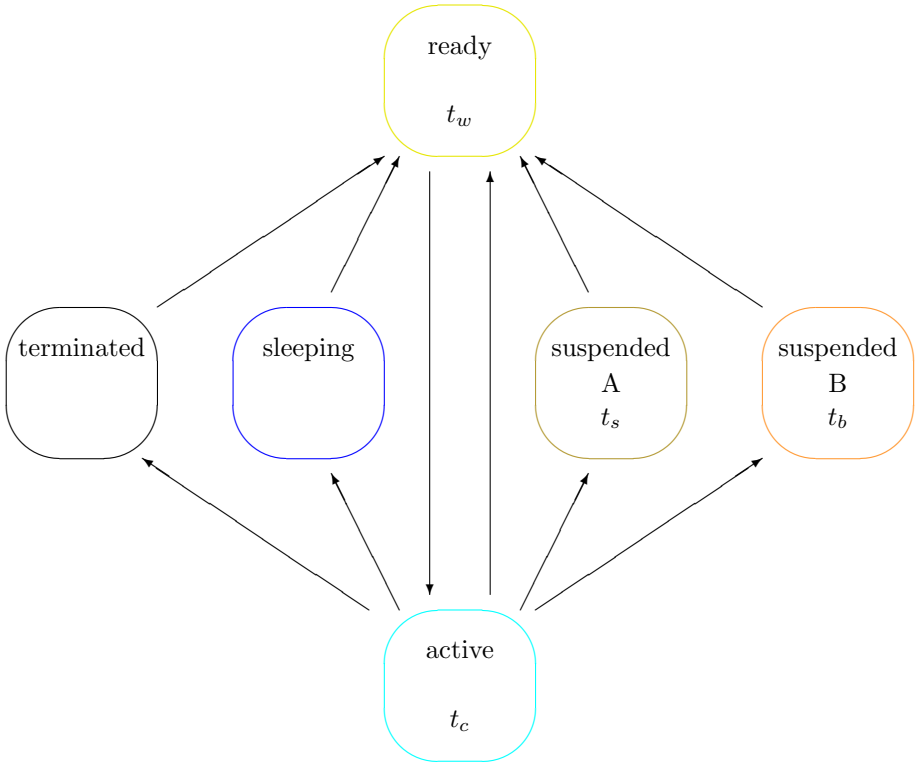
Ein (Rechen-)Prozess hat einen eigenen Adressraum. Diese Prozesse können nur über Betriebssystem-Mittel kommunizieren. Threads dagegen sind Leichtgewichtsprozesse ohne eigenen Adressraum. Sie können auch über globale Variablen kommunizieren.

Ein **Programm** ist eine statische Aufschreibung einer Folge von Anweisungen, während die Task als Programm*ablauf* zu verstehen ist.

Objekt-orientiert ist eine Task ein Objekt einer Klasse, die eine Methode hat, deren Code als Prozess oder Thread gestartet werden kann.

Das Verhalten einer Task wird für die hier nötigen zeitlichen Betrachtungen mit sechs Zuständen beschrieben.

Zustandsgraph einer Task:



Die bei den Zuständen angegebenen Zeiten sind die jeweiligen (kumulierten) Gesamtzeiten, die eine Task in einem Zustand während der Reaktionszeit t_r (näheres siehe unten) verbringt.

Die Bezeichnungsweisen sind sehr uneinheitlich. Daher sind hier einige der häufiger verwendeten Bezeichnungen alternativ angegeben.

Zustände:

terminated, beendet, neu: Die Task ist wurde noch nicht gestartet. Durch das Betriebssystem kann sie gestartet werden. Dann kommt sie in den Zustand *ready*. Wenn eine Task fertig ist, dann betritt sie ebenfalls diesen Zustand als *terminated* und kann dann eventuell neu gestartet werden.

sleeping, dormant, ruhend: Die Task ruht solange, bis sie auf Grund eines Ereignisses (Alarm) geweckt wird. Dann kommt sie in den Zustand *ready*.

ready, lauffähig, bereit, rechenbereit: Alle Voraussetzungen dafür, dass die Task laufen (d.h. die CPU benutzen) kann, sind erfüllt. Es ist nur noch eine Frage der Priorität, ob die Task läuft. Wenn sie die höchste Priorität bekommt, dann geht die Task in den Zustand *active* über.

active, running, busy, laufend, rechnend: Die Task läuft, d.h. wird vom Prozessor bearbeitet. Bei einem Einprozessorsystem kann nur eine Task in diesem Zustand sein. (Alle anderen Zustände können von mehreren Tasks eingenommen werden.)

Der Zustand *active* wird verlassen,

- wenn die Task die höchste Priorität verloren hat (zurück in Zustand *ready)*,
- oder wenn die Task an ihr natürliches Programmende gelaufen ist (dann in Zustand *sleeping)*,
- oder wenn eine Warte-Bedingung eingetreten ist (dann in Zustand *suspended* A oder B).

suspended, pended, blockiert, wartend: Die Task kann nicht laufen, da für sie eine Wartebedingung gilt, d.h. sie muss auf ein Ereignis warten, z.B. darauf, dass ein Betriebsmittel frei wird, oder auf I/O, auf einen Interrupt oder darauf, dass ein Zeitintervall abgelaufen ist. Nach Eintreten des Ereignisses geht die Task in den Zustand *ready*.

Abweichend von sonst üblichen Darstellungen spalten wir diesen Zustand in zwei Zustände auf:

- `A`: Die Task wartet auf ein Ereignis, das nur von ihrem zu steuernden Teilprozess abhängig ist (z.B. ein vom Teilprozess benutzter A/D-Wandler).
- `B`: Die Task wartet auf die Zuteilung eines – von anderen Tasks benutzten – Betriebsmittels (z.B. gemeinsam benutzte I/O-Karten oder Speicherplätze) oder kritischen Bereichs.

Randbemerkung: Wenn die CPU gar keine Task zu bearbeiten hat, dann ist sie im sogenannten ***idle state***. (Das ist aber ein Zustand der CPU, nicht der einer Task.)

4.3 Reaktionszeit

Um Aussagen über die Echtzeitbedingung machen zu können, müssen wir uns genauer mit der Reaktionszeit t_r des AS beschäftigen. Innerhalb der Reaktionszeit muss das AS typischerweise auf eine Anforderung – ausgelöst durch ein Prozessereignis (spezieller Prozesszustand, Interrupt oder Uhrzeit) – reagieren mit

- Analyse der Anforderung
- Messen mit einem Messinstrument
- Erfassen des Messwerts über eine I/O-Schnittstelle
- Verarbeiten des Messwerts und Berechnung einer Ausgabegröße
- Ausgeben der Ausgabegröße über eine I/O-Schnittstelle
- Steuern mittels eines Aktors

Zur Abschätzung von t_r muss man grundsätzlich den ***Worst-Case*** betrachten **(WCET, *worst-case execution time*)**.

Die Zeit, die für diese Tätigkeiten – **ohne** Berücksichtigung anderer Tasks – benötigt wird, nennen wir **Verarbeitungszeit** t_v. Sie setzt sich zusammen aus der **Prozessorzeit** t_c (Zeit im Zustand *active*) und der Zeit im Zustand *suspended (A)* t_s, weil die Task etwa auf einen A/D-Wandler, ein Messgerät oder ein anderes Ereignis **ihres zu steuernden** Prozesses während der Reaktionszeit warten muss.

$$t_v = t_c + t_s$$

Die Verarbeitungszeit kann man für eine isolierte Task relativ gut abschätzen oder messen. Die Reaktionszeit t_r dagegen ist die Verarbeitungszeit t_v plus die Zeit t_w, die die Task im Zustand *ready* auf die Prozessorzuteilung warten muss, plus die Zeit im Zustand *suspended (B)* t_b, wo die Task auf ein gemeinsam benutztes Betriebsmittel warten muss.

$$t_r = t_v + t_w + t_b$$

Die Zeit t_w ist abhängig von der Anzahl und den Prioritäten anderer Tasks und kann daher nur sehr schwer für eine einzelne Task bestimmt werden. Wir geben hier eine Abschätzung für die Task i:

$$t_{w_i} \leq \sum_{\text{alle } j \neq i} t_{c_j} \cdot \left\lceil \frac{t_{z_i}}{t_{p_j}} \right\rceil \qquad \text{Prio}(j)(\text{logisch} \geq)\text{Prio}(i)$$

Hierbei ist über alle Tasks j zu summieren, die höhere oder gleiche Priorität haben als die Task i. Die Task i wird in der Summe nicht berücksichtigt. Der gerundete Term ($\lceil x \rceil$ = ceil(x) = nächst höhere ganze Zahl) gibt an, wie oft der Prozess j innerhalb der maximal zulässigen Reaktionszeit t_{z_i} der Task i seine Prozessorzeit t_{c_j} benötigt. Für ein Worst-Case-Design sollte das Gleichheitszeichen genommen werden.

Wie kann man t_{b_i} abschätzen? Unter der Voraussetzung, dass das unten diskutierte Problem der Prioritätsinversion gelöst ist, muss für jedes von der Task i benutzte Betriebsmittel BM für alle Tasks $j \neq i$ die maximale Benutzungsdauer $t_{BM_{max_{j,i}}}$ innerhalb von $\min(t_{z_j}, t_{z_i})$ bestimmt werden.

$$t_{alleBM_{max_{j,i}}} = \sum_{alleBM} t_{BM_{max_{j,i}}}$$

Hierbei wird über alle Betriebsmittel summiert, die sich Task i und Task j während t_{r_i} teilen.

Dann ergibt sich als Abschätzung für t_{b_i}

$$t_{b_i} \leq \sum_{\text{alle } j \neq i} t_{alleBM_{max_{j,i}}} \cdot \left\lceil \frac{t_{z_i}}{t_{p_j}} \right\rceil$$

Hier muss – unabhängig von der Priorität – über alle Tasks summiert werden, wobei natürlich für alle Tasks, die mit der Task i kein Betriebsmittel teilen, der Term $t_{alleBM_{max_{j,i}}}$ Null ist. Für ein Worst-Case-Design sollte das Gleichheitszeichen genommen werden.

Wartezeiten, die während des Gebrauchs eines Betriebsmittels durch die Task i auftreten, wurden schon bei t_{s_i} berücksichtigt.

Bemerkungen:

- Die Bestimmung der Terme $t_{BM_{j,i}}$ ist oft nicht einfach. Als Beispiel betrachten wir zwei Tasks A und B, die über einen Ringpuffer kommunizieren. Der Zugriff auf den Ringpuffer geschieht unter gegenseitigem Ausschluss und dauert $10\mu s$. Task A schreibt alle $t_{z_A} = 100\mu s$ Daten in den Ringpuffer. Task B liest diese Daten aus dem Ringpuffer im Schnitt mit gleicher Geschwindigkeit oder etwas schneller, aber ziemlich unregelmäßig innerhalb ihres Zyklus von $t_{z_B} = 10\,000\mu s$.
  $t_{BM_{max_{B,A}}} = 10\mu s \quad \text{bis} \quad 100\mu s$
  $t_{BM_{max_{A,B}}} = 10\mu s$
  Bei $t_{BM_{max_{B,A}}}$ kann der Wert 100 auftreten, wenn Task B lange nicht gelesen hat, und auf einmal 10 mal oder noch öfter liest. Das würde bedeuten, dass Task A ihre Echtzeitbedingung nicht erfüllen könnte. Allerdings tritt der Wert 100 nicht auf, wenn wir den Zugriff auf den Ringpuffer für Task A priorisieren. Wenn dann Task A schreiben will, muss sie höchstens einen Lesezugriff von $10\mu s$ abwarten. Für die Zeiten t_b ergibt sich:
  $t_{b_A} = 10 \cdot \lceil \frac{100}{10\,000} \rceil = 10\mu s$
  $t_{b_B} = 10 \cdot \lceil \frac{10\,000}{100} \rceil = 1000\mu s$
  Eine Tabelle der folgenden Form mag hilfreich sein:

Task X → wartet $t_{alleBM_{max_{Y,X}}}$ auf Task Y ↓	A	B	C	...
A	0	10 + 50 = 60ms	100 + 40 = 140ms	...
B	10 + 60 = 70ms	0	80ms	...
C	120ms	80ms	0	...
...	...	...	...	0
t_{b_X}	190ms	140ms	220ms	...

- Die vorgestellten Abschätzungen orientieren sich an der ***rate monotonic analysis (RMA)*** bzw. ***rate monotonic scheduling (RMS)***[**?**][**?**].

- Der System-Overhead, d.h. die Prozessorzeit, die die Systemtask(s) benötigen (Taskwechselzeiten), ist hier mit zu berücksichtigen. Das ist im Detail sehr schwer durchzuführen, kann aber eventuell pauschal gemacht werden. Wenn z.B. der Systemoverhead erfahrungsgemäß etwa 10% beträgt, könnte die oben angegebene Summe mit dem Faktor 1.1 multipliziert werden.

  Allgemein ist das schwierig zu berücksichtigen. Innerhalb eines t_c kann es durchaus mehrere Taskwechsel geben. Oft sind die Taskwechselzeiten größer als die netto Rechenzeiten.

  Eine vernünftige Näherung dürfte sein, wenn man anstatt von t_c

  $$t_c + 2 \cdot \text{Taskwechselzeit}$$

  verwendet.

4.4 Relative Gesamtbelastung

Eine notwendige Voraussetzung für das Einhalten der Echtzeitbedingung ist, dass die relative Gesamtbelastung des Prozessors kleiner 1 bzw. kleiner 100% ist.

$$\text{Relative Gesamtbelastung} \quad = \sum_{\text{alle Tasks } i} \frac{t_{c_i}}{t_{p_i}} \qquad \leq 1$$

Wenn die Tasks diese Bedingung erfüllen, dann heißen sie **zeitlich verwaltbar**.

Die relative Gesamtbelastung kann berechnet werden, ohne dass das Tasking klar ist, indem man für jede Aktivität a die benötigte CPU-Zeit t_{c_a} und ihre Häufigkeit, d.h. ihre Prozesszeit t_{p_a} bestimmt, und obige Formel in folgender Formulierung verwendet:

$$\text{Relative Gesamtbelastung} \quad = \sum_{\text{alle Aktivitäten } a} \frac{t_{c_a}}{t_{p_a}} \qquad \leq 1$$

Nach dem Tasking kommen dann wahrscheinlich noch Terme für die Intertask-Kommunikation hinzu.

4.5 Priorität

Betrachten wir als Beispiel zwei Tasks, die jeweils einen realen Prozess steuern sollen, mit folgenden Zeiten:

Prozess	$t_z = t_p$ ms	$t_c = t_v$ ms
A	10.0	3.5
B	2.0	1.0

Wenn der Prozessor ausschließlich **einen** Teilprozess bedienen kann, dann sieht das für die beiden Teilprozesse folgendermaßen aus:

(Bild)

Wenn wir Task A, die Teilprozess A steuert, die höhere Priorität geben, wird die Echtzeitbedingung bei A eingehalten. Bei Teilprozess B kann die Echtzeitbedingung meistens nicht erfüllt werden, wie man an unten dargestelltem Szenario erkennt.

(Bild)

Wenn wir Task B, die Teilprozess B steuert, die höhere Priorität geben, wird die Echtzeitbedingung bei B und auch bei A eingehalten, wie man an unten dargestelltem Szenario erkennt. Allerdings haben wir hier noch vorausgesetzt, dass die höher priore Task die Task mit niedrigerer Priorität unterbrechen kann.

(Bild)

In diesem Beispiel ergibt sich die relative Gesamtbelastung zu $\frac{3.5}{10.0} + \frac{1.0}{2.0} = 0.85$.

Je nach Prioritätsvergabe ergeben sich mit der oben angegebenen Abschätzung für die Wartezeiten t_w folgende Werte:

Prozess	$t_z = t_p$	$t_c = t_v$	t_v	A höhere Priorität		B höhere Priorität	
				t_w	t_r	t_w	t_r
A	10.0	3.5	3.5	0.0	3.5	5.0	8.5
B	2.0	1.0	1.0	3.5	4.5	0.0	1.0

Diese Werte bestätigen das Ergebnis des Szenarios.

Damit Prioritäten sinnvoll vergeben werden können, muss das Betriebssystem ein ***priority based preemptive scheduling*** zur Verfügung stellen. (Eine Task muss auf Grund von Priorität unterbrechbar sein.)

Als Faustregel ergibt sich:

Prioritäten werden umgekehrt propotional zu den Prozesszeiten vergeben.

Diese Faustregel entspricht der Scheduling-Strategie ***rate monotonic scheduling strategy (analysis) (RMS(A))***.

Liu und Layland [**?**] haben gezeigt, dass mit RMS die Realzeit-Bedingung für alle Tasks eingehalten werden kann, wenn die relative Gesamtbelastung folgende Bedingung erfüllt:

$$\text{Rel.Ges.} \quad = \quad \sum_{i=1}^{n} \frac{t_{c_i}}{t_{p_i}} \quad \leq \quad n(2^{\frac{1}{n}} - 1)$$

Hierbei sei n die Anzahl der Tasks.

$n =$	1	2	3	4	10	100	∞
$n(2^{\frac{1}{n}} - 1) =$	1,000	0,828	0,780	0,757	0,718	0,696	$\ln 2 =$ 0,693

Weitere Beispiele:

Prozess	$t_z = t_p$ ms	$t_c = t_v$ ms
A	10.0	1.0
B	2.0	1.5

Prozess	$t_z = t_p$	$t_c = t_v$
A	24 h	8 h
B	7 ms	1 ms
C	600 s	300 s

4.6 Scheduling-Strategien

Das Kapitel hat sich an der **RMA** ***(Rate Monotonic Analysis)*** oder auch der **RMS** ***(Rate Monotonic Scheduling-Strategy)*** orientiert. Als Ergänzung seien hier erwähnt:

EDF ***(Earliest Deadline First)*****:** Bei jedem möglichen Taskwechsel ermittelt der Scheduler die früheste Deadline.

Die Task mit der frühesten Deadline bekommt den Prozessor. Bei RMA haben niedrigpriore Tasks die Tendenz den zulässigen Zeitbereich fast ganz auszunützen, was u.U. sehr gefährlich aussieht. Bei EDF können Tasks mit großen Prozesszeiten auch früher mit ihrer Reaktion fertig werden.

Allerdings ist EDF ein dynamisches Verfahren, was manche Regelwerke verbieten.

PTS ***(Preemption-Threshold Scheduling)*****:** Zusätzlich zu ihrer RMA-Priorität P_1 hat eine Task eine zweite Priorität $P_2 \geq P_1$. Oft haben mehrere Tasks diesselbe P_2 ***(preemption threshold)***.

Das PTS lässt Unterbrechungen nur durch Tasks zu, deren Priorität höher ist als die Preemption-Threshold P_2.

Insgesamt hat das weniger Taskwechsel zur Folge, ohne dass die Echtzeit-Bedingung verletzt wird. (Das Realtime-BS ThreadX bietet PTS an. Siehe dort auch den User Guide.)

Übung 3:

Task	t_p	$t_v = t_c$
A	6	3
B	8	3
C	10	1

Berechnen Sie die relative Gesamtbelastung.

Vergeben Sie Prioritäten nach RMS und simulieren Sie graphisch.

Wenden Sie EDF an und simulieren Sie graphisch.

Übung 4:

Task	t_p	$t_v = t_c$
A	6	2
B	8	2
C	10	4

Berechnen Sie die relative Gesamtbelastung.

Vergeben Sie Prioritäten nach RMS und simulieren Sie graphisch.

Wenden Sie EDF an und simulieren Sie graphisch.

Wenden Sie PTS an, wobei P_2 gleich der Priorität von Task B sein soll, und simulieren Sie graphisch.

4.7 Harte und weiche Echtzeitbedingungen

Bei vielen Systemen wird eine gelegentliche Verletzung der Echtzeitbedingung nicht als Systemzusammenbruch gewertet. Stankovic hat daher die Begriffe ***Hard and Soft Real-Time Constraints*** – harte und weiche Echtzeitbedingungen – geprägt.

4.7.1 Harte Echtzeitbedingung

Eine "harte" Echtzeitbedingung ***(hard real time constraint)*** liegt vor, wenn bei Nichterfüllung der Bedingung das System ausfällt *(system fails, loss of control)*.

(Bild)

$h(t_r)$ ist die Häufigkeit, mit der eine Reaktionszeit t_r auftritt. Bei der harten Echtzeitbedingung gilt:

$$h(t_r) = 0 \quad \text{für alle} \quad t_r > t_z$$

t_z ist ein "harter" Termin.

Beispiele sind eine Paketsortieranlage, wenn bei nicht rechtzeitiger Weichenstellung Pakete falsch sortiert werden, oder eine Flugzeugsteuerung oder ein Kofferband.

4.7.2 Weiche Echtzeitbedingung

Stankovic spricht von einer weichen EZB, wenn durch die Nichterfüllung einer EZB zwar die Systemleistung beeinträchtigt ***(soft real time constraint)*** *(system degraded)* ist, das System aber noch nicht ausgefallen ist.

(Bild)

In dem Fall spricht man eventuell erst dann von einem Systemausfall, wenn der Schwerpunkt der Häufigkeitsverteilung in die Nähe von t_z wandert.

$$\text{Schwerpunkt} = \frac{\int_0^\infty t_r \, h(t_r) \, dt_r}{\int_0^\infty h(t_r) \, dt_r} \; < \; t_z \cdot \text{Faktor}$$

Die Spezifikation des Faktors ist z.B. Gegenstand der System-Anforderungen. Beispiele für weiche Echtzeitbedingungen sind Buchungssysteme oder interaktive Systeme, wo das System normalerweise innerhalb einer Sekunde reagieren sollte, wo aber auch 10 Sekunden gelegentlich hingenommen werden.

4.8 Entwurfsregeln und Bemerkungen

Echtzeitbedingungen:

- $t_{v_i} \leq t_{z_i}$ für alle Tasks i ist eine *notwendige* Echtzeitbedingung.
- $\sum_i \frac{t_{c_i}}{t_{p_i}} \leq 1$ ist auch nur eine *notwendige* Echtzeitbedingung.
- $t_{r_i} \leq t_{z_i}$ für alle Tasks i ist eine *hinreichende* Echtzeitbedingung oder *die Echtzeitbedingung* schlechthin.
- Besonders kritische Zustände (Alarmzustände) dürfen u.U. die Echtzeitbedingungen verletzen.

Tasking: Die gesamte Automatisierungsaufgabe muss so in parallele Teilaufgaben zerlegt werden, dass die Echtzeitbedingungen eingehalten werden. Hierbei sollte man sich an den Prozesszeiten orientieren.

Faustregel: Pro Prozesszeit wird eine steuernde Task definiert.

Scheduling-Strategie: RMS, EDF oder PTS? RMS lässt sich bei fast allen Systemen anwenden. D.h. die Priorität wird umgekehrt proportional zu t_z vergeben. Da hier die Prioritäten vorab vergeben werden ("Vorabprioritäten", ***static priority***), ist das ein **statisches** Verfahren. Es gibt Regelwerke, wonach Prioritäten zur Laufzeit niemals verändert werden dürfen ("dynamische Prioritäten-Änderung", ***dynamic priority***).

Wie wir in der Übung 3 gesehen haben, lässt sich mit EDF die Realzeitbedingung bei mehr Systemen erfüllen als mit RMS. Allerdings ist EDF ein dynamisches Verfahren und lässt sich in vielen Systemen nicht realisieren. Ferner sollte die relative Gesamtbelastung 50 % nicht übersteigen, so dass RMS immer möglich ist.

PTS ist wieder ein statisches Verfahren, das allerdings nur von wenigen Systemen angeboten wird.

Relative Gesamtbelastung: Die relative Gesamtbelastung $\sum_i \frac{t_{c_i}}{t_{p_i}}$ sollte 0.3 bis 0.5 , d.h. 30% bis 50% nicht übersteigen. Die 85% des oben genannten Beispiels gelten als sehr gefährlich.

Zeitscheibenverfahren: Wenn das Betriebssystem ein Zeitscheibenverfahren ohne Möglichkeit der Prioritätsvergabe hat, dann ist unter folgenden Bedingungen immer noch ein Echtzeitbetrieb möglich: n sei die maximale Anzahl der Tasks. τ sei die Dauer einer Zeitscheibe *(time slice, time quantum,* Zeit, die einer Task zugeteilt wird). Dann gilt:

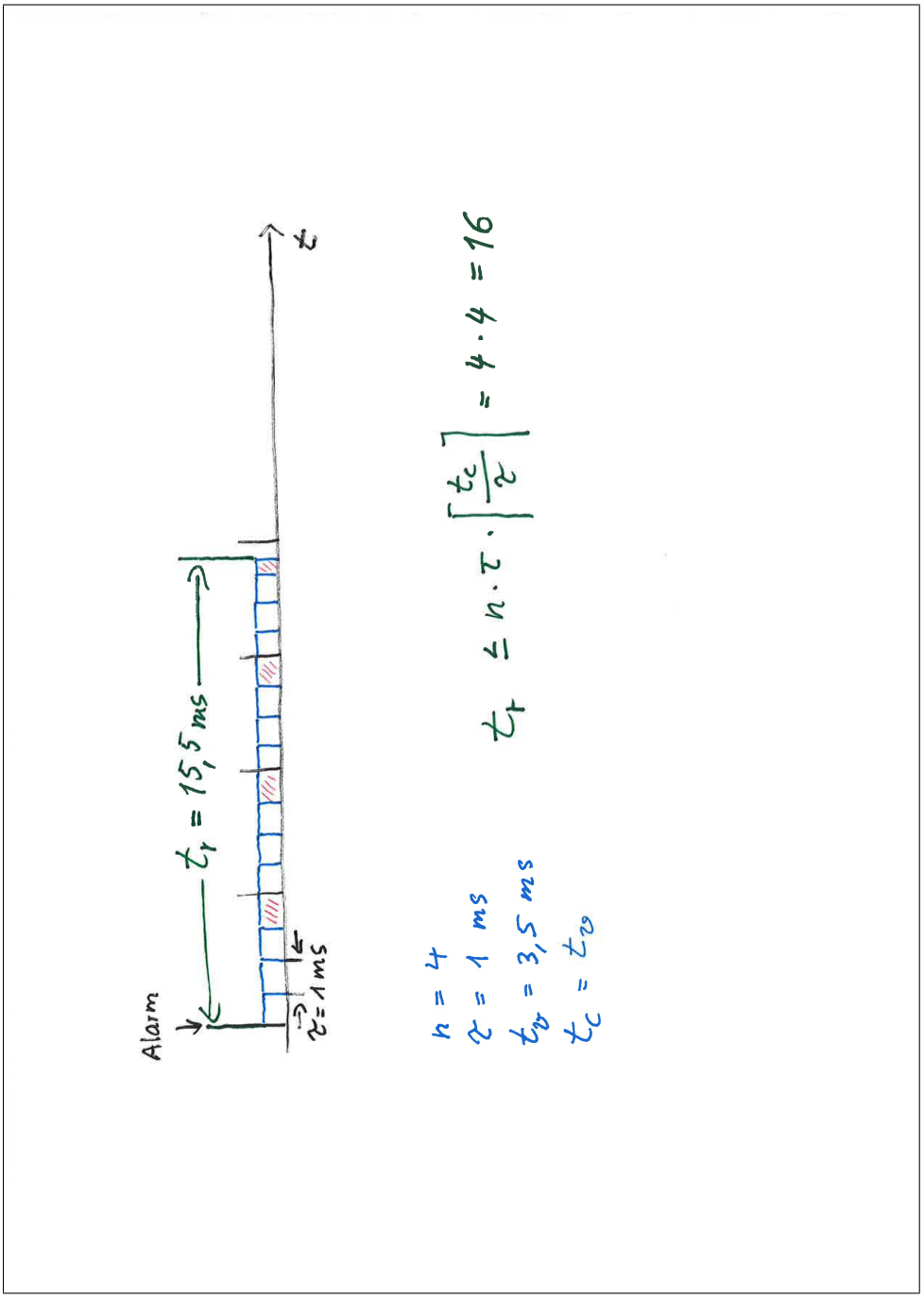
$$t_{r_i} \leq n \cdot \tau \cdot \left\lceil \frac{t_{c_i}}{\tau} \right\rceil + t_{s_i} + t_{b_i}$$

Denn pro Durchlaufzeit $n\tau$ durch alle Tasks wird an der Task i nur τ-lang gearbeitet. Um auf die Prozessorzeit t_{c_i} zu kommen, sind daher höchstens $\lceil \frac{t_{c_i}}{\tau} \rceil$ Durchläufe bzw. höchstens die Zeit $n \cdot \tau \cdot \lceil \frac{t_{c_i}}{\tau} \rceil$ notwendig.

Die Echtzeitbedingung $t_{r_i} \leq t_{z_i}$ wird zu

$$n \cdot \tau \cdot \left\lceil \frac{t_{c_i}}{\tau} \right\rceil + t_{s_i} + t_{b_i} \quad \leq \quad t_{z_i}$$

Wenn diese Bedingung erfüllt werden kann und wenn die Anzahl der Tasks den Wert n nie überschreitet, dann ist das eine ziemlich gute Möglichkeit ein Echtzeitsystem zu realisieren.



Kritische Bereiche: Kritische Bereiche sind Bereiche, die von allen oder manchen Tasks ausschließlich benutzt werden (exklusive Benutzung von Betriebsmitteln oder Ressourcen). Da diese Bereiche das Prioritätsschema stören können, weil die Tasks im kritischen Bereich eventuell mit wesentlich höherer Priorität läuft, sollten kritische Bereiche so kurz wie möglich gehalten werden.

In den Formeln sind diese Bereiche zwar durch t_{b_i} berücksichtigt, aber nicht bei Fällen, wo

die Tasks sich keinen kritischen Bereich teilen. In diesen Fällen sollte man die Abschätzung für t_w modifizieren und CPU-Zeiten in kritischen Bereichen für alle Tasks hinzunehmen, die dann mit höherer Priorität laufen.

Ablaufsteuerung: Die Vergabe von Prioritäten darf auf keinen Fall zur Ablaufsteuerung oder zum Schützen kritischer Bereiche verwendet werden, um etwa zu erreichen, dass eine Task ein Programmstück *vor* einer anderen Task ausführt, oder etwa nach dem Muster: Gib der Task, die einen kritischen Bereich betritt, für die Zeit im kritischen Bereich eine höhere Priorität als ihren Konkurrenten. Das ist deswegen sehr gefährlich, weil anzunehmen ist, dass im kritischen Bereich Wartebedingungen (z.B. auf I/O) eintreten, wodurch die Task suspendiert würde und damit eine niedrigerpriore Task Gelegenheit bekäme, den kritischen Bereich zu betreten.

(Wenn durch glückliche Umstände eine Ablaufsteuerung durch Priorität auf einem Ein-Prozessor-System vielleicht noch funktioniert hat, dann wird es auf einem Mehr-Prozessor-System nicht mehr funktionieren, weil wir da eventuell echte Parallelität haben und Tasks unterschiedlicher Priorität gleich schnell laufen können.)

Die Priorität ist *kein* Mittel zur Synchronisation oder Ablaufsteuerung. Semaphore, Boltvariable, Monitore, Rendezvous und Transactions sind Mittel für die Ablaufsteuerung.

Für das Schützen kritischer Bereiche sollten Lock-Mechanismen (z.B. MUTEXe) verwendet werden.

Jedesmal, wenn eine Wartezeit verwendet wird (um etwa andere Tasks zum Zuge kommen zu lassen), dann ist das ein Zeichen von schlampiger, meist auch gefährlicher Programmierung. Die Verwendung einer Wartezeit ist nur dann möglicherweise gerechtfertigt und eventuell unvermeidbar, wenn sie mit dem zu steuernden Prozess zu tun hat, d.h. wenn das Zeitverhalten die einzige Information ist, die über das Verhalten des realen Prozesses zur Verfügung steht.

Hard-Soft Real-Time Constraints: Auch technische Systeme haben bei genauer Betrachtung häufig weiche Echtzeitbedingungen. Oft ist es daher sinnvoll, eine harte *und* eine weiche Echtzeitbedingung zu spezifizieren. Bei einer Überdruckventilsteuerung z.B. sollte normalerweise innerhalb einer Sekunde reagiert werden, aber Explosionsgefahr besteht erst, wenn einige Minuten lang nicht reagiert wird. Dann wären etwa folgende Spezifikationen sinnvoll:

$$\begin{aligned} t_{z_{\text{soft}}} &= \text{1 Sekunde mit Faktor} = 2 \\ t_{z_{\text{hard}}} &= \text{60 Sekunden} \end{aligned}$$

Deterministisches Verhalten: Ein realer technischer Prozess verhält sich nie **deterministisch**. Denn sonst müsste man ihn nicht steuern. Im Gegenteil, wir müssen davon ausgehen, dass sich der technische Prozess relativ willkürlich verhält.

Allerdings muss sich das AS insofern deterministisch verhalten, als es auf alle möglichen Zustände des technischen Prozesses P in einer vorhersehbaren Weise reagiert. Ziel des AS ist es, deterministisches Verhalten des Gesamtsystems (Technischer Prozess P und AS = PAS) insofern zu garantieren, als dass gewisse spezifizierte Grenzen nicht überschritten werden.

Determinismus *(determinism)* bedeutet, dass zu jedem Systemzustand und jeder dann möglichen Eingabemenge die Ausgabemenge eindeutig ist.

Diese Definition betrifft alle Datenverarbeitungssysteme. Für Echtzeit-Systeme fordern wir zusätzlich ein Zeitverhalten:

Zeitlicher Determinismus *(temporal determinism)* bedeutet, dass die für eine Reaktion benötigte Zeit endlich und in Grenzen vorhersagbar ist.

Vorgehensweise:

- Ermittlung bzw. Definition von Prozesszeiten und zulässigen Reaktionszeiten.
- Tasking: Definition von Teilaufgaben bezüglich der Prozesszeiten mit konzeptioneller Vereinfachung durch Parallelisierung.
- Abschätzung oder Messung der Verarbeitungszeiten und Vergleich mit den zulässigen Reaktionszeiten.
- Berechnung der Relativen Gesamtbelastung.
- Vergabe von Prioritäten.
- Abschätzung der Reaktionszeiten und Vergleich mit den zulässigen Reaktionszeiten.

4.9 Prozess- bzw. Taskbindung

In diesem Abschnitt werden lediglich einige Begriffe vorgestellt, mit denen die Interaktion verschiedener Aktionen, Aktivitäten und Tasks beschrieben werden und die möglicherweise Hinweise für das Tasking liefern.

Zeitliche Bindung: Aktionen und Aktivitäten, die stets in einem definierten Zeitintervall und Reihenfolge ausgeführt werden, werden in einer Task zusammengefasst. Z.B. kann das eine Initialisierungstask sein.

Sequentielle Bindung: Aktivitäten sind so hintereinandergeschaltet, dass die Ausgabe der einen Aktivität die Eingabe der nächsten Aktivität ist (Datenfluss). Wenn die Aktivitäten nebenläufig sein sollen, werden sie durch eine Queue oder Pipe zeitlich entkoppelt.

Bindung durch Kommunikation: Aktivitäten hängen durch intensive Kommunikation so eng zusammen, dass die Nebenläufigkeit stark eingeschränkt ist. Hier bietet sich die Zusammenfassung in einer Task an.

Aktionsbindung: Einzelne Aktionen und Aktivitäten einer Task arbeiten an einer abgeschlossenen Aufgabe.

Informale Bindung: Verwendet eine Task Betriebsmittel, die von anderen Tasks nur im wechselseitigen Ausschluss bearbeitet werden können, so liegt bei dieser Task eine informale Bindung vor.

Harte Echtzeitbedingung: Eine Aktivität, die harten Echtzeitbedingungen unterliegt, wird innerhalb einer Task möglichst ohne Bindung an andere Tasks durchgeführt.

4.10 Beispiel Numerische Bahnsteuerung

Jede Millisekunde soll eine Koordinate x an eine Positioniereinrichtung gegeben werden. Die Steuerdaten liegen als Stützstellen $x(t_i)$ alle 10 ms vor: $t_{i+1} = t_i + 10$ ms. Die dazwischenliegenden Werte (Zwischenstützwerte) werden durch ein Polynom zweiten Grades interpoliert. Zur Bestimmung eines Zwischenstützwerts benötigt der Rechner $t_c = 160$ μs. Die Ausgabe an die Positioniereinrichtung dauert $t_v = 30$ μs, wovon 10 μs CPU-Zeit sind. Für die Feststellung, ob ein Wert ein Zwischenstützwert ist, werden $t_c = 10$ μs benötigt.

Alle 10 ms sind jedoch neue Polynomkoeffizienten zu berechnen. Dies dauert $t_c = 2$ ms.

(Bild)

Welche Prozesszeiten gibt es?

Welche maximal zulässigen Reaktionszeiten gibt es?

Wie groß ist die Relative Gesamtbelastung?

Eine Lösung mit einer Task ist ohne große Umstände offenbar nicht möglich, wie unten stehende Abbildung zeigt:

(Bild)

Wie sieht die Lösung mit mehreren Tasks aus? Dabei kann angenommen werden, dass eine eventuell nötige Intertaskkommunikation in "Schreibrichtung" 100 μs, in "Leserichtung" 50 μs kostet.

(Bild)

Müssen Prioritäten vergeben werden und wie?

Weisen Sie nach, dass die Echtzeitbedingung bei richtigem Tasking und Prioritätsvergabe erfüllt ist.

Beschreiben Sie näher die Intertaskkommunikation. D.h. welches Schreib- und Leseverhalten muss man fordern? Welche Art von Puffer ist geeignet und welchen Einfluss hat der Puffer auf das Verhalten der Tasks?

4.11 Übungen

4.11.1 Beispiel Durchflussregelung

Aufgabe Echtzeitbedingung bei Durchflussregelung

Der Durchfluss `D` und die Temperatur `T` eines Rohres sollen mit DDC geregelt werden. Der Durchfluss wird mit einem PID-Algorithmus, die Temperatur mit einem einfachen schaltenden Zweipunktregler geregelt. Dazu steht ein Einprozessor-Rechnersystem zur Verfügung.

Die Temperaturüberwachung soll jede 6 Millisekunden (6 ms) erfolgen. Bei der Durchflussregelung soll die Abtastperiode 60 ms betragen. Alle gemessenen Durchfluss-Zeit-Wertepaare sollen in eine Datenbank geschrieben werden. Ferner sollen neue Regelparameter aus jeweils 100 Durchfluss-Zeit-Wertepaaren berechnet werden und der Regelung zur Verfügung gestellt werden. Folgende Funktionen stehen für diese Aufgaben zur Verfügung:

`T_Erfassen` A/D-Wandlung der Temperatur: Dauert insgesamt 300 μs (μs = Mikrosekunden), wobei die CPU immer beschäftigt ist ($t_v = t_c = 300$ μs).

`T_Vergleich` Die Temperatur wird mit einem oberen und unteren Temperaturwert verglichen ($t_v = t_c = 100$ μs).

`H_Ausgeben` Ausgabe über eine Digital-I/O-Karte an eine Heizung, die je nach gemessener Temperatur an- oder abgeschaltet wird ($t_v = t_c = 200$ μs).

`D_t_Erfassen` A/D-Wandlung des Durchflusses `D` und Bestimmung der Zeit `t`, zu der der Durchflusswert anfällt. Der A/D-Wandler ist sehr langsam, daher ($t_v = 20$ ms $\quad t_c = 490$ μs).

`DDC_PID` Berechnung der Ventil-Steuergröße `u` mit Hilfe des DDC-PID-Algorithmus unter Verwendung der gerade aktuellen Regelparameter ($t_v = t_c = 4$ ms).

`u_Ausgeben` Ausgabe der Ventil-Steuergröße `u` über eine D/A-Karte an das Rohrventil ($t_v = 10$ ms $\quad t_c = 490$ μs).

`PID_Rechnen` Berechnung neuer PID-Regelparameter aus 100 Durchfluss-Zeit-Wertepaaren ($t_v = t_c = 1500$ ms).

`D_t_Schreiben` Ein Durchfluss-Zeit-Wertepaar wird auf einen gemeinsamen Speicherbereich geschrieben ($t_v = t_c = 10$ μs).

`D_t_Lesen` Ein Durchfluss-Zeit-Wertepaar wird von einem gemeinsamen Speicherbereich gelesen ($t_v = t_c = 10$ μs).

`PID_Schreiben` Ein Satz PID-Parameter wird auf einen gemeinsamen Speicherbereich geschrieben ($t_v = t_c = 10$ μs).

`PID_Lesen` Ein Satz PID-Parameter wird von einem gemeinsamen Speicherbereich gelesen ($t_v = t_c = 10$ μs).

`D_t_Speichern` Schreiben von 100 Durchfluss-Zeit-Wertepaaren in eine Datenbank. Zeitdauer insgesamt $t_v = 1$ s bis 2 s, CPU-Zeit $t_c = 10$ ms.

Das Schreiben auf und Lesen von einem gemeinsamen Speicherbereich erfolgt immer unter gegenseitigem Ausschluss. Das betrifft also die Methodenpaare `D_t_Schreiben`/`D_t_Lesen` und `PID_Schreiben`/`PID_Lesen`.

Aufgaben:
a) Welche Prozesszeiten t_p gibt es in diesem Prozess? Geben Sie die Werte an.

b) Wie kann durch geeignetes Tasking und Verteilung von Prioritäten die Echtzeitbedingung möglicherweise erfüllt werden? (Wenn Sie mit Prioritätszahlen arbeiten, geben Sie deutlich an, was bei Ihnen hohe und niedrige Priorität ist.)
(Antwort z.B. in folgender Art (der Form nach, Inhalt ist falsch) möglich:

```
TASK P Prio12        TASK T Prio20        TASK Prio33
jede 5 ms            jede 20 ms           Forever
T_Erfassen           D_t_Erfassen         for i=1 to 100
D_t_Erfassen         T_Erfassen              DDC_PID
T_Vergleich          D_t_Schreiben           D_t_Lesen
u_Ausgeben           PID_Schreiben        PID_Lesen            )
```

c) Berechnen Sie die relative Gesamtbelastung der CPU :

d) Offensichtlich kann die Echtzeitbedingung bei sequentieller Programmierung nicht eingehalten werden. Besteht auf Grund des errechneten Wertes für die relative Gesamtbelastung Hoffnung, dass die Echtzeitbedingung durch geeignetes Tasking eingehalten werden kann?
Antwort (ja/nein) mit kurzer Begründung:

e) Ist Ihr Wert für die relative Gesamtbelastung eher als kritisch zu betrachten?
Antwort (ja/nein) mit kurzer Begründung:

f) Welch ein gemeinsamer Speicherbereich würde sich für die Durchfluss-Zeit-Wertepaare eignen, d.h. welche Schreib- bzw. Lese-Eigenschaften muss der Speicher haben und wie groß müsste dieser mindestens gewählt werden?

g) Welch ein gemeinsamer Speicherbereich würde sich für die PID-Regelparameter eignen, d.h. welche Schreib- bzw. Lese-Eigenschaften muss der Speicher haben und wie groß müsste dieser mindestens gewählt werden?

h) Ist die Echtzeitbedingung erfüllt? Zur Beantwortung dieser Frage füllen Sie bitte unten stehende Tabelle aus. Die Tabelle ist für 5 Tasks ausgelegt. Wahrscheinlich haben Sie weniger Tasks verwendet. Schreiben Sie komplexere Berechnungsausdrücke für die Zahlen der Tabelle *unter* die Tabelle.

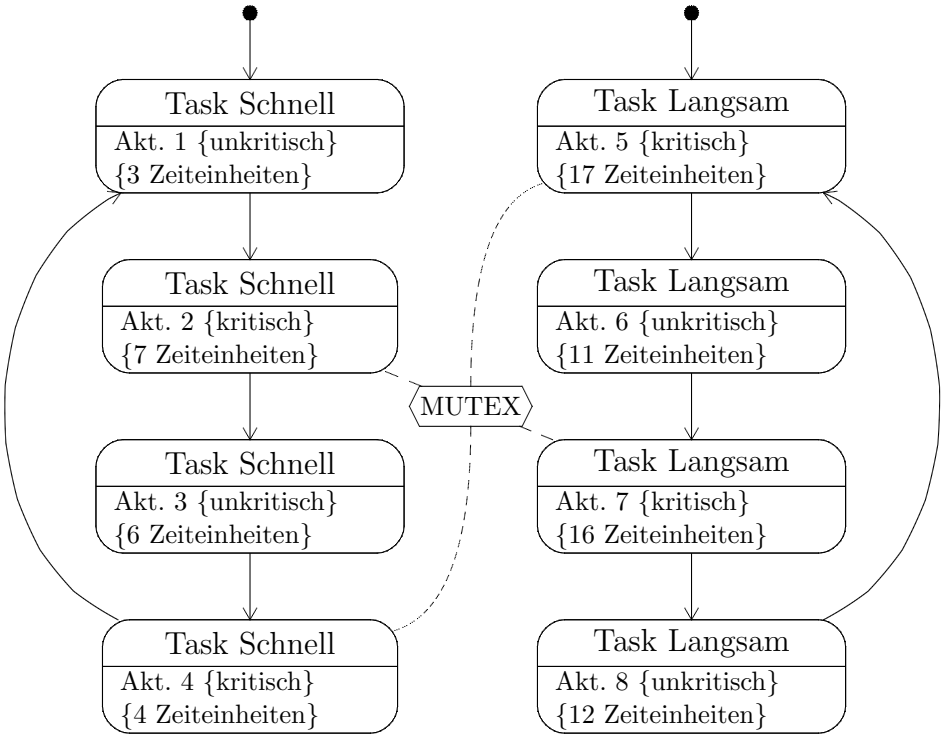
Task					
Zeit-einheit					
t_p					
t_z					
t_c					
t_s					
t_v					
t_w					
t_b					
t_r					
EZB?					

i) Nun machen wir noch eine zusätzliche Annahme: Die Temperaturregelung und die Durchflussregelung greifen über dieselbe serielle Schnittstelle auf den Prozess zu. Das betrifft die Methoden `T_Erfassen`, `H_Ausgeben`, `D_t_Erfassen` und `u_Ausgeben`. Die Zugriffe auf die Schnittstelle erfolgen unter gegenseitigem Ausschluss. Jeder Zugriff dauert 1 ms. `D_t_Erfassen` und `u_Ausgeben` benötigen mehrere solcher Zugriffe, höchstens aber 5. (Am Anfang und dann wird alle 5 ms abgefragt.)
Schätzen Sie die Zeiten t_{b_i} ab.

j) Diskutieren Sie an diesem Beispiel die "harte" und "weiche" Echtzeitbedingung.

4.11.2 Gegenseitiger Ausschluss

In folgendem Beispiel haben eine schnelle Task (`Schnell`) und eine langsame Task (`Langsam`) kritische und unkritische Aktivitäten. Die Aktivitäten verbrauchen keine CPU-Zeit, haben aber eine – hier in Ticks (irgendwelche Zeiteinheiten) angegebene – Dauer. Einige Aktivitäten sind "kritische" Aktivitäten, die nur unter gegenseitigem Ausschluss durchgeführt werden können und durch einen MUTEX geschützt werden.



Aufgabe 1: Machen Sie sich mit Hilfe eines Timeline- oder Sequenz-Diagramms klar, wann welche Aktivitäten laufen.

Aufgabe 2: Schätzen Sie auf Grund des Diagramms ab, was die jeweilige Worst-Case-Zyklusdauer ist.

Aufgabe 3: Schätzen Sie die jeweilige Worst-Case-Zyklusdauer unter Verwendung der Formeln im Abschnitt "Reaktionszeit" ab.

Aufgabe 4: Programmieren Sie eine Simulation.

Kapitel 5

Echtzeitbetriebssystem

Technische Prozesse sind dadurch charakterisiert, dass viele Teilprozesse gleichzeitig ablaufen. Für das Prozessautomatisierungssystem bedeutet das, dass viele Mess-, Steuer- und Regelaufgaben parallel durchgeführt werden müssen. Der Begriff *Echtzeit* kommt daher, dass Reaktionen auf Prozess-Alarme innerhalb vorgegebener Zeiten eine häufig gestellte Anforderung an das Prozessautomatisierungssystem sind.

Daher werden die technischen Teilprozesse durch entsprechende (quasi-)parallel laufende Rechenprozesse oder *Tasks* modelliert. Die Programmieraufgaben, die bei der Verwaltung solcher Prozesse anfallen, sind äußerst schwierig, aber auch immer wieder dieselben Aufgaben. Daher wird die Lösung dieser Aufgaben üblicherweise in das Betriebssystem verlagert.

5.1 Anforderungen an Echtzeitbetriebssystemkerne

Das beste Modell für die interne Struktur eines Betriebssystems ist eine Schichtenhierarchie, auf deren niedrigster Ebene der *Kern* liegt. Der Kern umfasst die wichtigsten, sogenannten "low level" Funktionen des Betriebssystems. Der Kern bildet die Schnittstelle zwischen der Hardware des Rechners und der übrigen Software des Betriebssystems. Der Kern sollte einerseits die einzige Hardware-abhängige Komponente des Betriebssystems sein, andererseits sollte er nur einen minimalen Satz an Operationen zur Verfügung stellen, aus denen der Rest des Betriebssystems konstruiert werden kann.

Heutzutage sind wichtige Anforderungen **Safety** und **Security**, die im IT-Kontext nur als englische Begriffe verwendet werden und üblicherweise folgendermaßen unterschieden werden:

- **Safety:** Sicheres Funktionieren des Systems, insbesondere ohne Gefährdung von Menschen. Zuverlässigkeit. Robustheit. Datensicherheit. Datenkonsistenz.
- **Security:** Sicherheit des Systems vor Attacken von außen. System-Schutz. Datenschutz. Cyber-Security.

Betriebssystemkerne, die in Prozessrechnern eingesetzt werden, heißen *Echtzeitkerne* oder *Realzeitkerne* und sollten folgende Anforderungen erfüllen [14]:

Multitasking: Viele konkurrierende Tasks (laufende Programme, Rechenprozesse) müssen quasiparallel von der CPU abgearbeitet werden. Der *Scheduler* teilt die CPU den Tasks gemäß einer Strategie zu. Die Tasks dürfen sich einerseits nicht stören, andererseits müssen sie auch kommunizieren ***(competing and communicating processes)***. Zur Steuerung dieser Aktivitäten muss der Betriebssystemkern Mechanismen wie *Semaphore, Monitore oder Rendezvous* zur Verfügung stellen.

Priority Based Preemptive Scheduling: Die Ereignisse der realen Welt haben unterschiedliche Wichtigkeitsgrade oder *Prioritäten*. Diese Prioritäten müssen bei der Zuteilung der CPU berücksichtigt werden. Beim prioritätsbasierten unterbrechenden Scheduling wird der Task, die die höchste Priorität hat und die CPU benötigt, die CPU auch zugeteilt, wobei niedriger priorisierte Tasks unterbrochen werden können.

Preemption-Threshold ist eine Eigenart von ThreadX. Wenn z.B. eine Thread eine Priorität von 30 hat und eine Preemption-Threshold von 23, dann kann sie von allen Threads mit Prioritäten 0 bis 22 unterbrochen werden, nicht aber von Threads mit Prioritäten größer 23. Sie selbst kann nur Threads mit Prioritäten größer 30 unterbrechen.

Control Loop with Polling Scheduling: In einer festgelegten Reihenfolge fragt der RTOS-Kern die Tasks, ob sie einen Dienst benötigen. Wenn das der Fall ist, bekommt die Task den Prozessor zur Ausführung des Dienstes, führt den Dienst aus und gibt die Kontrolle zurück an den Kern. Das ist ein sehr kooperativer Ansatz und ist auch unter ***cooperative scheduling*** bekannt. Ein Variante davon ist das **Zeitscheibenverfahren** ***(time-slicing)***, wo einer Task die CPU nur für ein kurzes Zeitintervall zur Verfügung steht mit dem Nachteil, dass man sehr viele, oft unnötige Kontextwechsel hat.

Diese Scheduling-Varianten kommen oft zum Einsatz bei Tasks gleicher Priorität, wenn ansonsten ein Priority Based Preemptive Scheduling verwendet wird.

Intertaskkommunikation (Interprozesskommunikation): Da die Tasks Nachrichten austauschen, müssen dafür effektive Mechanismen wie z.B. Pipes, Queues ***(asynchronous or synchronous message passing)*** oder Shared Memory zur Verfügung gestellt werden.

Synchronisationsmechanismen: (Locks, Semaphore, Rendezvous oder Monitore) müssen für die gemeinsame bzw. ausschließliche Nutzung von kritischen Ressourcen zur Verfügung gestellt werden.

Leistungsfähigkeit: Ein Echtzeitkern muss auf den "Worst Case" und nicht auf maximalen Durchsatz optimiert werden. Ein System, das eine Funktion konsistent in 50 μs ausführt, ist für die Prozesssteuerung eher geeignet als ein System, das für diese Funktion durchschnittlich 20 μs, in Ausnahmefällen aber 80 μs benötigt.

Interrupt: Es muss möglich sein, die Abarbeitung eines Interrupts außerhalb der eigentlichen Unterbrechungsschicht (ISR, *interrupt service routine*) durchzuführen. Das System muss auf einen Interrupt möglichst schnell reagieren, die Abarbeitung aber an andere Tasks weiterreichen können, damit es auf weitere Interrupts ebenso schnell reagieren kann.

Task, Prozess, Thread: Es werden die Begriffe **Task**, **Prozess** (hier nicht als "zu steuernder Prozess", sondern als "Rechenprozess") und ***Thread*** verwendet. Für uns ist Task der übergeordnete Begriff.

Ein (Rechen-)Prozess hat einen eigenen Adressraum. Diese Prozesse können nur über Betriebssystem-Mittel kommunizieren. Prozesse werden vom Betriebssystem verwaltet.

Threads dagegen sind Leichtgewichtsprozesse ***(lightweight process)*** ohne eigenen Adressraum. Sie können auch über globale Variable kommunizieren. Mehrere Threads können zu einem Prozess gehören. Der **Context-Switch** auf Thread-Ebene ist wesentlich einfacher als auf Prozess-Ebene.

Ein *Programm* ist eine statische Aufschreibung einer Folge von Anweisungen, während die *Task* als Programm*ablauf* zu verstehen ist.

Objekt-orientiert ist eine Task ein Objekt einer Klasse, die eine Methode hat, deren Code als Thread oder Prozess gestartet werden kann.

5.2 Typische Werkzeuge von Echtzeitbetriebssystemen oder Sprachen

5.2.1 Pipes, Queues

(siehe Kapitel Concurrency Utlilities und Kapitel Threads Ringpuffer)

5.2.2 Watchdog Timer

Ein Watchdog Timer ermöglicht den Aufruf einer Routine nach einer vorgegebenen Zeitverzögerung d. Innerhalb des Zeitintervalls d kann der Aufruf der Routine zurückgenommen werden bzw. kann der Timer zurückgesetzt werden ("padding des Hundes").

I.A. ist das so realisiert, dass die Funktion normalerweise mit `true`, im Abbruchsfall aber mit `false` zurückkommt.

5.2.3 Timetables

Startzeiten von Tasks erfolgen nach einem festgelegten Zeitplan.

5.3 Vergleich von Echtzeitbetriebssystemen

Ein belgische Firma testet und vergleicht Betriebssysteme (`www.dedicated-systems.com`). Das dürfte bei einer Betriebssystem-Entscheidung sehr interessant sein.

Bei einem Vergleich spielen insbesondere zwei Zeiten eine Rolle:

- ISR (etwa 1μs): Zeit, bis erster Befehl in einer Interrupt Service Routine nach einem Interrupt durchgeführt wird.
- IST (etwa 10μs): Zeit, bis in einer Interrupt Service Routine nach einem Interrupt ein Thread gestartet wurde.

Grob unterscheiden wir drei Architekturen:

- ***flat architecture:*** Betriebssystemkern und Anwendung befinden sich auf derselben (der einzigen) Privilegienebene. Anwendung und Betriebssystemkern sind nicht klar unterscheidbar. Es gibt keinerlei Speicherschutz.
- ***monolithic architecture:*** (WinCE) Betriebssystemkern enthält alle Bibliotheken, auf die die Anwendung direkt zugreift. Es gibt mindestens zwei Privilegienebenen, Kern und User. Für die Kern-Prozesse gibt es keinen Speicherschutz.
- ***micro kernel architecture:*** (VxWorks, QNX) Betriebssystemkern enthält keine Bibliotheken. Diese befinden sich auf einer eigenen Ebene. Die Anwendung kann aber nur über den Kern auf die Bibliotheken zugreifen. Voller Speicherschutz.

Kapitel 6

Echtzeitsystem-Entwicklung

Dieses Kapitel diskutiert Aspekte des Software-Engineerings, die speziell bei Echtzeitsystemen (eingebettete Systeme) oder generell **nebenläufigen *(concurrent)*** Systemen eine Rolle spielen und über die bisher genannten hinausgehen.

Echtzeitsysteme sind Automatisierungssysteme, die einen technischen Prozess steuern oder automatisieren. Diese Prozesse sind charakterisiert durch:

- Technische Prozesse bestehen aus vielen (Teil-)Prozessen.
- Die Prozesse laufen parallel **(Nebenläufigkeit)**.
- Die Prozesse unterliegen gewissen **Zeitbedingungen**.
- Prozesse sind Einzelkomponenten **(Task)** des Entwurfs.
- Prozesse steuernde Programme sind sequentielle Programme. Für ihre interne Strukturierung können Entwicklungsprinzipien des sequentiellen Entwurfs verwendet werden.
- Bei technischen Prozessen müssen **Sicherheits-Aspekte** berücksichtigt werden.

Die besondere Herausforderung bei der Entwicklung eines Echtzeit-Systems liegt darin, dass in der Regel sehr eng mit anderen Disziplinen – ("Hardware") **mechanisches Engineering** oder **elektrisches Engineering** – zusammengearbeitet werden muss. Dabei zeigt sich als Tendenz, dass immer mehr Funktionen der Hardware-Disziplinen durch Software realisiert werden.

Moderne Echtzeit-Software wird objekt-orientiert entwickelt. Dazu wird UML eingesetzt. Bezüglich UML gehen wir hier nur auf die speziellen Anforderungen von Echtzeitsystemen und deren Modellierung ein. Ein wichtiger Grund für den Erfolg des objektorientierten Ansatzes ist eine klare Unterscheidung von

> "**was** ein Objekt tut" (Abstraktion, Schnittstelle eines Objekts)
> und
> "**wie** ein Objekt etwas tut" (Realisierung, Implementierung, Repräsentation eines Objekts).

Dieses Prinzip wird unterstützt durch die konsequente Verwendung von Schnittstellen.

Echtzeitsysteme sind von Natur aus **nebenläufig, asynchron** und **verteilt** ***(concurrent, asynchronous, distributed)***. Die Steuerungsrechner von Robotern an einer Bandstraße laufen parallel nebeneinander. Die Abarbeitung erfolgt asynchron, d.h. es gibt i.A. keine feste Reihenfolge der Ereignisse verschiedener Roboter. Allerdings werden sich die Roboter gelegentlich synchronisieren, d.h. eventuell aufeinander warten. Ferner sind die Roboter räumlich verteilt.

Bei einem mechanischen technischen Prozess hat der mechanische Teil typischerweise eine Lebenszeit zwischen 5 und 20 Jahren, während die verwendete Hard- und Software-Technologie nur eine Lebenszeit zwischen 3 und 5 Jahren hat. Daher spielt der Aspekt der Portierung von Software von einem veralteten HW-SW-System zu einem moderneren System ein große Rolle.

6.1 Benutzerschnittstelle

Bei Echtzeitsystemen kann die Benutzerschnittstelle sehr unterschiedlich sein. Die Wechselwirkung mit einem menschlichen Bediener reicht vom Betätigen eines Schalters bis zu einer voll-animierten graphischen Oberfläche oder extrem spezialisierten Schnittstellen.

In der Luftfahrtindustrie ist das Prinzip ***fly-by-wire*** schon verwirklicht. Die Piloten haben keine direkte Verbindung mehr zum mechanischen System des Flugzeugs. Alles läuft über einen oder mehrere Computer. In der Automobilindustrie entwickelt man dieses Prinzip ***(drive-by-wire)*** gerade.

Wenn die Schnittstelle Mensch-Prozess gut gemacht ist, dann erleichtert sie ganz wesentlich die Prozessbedienung. Wenn sie aber schlecht und fehlerhaft ist, dann führt das zu Sicherheitsproblemen, insbesondere in Stress-Situationen oder bei ungewöhnlichen Zuständen.

6.2 Analyse

Bei Echtzeitsystemen hat sich der Begriff **Essentielles Modell** mit den Teilen **Umgebungs-Modell** und **Verhaltens-Modell** etabliert.

Das Umgebungs-Modell ist die Beschreibung des zu automatisierenden technischen Prozesses. Dieser bildet die "Umgebung" des zu entwickelnden Automatisierungssystems. Z.B. könnte hier eine Liste der externen Ereignisse (vom technischen Prozess kommend) und der externen Aktionen (auf den technischen Prozess wirkend) erstellt werden.

Das Verhaltens-Modell beschreibt, wie das Automatisierungs-System auf die Ereignisse zu reagieren hat.

Oft ist ein Datenflussdiagramm bei Echtzeitsystemen sehr nützlich (siehe Kapitel "Funktionales Modell").

6.3 Testen

Testen ist bei Echtzeitsystemen besonders schwierig, da der reale Prozess i.A. nicht unter beliebigen Szenarien gefahren werden kann. Der reale Prozess kann nicht bis zu seiner Zerstörung

gefahren werden. Daher muss oft simuliert werden, was die Erstellung eines Simulationsmodells erfordert, das aber immer auch eine gewisse "Entfernung" vom realen Prozess hat.

6.4 Simulation

Man scheint sich darüber einig zu sein, dass die Einführung eines Simulationsschritts insgesamt Zeit spart.

Die Entwicklung eines Simulationsmodells wird häufig als überflüssige Projektverzögerung gesehen. Denn es ist sehr aufwendig, ein solches Modell zu entwickeln. Aber alle Probleme, die in einer "freundlichen" Simulationsumgebung erkannt werden, können dort viel ökonomischer gelöst werden.

Es kann allerdings sein, dass die Übertragung eines Systems von der Simulationsumgebung in die Realität nicht trivial ist.

Das Simulationsmodell sollte nicht mehr Detail enthalten als für den Test der Steuerungssoftware notwendig ist.

Ein Simulationsmodell fördert das Prozessverständnis.

6.5 Agenten-Modell

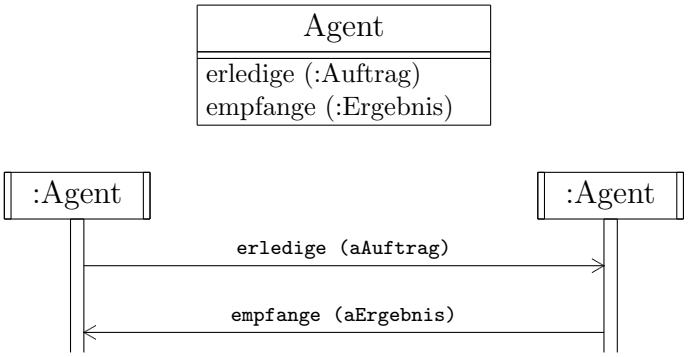
Wir modellieren Echtzeitsysteme mit **aktiven Objekten** (UML-Konstrukt) oder **Prozessen**. In der Echtzeitliteratur wird dafür oft der Begriff **Aktor** ***(actor)*** [18] verwendet, der in UML und auch sonst in der Prozessdatenverarbeitung allerdings eine andere Bedeutung hat. Wir entscheiden uns daher hier für den Begriff **Agent**. Der Agent ist ein aktives Objekt und der Begriff "Agent" unterstreicht die **Autonomie** des Objekts. Außerdem wird eine gewisse "Intelligenz" des Agenten suggeriert. Ein Agent kann eine oder mehrere Funktionen oder Methoden als **Task** durchführen, d.h. nebenläufig zu anderen Tasks sein.

Das Agenten-Modell ist sehr allgemein. Ein Agent kann z.B. einen Prozess, eine Steuereinheit, einen Rechner, einen Speicherbereich, einen Sensor oder Aktor repräsentieren.

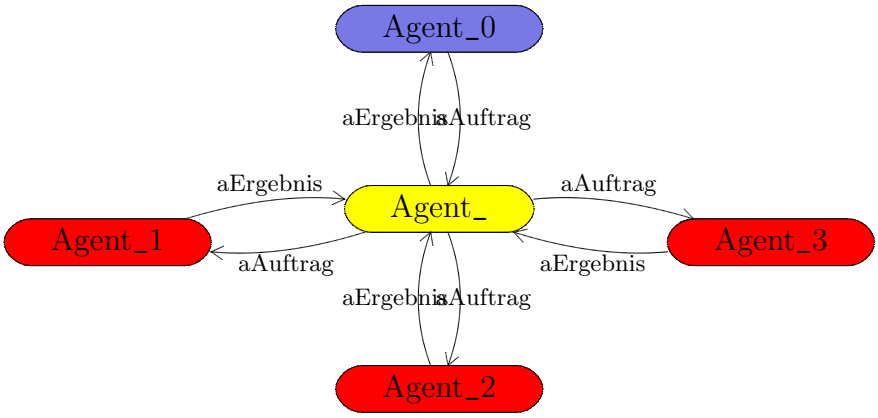
6.5.1 Spezifikation des Agenten-Modells

Agenten kommunizieren nur über **asynchrone Botschaften** ***(asynchronous message passing)***. Ein Agent muss also nicht auf eine Antwort seiner Botschaft an einen anderen Agenten warten, sondern kann seine Aktivitäten parallel weiter ausführen.

Typischerweise erhält der Agent als Botschaft **Aufträge** anderer Agenten, die dann asynchron ein **Ergebnis** von ihm erwarten. Der Agent kann den Auftrag entweder selbst komplett bearbeiten oder er beauftragt weitere Agenten.



Als Datenflussdiagramm (siehe Kapitel "Funktionales Modell") könnte das folgendermaßen aussehen:



Das Senden einer Botschaft an einen Agenten ist als **Ereignis** ***(event)*** definiert. Es gibt keine globale Reihenfolge von Ereignissen.

Regeln für den Botschaftenaustausch:

1. Botschaften werden ***point-to-point*** versendet, d.h. von Agent zu Agent.
2. Das Senden einer Botschaft ist eine ***non-blocking*** Operation. Der Sender wartet nicht auf eine Antwort, sondern kann seine Programmausführung sofort wieder aufnehmen.
3. Gesendete, aber noch nicht verarbeitete Botschaften werden gepuffert. Botschaften gehen nicht verloren. Sie werden beliebig lang gepuffert. (Im Gegensatz dazu werden **Signale** nicht gepuffert. Wenn Signale nicht empfangen werden, gehen sie verloren.)
4. Gesendete Botschaften erreichen garantiert ihr Ziel, aber möglicherweise verzögert durch den Kommunikationsweg. Eine Reihenfolge der Botschaften wird nicht garantiert.
5. Eine Botschaft enthält den Absender, damit irgendwann eine Antwort an den Absender geschickt werden kann.

6.5.2 Realisierung des Agenten-Modells

Zur Diskussion der Realisierungsmöglichkeiten des Agenten-Modells beschränken wir uns auf Java.

Die Aktivität des Agenten wird als Thread realisiert, der gegebenenfalls Botschaften an andere Agenten verschickt.

Eine Botschaft ist ein Objekt einer Klasse, die vorzugsweise eine entsprechende Schnittstelle, etwa die Schnittstelle `Botschaft` oder `Auftrag/Ergebnis` realisiert.

Botschaften werden versendet, indem eine Methode eines anderen Agenten mit dem Botschaftsobjekt als Argument aufgerufen wird. Diese Methoden könnten

```
empfange (Botschaft x)
```

oder spezifischer

```
erledige (Auftrag x)
empfange (Ergebnis x)
```

heißen.

Als Transport-Medium kann natürlich eine Message-Queue verwendet werden.

Bei verteilten Systemen kann dies über RMI oder – elementarer – über UDP-Botschaften realisiert werden.

Um Botschaften (Ergebnis oder Aufträge) zu empfangen, muss ein Agent die Methoden `empfange (Botschaft x)` (`empfange (Ergebnis x)` oder `erledige (Auftrag x)`) realisieren. Diese Methoden werden die Botschaft (das Ergebnis oder den Auftrag) nur in einen Puffer schreiben, typischerweise einen Ringpuffer, der prinzipiell unendlich groß werden kann.

Wie erfährt nun der Agent, d.h. der Thread des Agenten ("Agenten-Thread") von dem Eingang einer Botschaft? Wahrscheinlich ist es am elegantesten, wenn ein zweiter "Lese-Thread" kontinuierlich versucht, Botschaften aus dem Ringpuffer zu lesen. (Ein vernünftiger Ringpuffer wird diesen Thread suspendieren, wenn keine Botschaften zu lesen sind.) Der Lese-Thread kann dann den eigentlichen Agenten-Thread benachrichtigen und zur Behandlung des Ereignisses auffordern. Das erfolgt synchron, d.h. der Lese-Thread wird warten, bis das Ereignis behandelt ist, und dann die nächste Botschaft lesen. Der Lese-Thread kann eventuell auch entscheiden, ob eine Botschaft gerade relevant ist oder verworfen werden kann. (Ein Zurückstellen in den Puffer dürfte nicht sinnvoll sein, da dann der Lese-Thread dauernd mit Lesen und wieder Zurückstellen beschäftigt wäre.)

6.6 Beenden von Tasks

Es ist normalerweise nicht problematisch eine Task zu starten. Auch das Ende einer Task ist unproblematisch, wenn sie an ihr natürliches Ende läuft. Aber es gibt Fälle, wo Tasks von außen angehalten oder abgebrochen werden müssen. Die Schwierigkeit dieser Fälle wird i.A. unterschätzt. Die dafür in vielen Systemen angebotenen (auf Tasks anwendbare) Methoden `suspend/resume`

und `kill`, `cancel` oder `stop` sind gefährlich und führen zu Laufzeitproblemen, die kaum zu lösen sind. Bei diesen Methoden kann eine Task nicht die Aufräumarbeiten durchführen, die normalerweise vor einem Abbruch immer notwendig sind.

Anstatt eines "gewaltsamen" Eingriffs in den Ablauf der Task, sollte ein **kooperativer** Mechanismus gewählt werden, wobei der Task ein Interrupt geschickt wird, den sie selbst behandelt. Da dies das Design und die Implementierung verkompliziert, wird diese Problematik – Beendingung von Tasks ***(end-of-lifecycle issues, graceful shutdown)*** – gern vernachlässigt.

Der kooperative Mechanismus funktioniert i.A. folgendermaßen:

1. **Taskstruktur:** Die Task verwaltet eine ihr zugeordnete **Zustandvariable** mit folgenden Werten:
   (a) `laufend` ***(running)***
   (b) `beendet` ***(terminated)***
   (c) `suspendiert` ***(suspended)***

   Diese Zustandsvariable kann von außerhalb der Task *nur* gelesen werden.

   Ferner verwaltet die Task eine ihr zugeordnete **Zustandsänderungsvariable** mit folgenden Werten:
   (a) `beenden` ***(cancel)***
   (b) `suspendieren` ***(suspend)***
   (c) `weiter` ***(resume)***

   Die Zustandsänderungsvariable kann von außerhalb der Task gesetzt werden.

2. **Taskverhalten:** Die Task überprüft in passenden, aber möglichst kurzen Zeitabständen die Zustandsänderungsvariable. Entsprechend dem Wert dieser Variablen wird die Task entweder
   (a) einfach weiterlaufen (im Zustand `laufend` auf `weiter` gesetzt) oder
   (b) sich "vernünftig" beenden (auf `beenden` gesetzt) oder
   (c) "vernünftig" anhalten (auf `suspendieren` gesetzt) oder
   (d) aus einer Suspendierung weiterlaufen (im Zustand `suspendiert` auf `weiter` gesetzt).

   Die Task aktualisiert bei Erreichen eines neuen Zustands die Zustandsvariable.

   Die Task darf im Zustand `laufend` durchaus auch "schlafen" oder sich anderweitig suspendieren, wenn es dafür einen Interrupt-Mechanismus gibt.

   Nach einem Interrupt muss auf jeden Fall die Zustandänderungsvariable überprüft werden.

3. **Abbruch** ***(kill, stop, cancel)*** oder **Beenden** einer Task:
   (a) Die Zustandsänderungsvariable der Task wird auf `beenden` gesetzt.
   (b) Der Task wird ein Interrupt geschickt, um sie eventuell zu wecken.

4. **Suspendierung** ***(suspend)*** einer Task:
   (a) Die Zustandsänderungsvariable der Task wird auf `suspendieren` gesetzt.

   (b) Der Task wird ein Interrupt geschickt, um sie eventuell zu wecken.

5. **Weiterlaufenlassen *(resume)*** einer Task:

   (a) Die Zustandsänderungsvariable der Task wird auf `weiter` gesetzt.

   (b) Der Task wird ein Interrupt geschickt für den Fall, dass sie schläft, was sie in dem Fall ja auch tun sollte.

Bemerkungen:

Erzeuger-Verbraucher-Kommunikation: Oft ist es sehr schwierig eine Erzeuger-Verbraucher-Kommunikation über eine Queue sicher herunterzufahren, wenn die Schreib- und Lese-Methoden der Queue nicht unterbrechbar sind, was oft der Fall ist. Bevor man eigene Queues schreibt, wird man verschiedene Tricks versuchen. Oft wird eine sogenannte Giftpille ***(poison pill)*** verwendet, die der Erzeuger bei Abbruch der Kommunikation in die Queue schreibt. Der Verbraucher bricht dann ab, wenn er die Giftpille gelesen hat. Das Verfahren kann sich erheblich verkomplizieren, wenn man mehrere Erzeuger und Verbraucher hat (siehe [6] Seite 147).

Task Leakage: Man verliert eine Task, weil sie auf Grund eines Fehlers abgebrochen wird. Das wird oft nicht bemerkt, wenn diese Task eine von vielen gleichartigen Tasks ist, wenn z.B. ein Service mehrfach gestartet wird. Hier muss man eventuell Mechanismen einbauen, die das Verschwinden einer Task signalisieren.

Kapitel 7

Threads

Java bietet die Möglichkeit der Parallel-Programmierung, indem man besondere Methoden als eigene **Threads** ***(thread)*** laufen lassen kann. Bei einem Ein-Prozessor-System sind das quasi-parallel ablaufende Steuerflüsse innerhalb eines Programms. Ein Scheduler simuliert die Parallelität, wobei es verschiedene Strategien gibt.

Man unterscheidet gemeinhin **Prozesse** und **Threads**. Prozesse haben einen eigenen Adressraum (Speicherbereich) und können nur über Betriebssystem-Mittel kommunizieren. Threads haben keinen eigenen Adressraum und können z.B. auch über gemeinsame Variablen kommunizieren. Threads werden auch als **Leichtgewichts-Prozesse** ***(light weight process)*** bezeichnet. Als übergeordneter Begriff ist **Task** zu sehen, der die Ausführung eines **Programms** bezeichnet. Ein Programm ist statisch und kann als Task mehrmals und auch parallel ausgeführt werden.

Der Java-Scheduler lässt einen aktiven Thread so lange laufen, bis er entweder von einem Thread mit höherer Priorität unterbrochen wird, oder bis er selbst die Kontrolle abgibt, weil er durch Ein- oder Ausgabe blockiert wird oder weil er ein `yield`, `sleep` oder `wait` macht ***(priority based preemptive scheduling without time slicing)***. Sind zu einem Zeitpunkt mehrere Threads mit höchster Priorität bereit zu laufen, dann beginnt der Scheduler mit dem Thread, der am längsten gewartet hat.

Man muss aber die Gegebenheiten des zugrundeliegenden Betriebssystems beachten: evtl. gibt es doch Zeitscheiben oder weniger (bzw. mehr) Prioritätsstufen. Windows hat zum Beispiel nur sieben Prioritätsstufen und unter Linux steht uns nur eine – also keine! – Prioritätsstufe zur Verfügung.

Der Programmierer sollte sich nicht auf ein Verhalten des Schedulers verlassen. Er sollte weder "Gerechtigkeit noch nachvollziehbare Logik" vom Scheduler erwarten [20].

7.1 Einzelner Thread

Zur Erzeugung eines Threads muss man eine Klasse schreiben, die entweder von der Klasse `Thread` erbt oder die die Schnittstelle `Runnable` realisiert. Dabei wird die Methode `run ()` mit dem eigentlichen Thread-Code implementiert.

Die den Thread startende Methode `start ()` stellt die Klasse `Thread` zur Verfügung. Sie ruft im wesentlichen die Methode `run ()` auf, d.h sie lässt den Code der Methode `run ()` als Thread laufen. Die Schnittstelle `Runnable` deklariert die Methode `run ()`. Die Methode `run ()` enthält die eigentliche Aktivität des Threads.

Mit einem Objekt vom Typ der Schnittstelle `Runnable` kann ein Objekt vom Typ `Thread` initialisiert werden, so dass die beiden Möglichkeiten, eigenen Code als Thread laufen zu lassen, nun folgendermaßen aussehen:

1. Die eigene Klasse, etwa `MeinThread1`, erbt von `Thread` und überschreibt die Methode `run ()` mit eigenem Code:

```
public class   MeinThread1 extends Thread
   {
   public void run ()
      {
      // Eigentlicher Threadcode
      }
   }
```

   Im Anwendungscode wird der eigene Thread erzeugt und durch Aufruf von `start ()` gestartet:

```
MeinThread1 p = new MeinThread1 ();
p.start ();
```

2. Die eigene Klasse, etwa `MeinThread2`, realisiert `Runnable` und implementiert die Methode `run ()` mit eigenem Code:

```
public class   MeinThread2 implements Runnable
   {
   public void run ()
      {
      // Eigentlicher Threadcode
      }
   }
```

   Im Anwendungscode wird ein Thread mit einem Objekt der eigenen Klasse initialisiert und durch Aufruf von `start ()` gestartet. Dabei wird aber das `run ()` der eigenen Klasse als Thread gestartet:

```
MeinThread2 ob = new MeinThread2 ();
Thread   p = new Thread (ob);
p.start ();
```

Diese Variante muss man wählen, wenn man sonst mehrfach erben müsste.

Die beiden folgenden Beispiele zeigen jeweils eine primitive Ausarbeitung dieser Konzepte.

Erben von `Thread`:

```
public class   ErbtThread
   {
   public static void   main (String[] argument)
      {
      MeinThread1 herr = new MeinThread1 ("Herr");
      herr.start ();

      MeinThread1 knecht = new MeinThread1 ("Knecht");
      knecht.start ();
      }
   }

class MeinThread1 extends Thread
   {
   private String name;

   public   MeinThread1 (String name)
      {
      this.name = name;
      }

   public void run ()
      {
      while (true)
         {
         System.out.println ("Thread " + name + " läuft.");
         //yield ();
         }
      }
   }
```

Implementieren von `Runnable`:

```
public class   ImpleRunnable
   {
   public static void   main (String[] argument)
      {
      MeinThread2 herr = new MeinThread2 ("Herr");
      Thread   herrTh = new Thread (herr);
      herrTh.start ();
```

```
      MeinThread2 knecht = new MeinThread2 ("Knecht");
      Thread   knechtTh = new Thread (knecht);
      knechtTh.start ();
      }
   }

class MeinThread2 implements Runnable
   {
   private String name;

   public   MeinThread2 (String name)
      {
      this.name = name;
      }

   public void run ()
      {
      while (true)
         {
         System.out.println ("Thread " + name + " läuft.");
         //Thread.yield ();
         }
      }
   }
```

7.2 Methoden der Klasse `Thread`

`start ()`: Mit `start ()` wird ein Thread gestartet. Der Thread läuft solange, bis die `run`-Methode an ihr natürliches Ende gelangt ist. Ein Neustart ist nur möglich, indem man das Thread-Objekt neu erzeugt und wieder startet. D.h. ein Thread-Objekt kann nur ein einziges Mal als Thread gestartet werden.

`Thread.sleep ()`: Mit `sleep (long ms)` bzw. `sleep (long ms, int nanos)` wird der Thread `ms` Millisekunden und eventuell `nanos` Nanosekunden angehalten. (Bei `sleep (0)` wird der Thread nicht angehalten. Negative Werte sind illegal.)

`Thread.yield ()`: Mit `yield ()` wird die Kontrolle an einen lauffähigen Thread gleicher Priorität abgegeben. `yield ()` und `sleep (...)` sind `static` Methoden. Diese kann ein Thread also nur für sich selbst aufrufen.

Mit `yield ()` kann ein "kooperatives" Verhalten oder Multitasking unter Threads gleicher Priorität programmiert werden.

`join ()`: `p.join ()` wartet solange, bis der Thread `p` zu Ende gekommen ist. Mit `p.join (long ms)` bzw. `p.join (long ms, int nanos)` wird höchstens `ms` Millisekun-

den und eventuell `nanos` Nanosekunden gewartet. (Bei `p.join (0)` wird unendlich lange gewartet.)

`interrupt ():` Mit `interrupt ()` wird ein Unterbrechungsflag für die Task gesetzt (auf `true`), das mit `isInterrupted ()` oder `Thread.interrupted ()` abgefragt werden kann, wobei nur die statische Methode `Thread.interrupted ()` das Flag wieder zurücksetzt (auf `false`).

`sleep`, `join` und `wait` werden mit `interrupt ()` abgebrochen, wobei eine `InterruptedException` geworfen und gleichzeitig das Flag zurückgesetzt wird. Wenn das `interrupt ()` vor erreichen von `sleep`, `join` und `wait` erhalten wird, dann wird es bei Erreichen dieser Methoden wirksam, wobei die `InterruptedException` geworfen wird, es sei denn, dass vorher `Thread.interrupted ()` aufgerufen wurde.

`join ()` und `interrupt ()` sind die einzigen direkten Inter-Thread-Kommunikations-Möglichkeiten (indirekt auch `notify ()` und `notifyAll ()`).

`stop (), suspend (), resume ():` Die Methoden `stop`, `suspend` und `resume` gibt es nicht mehr, weil sie sehr fehlerträchtig sind. Daher stellt sich die Frage, wie ein Thread zu unterbrechen bzw. abzubrechen ist. Für das Unterbrechen bieten die Methoden `sleep`, `join` und `wait` genügend Möglichkeiten. Für den Abbruch wird folgendes empfohlen: Der Thread sollte regelmäßig eine Variable (für prompte Reaktion vorzugsweise als `volatile` deklariert) überprüfen. Wenn die Variable anzeigt, dass der Thread stoppen sollte, dann sollte sie an ihr natürliches oder an ein vernünftiges Ende laufen.

`Name:` Die Klasse `Thread` hat das Attribut `Name` mit den Methoden `setName (String name)` und `getName ()`. (Daher wäre in dem Beispiel oben die Verwaltung des Namens überflüssig gewesen, wenn man von `Thread` erbt.)

`isAlive ():` Mit der Methode `isAlive ()` kann überprüft werden, ob ein Thread noch läuft bzw. überhaupt gestartet wurde.

`Thread.currentThread ():` Gibt eine Referenz auf den gerade laufenden Thread zurück.

7.3 Gruppen von Threads

Jeder Thread gehört einer Threadgruppe an. Jede Threadgruppe, außer der "Wurzel"-Threadgruppe, gehört einer Threadgruppe an.

Mit der Gruppierung von Threads kann man Steuermethoden wie z.B. `interrupt ()` nicht nur für einzelne Threads, sondern für ganze Gruppen von Threads aufrufen. Ferner kann man Gruppenhierarchien bilden.

Eine Thread-Gruppe kann eine maximale Priorität haben.

Mit den Konstruktoren

```
ThreadGroup (String name) und
ThreadGroup (ThreadGroup gruppe, String name)
```

können Threadgruppen mit einem Namen angelegt werden und gegebenenfalls einer anderen Threadgruppe untergeordnet werden. (Default ist die Wurzel.) Ein Thread kann mit den Konstruktoren

`Thread (ThreadGroup gruppe, String name)` oder
`Thread (ThreadGroup gruppe, Runnable objekt, String name)`

einer Gruppe zugeordnet werden.

7.4 Priorität

Mit

```
public final void setPriority (int prioritaet)
```

kann die Priorität eines Threads gesetzt werden. Je höher die Zahl, desto höher ist die Priorität. Sie ist einstellbar von `Thread.MIN_PRIORITY` bis `Thread.MAX_PRIORITY`. Ein Thread wird defaultmäßig mit der Priorität des ihn anlegenden Prozesses ausgestattet. Default ist `Thread.NORM_PRIORITY`.

7.5 Synchronisation

Java garantiert, dass die meisten primitiven Operationen atomar (unteilbar) durchgeführt werden. Das sind z.B. Operationen, zu denen der Zugriff auf die Standardtypen (außer `long` und `double`) und auch die Referenztypen gehört (gemeint sind z.B. Zuweisungen an einen Referenztyp selbst).

Zum Schutz von "kritischen Bereichen" stellt Java ein Monitorkonzept zur Verfügung. Jedes Objekt mit kritischem Bereich bekommt zur Laufzeit einen Monitor. Kritische Bereiche sind eine oder mehrere Methoden oder Anweisungsblöcke, die von nicht mehr als einem Thread gleichzeitig durchlaufen werden dürfen.

Das Grundprinzip ist, dass individuelle Code-Blöcke über den Konstrukt

```
synchronized (irgendeinObjekt) { irgendeinCode }
```

synchronisiert werden können.

Eine `synchronized` Methode ist eine Methode, deren gesamte Implementation als ein `synchronized` Block

```
synchronized (this) { Methodencode }
```

zu verstehen ist. Das Monitor-Objekt ist das Objekt, wofür die Methode aufgerufen wird. Bei `static` Metoden wird das zur Klasse gehörige Klassenobjekt verwendet:

```
synchronized (this.getClass ()) { staticMethodencode }
```

Synchronisation ist so implementiert, dass für jedes Objekt (inklusive Klassenobjekten) ein nicht zugängliches Lock verwaltet wird, das im wesentlichen ein Zähler ist. Wenn der Zähler beim Betreten eines über das Objekt synchronisierten Blocks oder einer synchronisierten Methode *nicht* Null ist, dann wird der Thread blockiert (suspendiert). Er wird in eine Warteschlange für das Objekt gestellt ***(Entry Set)***, bis der Zähler Null ist. Wenn ein synchronisierter Bereich betreten wird, dann wird der Zähler um Eins erhöht. Wenn ein synchronisierter Bereich verlassen wird, dann wird der Zähler um Eins erniedrigt, (auch wenn der Bereich über eine Exception verlassen wird.)

Ein Thread, der ein Lock auf ein Objekt hat, kann mehrere kritische Bereiche oder Methoden desselben Objekts benutzen, wobei der Zähler beim Betreten des Bereichs hochgesetzt, beim Verlassen heruntergesetzt wird.

Nicht-synchronisierte Methoden oder Bereiche können von anderen Threads verwendet werden, auch wenn ein Thread ein Lock auf das Objekt hat. Insbesondere können alle Daten eines eventuell gelockten Objekts benutzt werden, sofern sie zugänglich (sichtbar) sind.

Der Modifikator `synchronized` wird nicht automatisch vererbt. Wenn also eine geerbte Methode auch in der abgeleiteten Klasse `synchronized` sein soll, so muss das angegeben werden. Bei Schnittstellen kann `synchronized` nicht angegeben werden, aber die Methoden können `synchronized` implementiert werden.

7.5.1 `synchronized`

Syntaktisch gibt es folgende Möglichkeiten:

1. `synchronized` als Modifikator einer Methode: Bevor die Methode von einem Thread für ein Objekt abgearbeitet wird, wird das Objekt von dem betreffenden Thread zum ausschließlichen Gebrauch von synchronisiertem Code des Objekts reserviert. Dabei wird der Thread eventuell so lange blockiert, bis der synchronisierte Code des Objekts nicht mehr von anderen Threads benötigt wird.

2. `synchronized (`*Ausdruck*`)` *Block* : Der Ausdruck muss in einem Objekt oder Feld resultieren. Bevor ein Thread den Block betritt, wird versucht, ein Lock auf das Resultat des Ausdrucks zum ausschließlichen Gebrauch synchronisierten Codes zu bekommen. Dabei wird der Thread eventuell so lange blockiert, bis der synchronisierte Code nicht mehr von anderen Threads benötigt wird.

3. `synchronized (`*Ausdruck*`.getClass ())` *Block* : Der Ausdruck muss in einem Objekt resultieren, dessen Klasse ermittelt wird. Damit werden `synchronized` Klassenmethoden geschützt. Bevor ein Thread den Block betritt, werden alle `synchronized` Klassenmethoden der betreffenden Klasse oder über die Klasse synchronisierte Code-Stücke zum ausschließlichen Gebrauch reserviert.

4. Konstruktoren können nicht `synchronized` definiert werden. Das ergibt einen Compilezeit-Fehler.

Mit `synchronized` werden Codestücke definiert, die *atomar* oder *unteilbar* bezüglich eines Objekts durchzuführen sind. Das Problem des gegenseitigen Ausschlusses ist damit schon gelöst und wesentlich eleganter gelöst als etwa mit Semaphoren.

Bemerkung: Das Problem der Prioritäts-Inversion – ein hochpriorer Thread läuft effektiv mit niedrigerer Priorität, wenn er versucht, ein Lock zu bekommen, das ein niedriger-priorer Thread hat – wird von vielen JVMs durch Prioritäts-Vererbung gelöst. Das ist aber nicht verpflichtend für eine JVM.

7.5.2 `wait`, `notify` und `notifyAll`

Die Methoden `wait ()`, `wait (long ms)`, `wait (long ms, int nanos)`, `notify ()` und `notifyAll ()` sind Methoden der Klasse `Object` und stehen daher für jedes Objekt zur Verfügung. Diese Methoden können nur in kritischen (`synchronized`) Bereichen des Objekts aufgerufen werden, d.h. wenn für das Objekt, wofür sie aufgerufen werden, ein Lock erhalten wurde (d.h. – wie man auch sagt – der Monitor des Objekts betreten wurde).

Wenn ein Thread für irgendein Objekt `x` die Methode `wait`

```
x.wait ();
```

aufruft, dann wird er in eine dem Objekt `x` zugeordnete `wait`-Warteschlange gestellt ***(Wait Set)***. Außerdem wird der kritische Bereich freigegeben, d.h. der Thread gibt sein Lock auf das Objekt `x` ab (d.h. der Monitor wird verlassen).

Wenn ein anderer Thread

```
x.notify ();
```

aufruft, dann wird *ein* Thread aus der `wait`-Warteschlange von `x` entfernt und wieder lauffähig. Dieser Thread muss sich aber nun um ein Lock für das Objekt `x` bemühen, um den kritischen Bereich weiter zu durchlaufen. Er muss also mindestens solange warten, bis der notifizierende Thread das Objekt freigegeben hat (d.h. den Monitor verlassen hat).
(Bemerkung: Mit der `Thread`-Instanzmethode `interrupt ()` kann ein spezifischer Thread aus der `wait`-Warteschlange geholt werden, wobei allerdings die `InterruptedException` geworfen wird.)

Wenn

```
x.notifyAll ();
```

aufgerufen wird, dann werden *alle* Threads aus der `wait`-Warteschlange von `x` entfernt und wieder lauffähig. Diese Threads müssen sich aber nun um ein Lock für das Objekt `x` bemühen, um den kritischen Bereich weiter zu durchlaufen. Ein Objekt mit kritischen Bereichen hat also konzeptionell *zwei* Warteschlangen:

- eine für den Monitor (d.h. die kritischen Bereiche) ***(Entry Set)***
- und eine für das `wait` ***(Wait Set)***.

Das Monitor-Konzept kann man auch unter folgenden Gesichtspunkten betrachten:

Ressourcen-Sicht:
Die Operationen

`"beantrage Ressource"` und `"gib Ressource frei"`
(bzw. `"beantrage Lock"` und `"gib Lock frei"`)
entsprechen in Java dem Paar:
`"synchronized (x) {"` und `"}"`

Nur der Thread, der ein Lock beantragt und erhalten hat, kann es auch wieder freigeben. Bekommt der Thread das Lock nicht, muss er warten (und nur dann).

"`x`" ist hierbei als Ressource zu verstehen.

Ereignis-Sicht:
Die Operationen

`"warte auf Ereignis"` und `"sende Ereignis"`
(bzw. `"warte auf Event"` und `"sende Event"`)
entsprechen in Java dem Paar:
`"x.wait ()"` und `"x.notify ()"` (oder `"x.notifyAll ()"`)

`"x"` ist dabei als Ereignis ***(Event)*** zu verstehen. Ein Thread, der auf ein Ereignis wartet, muss auf jeden Fall warten. Er kann das Ereignis nur von einem anderen Thread bekommen.

Bemerkung: In Java sind diese beiden Sichten insofern etwas vermischt, als man die Ressource `x` benötigt, um die Ereignis-Operationen `wait` oder `notify/All` durchführen zu können.

7.6 Beispiele

7.6.1 Erzeuger-Verbraucher-Problem

Beim Erzeuger-Verbraucher-Problem ***(producer-consumer-problem)*** generiert der Erzeuger ein Objekt (z.B. einen Satz von xyz-Koordinaten für die Position eines Roboterarms) und stellt es dem Verbraucher (z.B. dem Roboter) zur Verfügung, der das Objekt dann verarbeitet (z.B. sich an die angegebene Position bewegt). Es kommt hierbei darauf an, dass jedes erzeugte Objekt auch verbraucht wird und dass kein Objekt zweimal verbraucht wird.

Für die Übergabe des Objekts programmieren wir einen Puffer, der die Methoden

```
void  legeAb (Object x)
```
und
```
Object   entnehme ()
```

zur Verfügung stellt. Der Erzeuger soll `legeAb` für das erzeugte Objekt aufrufen. Der Verbraucher soll sich mit `entnehme` das Objekt holen. Diese Methoden sind so programmiert, dass, falls das alte Objekt noch nicht entnommen wurde, kein neues Objekt abgelegt werden kann und die Erzeuger-Task warten muss. Umgekehrt muss die Verbraucher-Task warten, wenn kein neues Objekt vorliegt.

```
public class   Puffer
   {
   private boolean   belegbar = true;
   private Object transferObjekt;
```

```
   public synchronized void   legeAb (Object x)
      {
      while (!belegbar)
         {
         try
            {
            wait ();
            }
            catch (InterruptedException e) { e.printStackTrace (); }
         }
      transferObjekt = x;
      belegbar = false;
      notify ();
      System.out.println (transferObjekt + " wurde abgelegt von "
         + Thread.currentThread ().getName () + ".");
      }
   public synchronized Object entnehme ()
      {
      while (belegbar)
         {
         try
            {
            wait ();
            }
            catch (InterruptedException e) { e.printStackTrace (); }
         }
      belegbar = true;
      notify ();
      System.out.println ("          " + transferObjekt
         + " wurde entnommen von "
            + Thread.currentThread ().getName () + ".");
      return transferObjekt;
      }
   }  // end class Puffer
```

```
public class   Erzeuger extends Thread
   {
   static java.util.Random zufall = new java.util.Random ();
   static class   Position
      // Objekte dieser Klasse werden transferiert.
      {
      int x = zufall.nextInt (9);
      int y = zufall.nextInt (9);
      int z = zufall.nextInt (9);
      public String  toString ()
         {
         String   s = super.toString ();
         s = s.substring (s.indexOf ('@'));
         return "Position (" + s
            + ")[x = " + x + "  y = " + y + "  z = " + z + "]";
         }
      }  // end class Erzeuger.Position
   public static void   warte (String botschaft)
      {
      System.out.println (botschaft);
      try
         {
         Thread.sleep (zufall.nextInt (500));
         }
         catch (InterruptedException e) { e.printStackTrace (); }
      }
   private Puffer puffer;
   public   Erzeuger (String name, Puffer puffer)
      {
      setName ("Erzeuger " + name);
```

```
      this.puffer = puffer;
      }
   public void run ()
      {
      while (true)
         {
         Object   x = new Position ();
         warte (x + " wurde erzeugt von " + getName () + ".");
         puffer.legeAb (x);
         }
      }
   }  // end class Erzeuger
```

```
public class   Verbraucher extends Thread
   {
   private Puffer puffer;
   public   Verbraucher (String name, Puffer puffer)
      {
      setName ("Verbraucher " + name);
      this.puffer = puffer;
      }
   public void run ()
      {
      while (true)
         {
         Object   x = puffer.entnehme ();
         Erzeuger.warte ("          " + x + " wird verbraucht von " + getName () + ".");
         }
      }
   }  // end class Verbraucher
```

```
public class   ErzeugerVerbraucher
   {
   public static void   main (String[] arg)
      {
      Puffer   puffer = new Puffer ();
      Erzeuger erzeuger = new Erzeuger ("Oskar", puffer);
      Verbraucher verbraucher = new Verbraucher ("Katrin", puffer);
      erzeuger.start ();
      verbraucher.start ();
      }
   }  // end class ErzeugerVerbraucher
```

Was passiert aber, wenn viele Erzeuger und Verbraucher denselben Puffer verwenden:

```
public class   VieleErzeugerVerbraucher
   {
   public static void   main (String[] arg)
      {
      Puffer   puffer = new Puffer ();
      for (int i = 0; i < 10; i++)
         {
         Erzeuger erzeuger = new Erzeuger ("" + (i + 1), puffer);
         Verbraucher verbraucher = new Verbraucher ("" + (i + 1), puffer);
         erzeuger.start ();
         verbraucher.start ();
         }
      }
```

```
} // end class VieleErzeugerVerbraucher
```

Wenn man das lang genug laufen lässt, dann kommt es zu einer Verklemmung, da eventuell alle Erzeuger in der Warteschlange sind, und das letzte `notify` eines Verbrauchers gerade wieder nur einen Verbraucher weckt.

Dieses Problem kann gelöst werden, indem man

```
notify ()
```

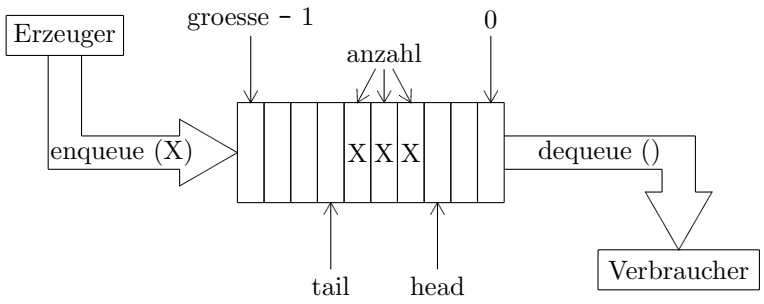
durch

```
notifyAll ()
```

ersetzt. Im Zweifelsfall ist es sicherer, `notifyAll` zu verwenden!

7.6.2 Beispiel: Ringpuffer

Eine allgemeinere Lösung des Erzeuger-Verbraucher-Problems bietet der Ringpuffer (Zirkularpuffer, Kanal, Pipe, Queue), der hier für Objekte implementiert ist. Durch einen Ringpuffer werden Erzeuger und Verbraucher zeitlich entkoppelt.



Bemerkung: Das Erzeuger-Verbraucher-Muster ist ein wichtiges nebenläufiges Entwurfs-Muster, etwa auch unter der Sicht, dass der Erzeuger Arbeitseinheiten definiert, die vom Verbraucher zu erledigen sind. Arbeiten werden zeitlich verschoben. Dadurch entstehen weniger Code-Abhängigkeiten. Ferner wird das Arbeits-Management *(workload management)* vereinfacht.

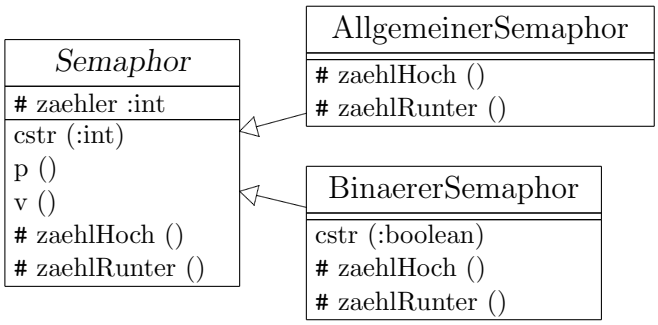
Die Klasse `RingPuffer` bietet zwei Methoden an: `enqueue (Object ob)` und `dequeue ()`.

```
import   java.util.*;
public class   RingPuffer
   {
   private Object[]  f;
   private int groesse;
   private int anzahl = 0;
```

```
   private int tail = 0;
   private int head = -1;
   public   RingPuffer (int groesse)
      {
      this.groesse = groesse;
      f = new Object[groesse];
      }
   synchronized public void   enqueue (Object ob)
       {
       while (anzahl >= groesse)
         try
            {
            wait ();
            }
            catch (InterruptedException e)
               {
               System.err.println ("Fehler in RingPuffer.enqueue:");
               System.err.println ("   " + e);
               }
       f[tail] = ob;
       anzahl = anzahl + 1;
       tail = tail + 1;
       if (tail == groesse) tail = 0;
       notifyAll ();
       }
   synchronized public Object dequeue ()
      {
      while (anzahl == 0)
         try
            {
            wait ();
            }
            catch (InterruptedException e)
               {
               System.err.println ("Fehler in RingPuffer.dequeue:");
               System.err.println ("   " + e);
               }
      anzahl = anzahl - 1;
      head = head + 1;
      if (head == groesse) head = 0;
      notifyAll ();
      return f[head];
      }
   public static void   main (String [] argument)
      {
      RingPuffer  rp = new RingPuffer (3);
      for (int i = 0; i < 5; i++)
         {
         new Erzeuger (rp, "E" + (i + 1), 7).start ();
         new Verbraucher (rp, "V" + (i + 1), 7).start ();
         }
      }
   Random   zufall = new Random ();
   }
class Erzeuger extends Thread
   {
   private RingPuffer   rp;
   private String name;
   private int wieoft;
   public   Erzeuger (RingPuffer rp, String name, int wieoft)
      {
      this.rp = rp;
      this.name = name;
      this.wieoft = wieoft;
      }
   public void run ()
      {
      for (int i = 0; i < wieoft; i++)
```

```
        {
        rp.enqueue (name + "." + (i + 1));
        System.out.println (name + ": " + (i + 1));
        try
           {
           sleep ((int)(rp.zufall.nextDouble () * 10000));
           }
           catch (InterruptedException e)
              {
              System.err.println ("Fehler in Erzeuger.run:");
              System.err.println ("   " + e);
              }
        }
     }
  }
class Verbraucher extends Thread
  {
  private RingPuffer   rp;
  private String name;
  private int wieoft;
  public   Verbraucher (RingPuffer rp, String name, int wieoft)
     {
     this.rp = rp;
     this.name = name;
     this.wieoft = wieoft;
     }
  public void run ()
     {
     for (int i = 0; i < wieoft; i++)
        {
        Object   s = rp.dequeue ();
        System.out.println (name + ": " + s);
        try
           {
           sleep ((int)(rp.zufall.nextDouble () * 10000));
           }
           catch (InterruptedException e)
              {
              System.err.println ("Fehler in Verbraucher.run:");
              System.err.println ("   " + e);
              }
        }
     }
  }
```

7.6.3 Beispiel: Emulation von Semaphoren



```
public abstract class   Semaphor
   {
   public   Semaphor (int zaehler)
      {
      if (zaehler < 0) this.zaehler = 0;
      else this.zaehler = zaehler;
      }
   public synchronized void   p ()
      {
      while (zaehler == 0)
         {
         try { wait (); }
         catch (InterruptedException e) {}
         }
      zaehlRunter ();
      }
   public synchronized void   v ()
      {
      zaehlHoch ();
      notifyAll ();  // notify () würde hier auch genügen.
      }
   protected int   zaehler;
   protected abstract void  zaehlHoch ();
   protected abstract void  zaehlRunter ();
   }
```

```
public class   AllgemeinerSemaphor extends Semaphor
   {
   public   AllgemeinerSemaphor (int zaehler)
      {
      super (zaehler);
      }
   protected void  zaehlHoch ()
      {
      zaehler++;
      }
   protected void  zaehlRunter ()
      {
      zaehler--;
      if (zaehler < 0) zaehler = 0;
      }
   }
```

```
public class   BinaererSemaphor extends Semaphor
   {
   public   BinaererSemaphor (boolean frei)
      {
      super (frei ? 1 : 0);
      }
   protected void  zaehlHoch ()
      {
      zaehler = 1;
      }
   protected void  zaehlRunter ()
      {
      zaehler = 0;
      }
   }
```

Als Anwendung schreiben wir ein primitives Programm zum automatischen Schweißen mit einem Roboterarm. Folgende Aufgaben sind hintereinander durchzuführen:

uB: (unkritische) Berechnung der nächstens Armposition.

kP: (kritische) Positionierung des Arms.

uV: (unkritische) Vorbereitung des Schweißwerkzeugs.

kS: (kritisches) Schweißen.

uB kann parallel zu uV und kS durchgeführt werden. uV kann parallel zu uB und kP durchgeführt werden. kP und kS dürfen nicht gleichzeitig und kP muss vor kS durchgeführt werden. Um möglichst effektiv zu arbeiten, wollen wir alle parallel durchführbaren Aktivitäten auch parallel durchführen.

Das Programm dazu ist:

```
import java.util.*;
public class   SchweissRoboter
   {
   BinaererSemaphor  s1 = new BinaererSemaphor (true);
   BinaererSemaphor  s2 = new BinaererSemaphor (false);
   public static void   main (String[] argument)
      {
      SchweissRoboter   r = new SchweissRoboter ();
      new ArmThread (r).start ();
      new SchweissThread (r).start ();
      }
   public void uB (int n) { operation ("uB" + n); }
   public void kP (int n) { operation ("kP" + n); }
   public void uV (int n) { operation ("uV" + n); }
   public void kS (int n) { operation ("kS" + n); }
   private static Random   rand = new Random ();
   private static void  warte (Random rand)
      {
      try
         {
         Thread.sleep ((long)(Math.abs (rand.nextInt ()) % 100));
         }
         catch (InterruptedException e) {}
      }
   private void   operation (String opName)
      // Simuliert die Operationen des Roboters durch Bildschirmausgaben
      // mit unterschiedlich vielen Zeilen.
      {
      if (Thread.currentThread ().getName ().equals ("SchweissThread"))
         opName = "                  " + opName;
      int   j = rand.nextInt ();
      j = j < 0 ? -j : j;
      j = j % 4 + 1;
      System.out.println (opName + " Anfang"); warte (rand);
      for (int i = 0; i < j; i++)
         {
         System.out.println (opName + " ......"); warte (rand);
         }
      System.out.println (opName + " Ende. "); warte (rand);
      }
```

```
   }
class ArmThread extends Thread
   {
   private SchweissRoboter r;
   public   ArmThread (SchweissRoboter r)
      {
      this.r= r;
      setName ("ArmThread");
      }
   public void run ()
      {
      int   n = 0;
      while (true)
         {
         n++;
         r.uB (n);
         r.s1.p ();
         r.kP (n);
         r.s2.v ();
         }
      }
   }
class SchweissThread extends Thread
   {
   private SchweissRoboter r;
   public   SchweissThread (SchweissRoboter r)
      {
      this.r= r;
      setName ("SchweissThread");
      }
   public void run ()
      {
      int   n = 0;
      while (true)
         {
         n++;
         r.uV (n);
         r.s2.p ();
         r.kS (n);
         r.s1.v ();
         }
      }
   }
```

7.7 Dämonen

Der Java-Interpreter wird erst verlassen, wenn alle Threads aufgehört haben zu laufen. Allerdings nimmt der Java-Interpreter dabei keine Rücksicht auf sogenannte **Dämonen**. Ein Dämon ***(daemon)*** ist ein Thread, der immer im Hintergrund läuft und nie aufhört zu laufen. Der Java-Interpreter wird also verlassen, wenn nur noch Dämonen laufen.

Mit der Methode `setDaemon (true)` wird ein Thread zum Daemon gemacht. Die Methode kann nur *vor dem Start* eines Threads aufgerufen werden.

Mit `isDaemon ()` kann festgestellt werden, ob ein Thread ein Dämon ist.

Alle Threads, die von einem Dämon erzeugt werden, sind ebenfalls Dämonen.

Dämonen sollten sparsam verwendet werden, da sie ja, ohne die Möglichkeit von Aufräumarbeiten, rücksichtslos bei Abschalten der JVM abgebrochen werden. Z.B. kann ein Log-Service-Dämon die letzten Nachrichten evtl. nicht mehr in einen File schreiben.

7.8 Übungen

7.8.1 `start/run`-Methode

Erklären Sie den Unterschied zwischen

```
MeinThread1 p = new MeinThread1 ();
p.start ();
```

und

```
MeinThread1 p = new MeinThread1 ();
p.run ();
```

7.8.2 Threads

Verändern Sie das Herr-Knecht-Beispiel so, dass das Hauptprogramm den Knecht nach 6 Sekunden stoppt. Der Herr soll seine while-Schleife nur 1000 mal durchlaufen. Das Hauptprogramm soll dann warten, bis der Herr zu Ende gelaufen ist.

7.8.3 Klasse `GroupTree`

Schreiben Sie eine Klasse `GroupTree`, deren `static` Methode `dumpAll ()` alle Threads und ihre Threadgruppen listet. (Hinweis: Die Klasse `ThreadGroup` bietet hier geeignete Methoden an.)

7.8.4 Sanduhr

In dem unten angegebenen Beispiel dauert die Abarbeitung der Methode `dauertLange ()` etwa 5 bis 10 Sekunden. Aufgabe ist es, diese Zeit zu überbrücken, indem jede halbe Sekunde ein Punkt nach Standard-Out geschrieben wird.

```
public class   Sanduhr
   {
   public static void   main (String[] arg)
      {
      Sanduhr  su  = new Sanduhr ();
      su.dauertLange ();
      }
```

```
public void dauertLange ()
   {
   System.out.println ("Achtung! Das dauert jetzt ein bisschen.");
   try
      {
      Thread.sleep ((long)(Math.random ()*5000 + 5000));
      }
      catch (InterruptedException e) {}
   System.out.println ("Endlich! Es ist vorbei.");
   }
}
```

7.8.5 Konten-Schieberei

1. Schreiben Sie eine ganz primitive Klasse `Konto`, die nur einen Kontostand verwaltet und einfache `get`-und `set`-Methoden zur Verfügung stellt. Der Konstruktor soll einen Anfangs-Kontostand übernehmen.

2. Schreiben Sie ein Klasse `Transaktion`, die als Thread laufen kann. Der Konstruktor übernimmt ein Feld von Konten, einen Betrag und eine Anzahl Transaktionen, die durchzuführen sind.

   Der eigentliche Thread-Code soll folgendes tun: Aus dem Feld von Konten sollen zwei Konten zufällig herausgesucht werden. Und es soll der im Konstruktor angegebene Betrag von dem einen zum anderen Konto übertragen werden. Das soll so oft passieren, wie im Konstruktor angegeben wurde. Damit das gewünschte Ergebnis erreicht wird, muss an vielen Stellen eine zufällige Anzahl von `yield ()` eingestreut werden.

3. Schreiben Sie eine Klasse `KontenSchieberei`, die nur aus einem `main`-Programm besteht, das folgendes tut:

   (a) Ein Feld von etwa 10 Konten wird angelegt.

   (b) Die Summe aller Konten wird bestimmt und ausgegeben.

   (c) Es werden jetzt etwa 100 Transaktions-Objekte angelegt und gestartet.

   (d) Es wird gewartet, bis alle Transaktionen zu Ende gelaufen sind.

   (e) Die Summe aller Konten wird bestimmt und ausgegeben.

4. Als Ergebnis wird erwartet, dass die Transaktionen sich gegenseitig stören und daher die Gesamtsumme nicht mehr stimmt. Wie kann man das reparieren?

7.8.6 Join-Beispiel

Im folgenden Beispiel wird ein Feld `feld` willkürlich mit `true` oder `false` belegt.

Der unten programmierte Thread `Zaehler` bestimmt in einem Bereich des Feldes von `anfang` bis `ende` die Anzahl der wahren und falschen Werte.

Es sollen nun mehrere Threads angelegt werden, die jeweils für einen Teil des Feldes für das Zählen verantwortlich sind. Das ganze Feld soll abgedeckt werden. Die Threads sollen gestartet werden. Das Hauptprogramm soll warten, bis alle Threads zu Ende gekommen sind, und dann das Resultat ausgeben.

```
import   java.util.*;
public class   JoinBeispiel
   {
   public static void   main (String[] arg)
      {
      int   groesse = 10000;
      boolean[]   feld = new boolean[groesse];
      for (int i = 0; i < groesse; i++)
         {
         feld[i] = Math.random () > 0.5 ? true : false;
         }
      ArrayList<Zaehler>   th = new ArrayList<Zaehler> ();
      // Aufgabe:
      // ...
      // Threads anlegen und in ArrayList th verwalten.
      // ...
      // Threads starten.
      // ...
      // Auf das Ende der Threads warten.
      // Ende der Aufgabe.
      int   tresultat = 0;
      int   fresultat = 0;
      for (Zaehler t :th)
         {
         tresultat = tresultat + t.tresultat;
         fresultat = fresultat + t.fresultat;
         }
      System.out.println ("Von " + groesse + " Werten sind " + tresultat
         + " true.");
      System.out.println ("Von " + groesse + " Werten sind " + fresultat
         + " false.");
      System.out.println ("Kontrolle: " + groesse + " = " + tresultat
         + " + " + fresultat);
      System.out.println ();
      }
   }
class Zaehler
   extends  Thread
   {
   boolean[]   feld;
   int   anfang;
   int   ende;
   int   tresultat;
   int   fresultat;
   public   Zaehler (boolean[] feld, int anfang, int ende)
      {
      this.feld = feld;
      this.anfang = anfang;
      this.ende = ende;
      }
   public void run ()
      {
      tresultat = 0;
      fresultat = 0;
      for (int i = anfang; i <= ende; i++)
         {
         if (feld[i]) tresultat = tresultat + 1;
         else fresultat = fresultat + 1;
         try
            {
            Thread.sleep ((long) (Math.random () * 10));
            }
            catch (InterruptedException e) { e.printStackTrace (); }
         }
      }
   }
```

7.8.7 Witz

In einer Stadt von n Personen erfindet eine Person einen Witz und erzählt ihn anderen Personen, die ihn nach folgenden Regeln weitererzählen:

1. Die Personen treffen zufällig aufeinander.
2. Wird der Witz einer Person erzählt, die ihn noch nicht kennt, dann erzählt auch sie den Witz weiter.
3. Wird der Witz einer Person erzählt, die ihn schon kennt, dann hören beide auf, den Witz weiterzuerzählen.

Wieviel Personen erfahren von dem Witz? Lösen Sie das Problem durch Simulation, indem Sie für jede Person ein Objekt anlegen, das als Witz-verbreitender Thread laufen kann.

7.8.8 Schweiß-Roboter mit Semaphoren

Bringen Sie das folgende Schweiss-Roboter-Programm zum Laufen, indem Sie eine Semaphor-Klasse erstellen und die beiden Threads für das Positionieren und Schweißen erstellen.

```
import java.util.*;
public class   SchweissRoboter
   {
   public static void   main (String[] argument)
      {
      SchweissRoboter   r = new SchweissRoboter ();
      new ArmThread (r).start ();
      new SchweissThread (r).start ();
      }
   public void uB (int n) { operation ("uB" + n); }
   public void kP (int n) { operation ("kP" + n); }
   public void uV (int n) { operation ("uV" + n); }
   public void kS (int n) { operation ("kS" + n); }
   private static Random   rand = new Random ();
   private void   operation (String opName)
      // Simuliert die Operationen des Roboters durch Bildschirmausgaben
      // mit unterschiedlich vielen Zeilen.
      {
      if (Thread.currentThread ().getName ().equals ("SchweissThread"))
         opName = "                  " + opName;
      int   j = rand.nextInt ();
      j = j < 0 ? -j : j;
      j = j % 4 + 1;
      System.out.println (opName + " Anfang"); Thread.yield ();
      for (int i = 0; i < j; i++)
         {
         System.out.println (opName + " ......"); Thread.yield ();
         }
      System.out.println (opName + " Ende. "); Thread.yield ();
      }
   }
```

7.8.9 Schweiß-Roboter ohne Semaphore

Programmieren Sie dieses Beispiel unter Verwendung des Monitor-Konzepts von Java, d.h. ohne Verwendung von Semaphoren.

Hinweis: Schreiben Sie für die kritischen Roboter-Operationen eigene Methoden (in der Klasse `SchweissRoboter`), wobei diese Methoden `synchronized` sind und mit geigneten `wait` und `notify` zu versehen sind. Mit einem Boolean wird verwaltet, welche Operation gerade dran ist.

7.8.10 Timeout für Ringpuffer-Methoden

1. Schreiben Sie für die Methoden `enqueue (...)` und `dequeue ()` überladene Versionen, bei denen man ein Timeout angeben kann.
2. Priorisierter Zugriff auf Ringpuffer: Die Methoden `enqueue (...)` und `dequeue ()` sollen bezüglich einander priorisierbar sein. D.h. wenn `enqueue (...)` Priorität über `dequeue ()` hat, dann wird garantiert, dass bei durch gegenseitigen Ausschluss blockierten Tasks, eine `enqueue`-Task Vorrang hat.

7.8.11 Problem der speisenden Philosophen

Ein berühmtes Synchronisations-Problem ist das Problem der n Philosophen, die an einem runden Tisch sitzen und zyklisch immer wieder denken und speisen. Für jeden Philosophen ist *eine* Gabel ausgelegt. Um aber speisen zu können, benötigt ein Philosoph auch die Gabel seines rechten Nachbarn. Das bedeutet, dass nicht alle Philosophen gleichzeitig speisen können. (Die Philosphen wechseln nicht ihre Plätze.) Es gilt nun zu verhindern, dass zwei Philosophen dieselbe Gabel greifen. Natürlich darf auch keine Verklemmung auftreten.

1. Lösen Sie dieses Problem mit dem Monitor-Konzept von Java.
2. Lösen Sie dieses Problem mit einfachen Semaphoren.
3. Lösen Sie dieses Problem mit Semaphorgruppen.

7.8.12 Stoppuhr

Entwickeln Sie eine Stoppuhr.

7.8.13 Barkeeper

Der Barkeeper schenkt an die Kunden Whiskey oder Wodka aus. Wenn er gerade die Whiskeyflasche in der Hand hat, dann schenkt er Wiskey aus, solange es Kunden gibt, die Whiskey möchten. Wenn niemand mehr Whiskey möchte, dann wartet er noch 2 Sekunden, ob nicht doch noch einer Whiskey möchte. Wenn das der Fall ist, wird auch noch der Whiskeykunde bedient und eventuell noch weitere plötzlich aufgetauchte Whiskeykunden.

Ebenso verhält es sich mit dem Wodka.

Wenn niemand mehr Whiskey oder Wodka möchte, dann spült er ein paar Gläser. Danach schaut er wieder, ob es Kunden gibt.

Jedesmal, wenn er eingeschenkt hat oder Gläser gespült hat, atmet er tief durch.

Programmieren Sie die unten zur Verfügung gestellte Klasse `Barkeeper` so, dass er sich wie oben beschrieben verhält.

Die Klasse `Kunde` sollte dabei nicht verändert werden.

```
import   java.util.*;
public class   BarkeeperHaupt
   // Implementation über Algorithmic State Machine
   {
   public static void   main (String[] argument)
      {
      Barkeeper   bk = new Barkeeper ("Bartender");
      Kunde k1 = new Kunde ("Boris", "          ", bk);
      Kunde k2 = new Kunde ("Fedor", "                    ", bk);
      Kunde k3 = new Kunde ("Ivano", "                              ", bk);
      }
   public static void   jemandTutWas (String offset, String jemand, String was)
      {
      pausiere ();
      System.out.println (offset + jemand + " " + was);
      }
   private static java.util.Random  r = new java.util.Random ();
   public static int zufall () { return BarkeeperHaupt.zufall (6); }
   public static int zufall (int n)
      {
      return Math.abs (r.nextInt ()) % n + 1;
      }
   public static void   pausiere ()
      {
      try
         {
         Thread.sleep (BarkeeperHaupt.zufall (200));
         }
         catch (InterruptedException e) { }
      }
   }  // end BarkeeperHaupt

class Barkeeper
   {
   private String name;
   public   Barkeeper (String name)
      {
      this.name = name;
      }
   public void whiskey (String name, String offset)
      {
      BarkeeperHaupt.jemandTutWas ("", this.name, "nimmt Whiskeyflasche.");
      BarkeeperHaupt.jemandTutWas ("", this.name, "schenkt " + name + " Whiskey ein.");
      BarkeeperHaupt.jemandTutWas (offset, name, "bekommt Whiskey.");
      BarkeeperHaupt.jemandTutWas (offset, name, "hört Prost Whiskey.");
      BarkeeperHaupt.jemandTutWas ("", this.name, "schaut nach Whiskeykunden.");
      BarkeeperHaupt.jemandTutWas ("", this.name, "stellt Flasche Whiskeyflasche ab.");
      BarkeeperHaupt.jemandTutWas ("", this.name, "spült paar Gläser.");
      BarkeeperHaupt.jemandTutWas ("", this.name, "atmet tief durch.");
      }
   public void wodka (String name, String offset)
      {
      BarkeeperHaupt.jemandTutWas ("", this.name, "nimmt Wodkaflasche.");
      BarkeeperHaupt.jemandTutWas ("", this.name, "schenkt " + name + " Wodka ein.");
```

```
      BarkeeperHaupt.jemandTutWas (offset, name, "bekommt Wodka.");
      BarkeeperHaupt.jemandTutWas (offset, name, "hört Prost Wodka.");
      BarkeeperHaupt.jemandTutWas ("", this.name, "schaut nach Wodkakunden.");
      BarkeeperHaupt.jemandTutWas ("", this.name, "stellt Flasche Wodkaflasche ab.");
      BarkeeperHaupt.jemandTutWas ("", this.name, "spült paar Gläser.");
      BarkeeperHaupt.jemandTutWas ("", this.name, "atmet tief durch.");
      }

   }  // end Barkeeper

class Kunde extends Thread
   {
   private String name;
   private String offset;
   private Barkeeper bk;
   public   Kunde (String name, String offset, Barkeeper bk)
      {
      this.name = name;
      this.offset = offset;
      this.bk = bk;
      this.start ();
      }
   int n = 0;
   public void run ()
      {
      java.util.Random  r = new java.util.Random ();
      BarkeeperHaupt.jemandTutWas (offset, name, "kommt.");
      while (true)
         {
         n++;
         for (int i = 0; i < BarkeeperHaupt.zufall (3); i++)
            {
            BarkeeperHaupt.jemandTutWas (offset, name + n, "wählt.");
            }
         switch (BarkeeperHaupt.zufall (4))
            {
            case 1:
               BarkeeperHaupt.jemandTutWas (offset, name + n, "bestellt Whiskey.");
               bk.whiskey (name + n, offset);
               break;
            case 2:
               BarkeeperHaupt.jemandTutWas (offset, name + n, "bestellt Wodka.");
               bk.wodka (name + n, offset);
               break;
            case 3:
               BarkeeperHaupt.jemandTutWas (offset, name + n, "bestellt Wodka und Whiskey.");
               Thread   t1 = new Thread ()
                  {
                  public void run ()
                     {
                     bk.wodka (name + n, offset);
                     }
                  };
               t1.start ();
               bk.whiskey (name + n, offset);
               try { t1.join (); } catch (InterruptedException e) {}
               break;
            case 4:
               BarkeeperHaupt.jemandTutWas (offset, name + n, "bestellt Whiskey und Wodka.");
               Thread   t2 = new Thread ()
                  {
                  public void run ()
                     {
                     bk.whiskey (name + n, offset);
                     }
                  };
               t2.start ();
               bk.wodka (name + n, offset);
               try { t2.join (); } catch (InterruptedException e) {}
               break;
            }
```

```
        BarkeeperHaupt.jemandTutWas (offset, name + n, "hat genug.");
        BarkeeperHaupt.jemandTutWas (offset, name + n, "geht.");
        BarkeeperHaupt.jemandTutWas (offset, name + n, "kommt wieder.");
        }
    }
}
```

Kapitel 8

Thread-Design

In diesem Kapitel werden Regeln für den Entwurf von nebenläufigen Programmen mit Tasks, insbesondere Java-Threads vorgestellt. Diese Regeln lassen sich ohne Weiteres auf andere nebenläufige Systeme verallgemeinern. Z.B. kann alles, was über die Verwendung von `synchronized` gesagt wird, auf alle Probleme des gegenseitigen Ausschlusses angewendet werden, und damit z.B. auf MUTEXe.

8.1 Thread Safety

Thread-Sicherheit ***(thread safety)*** bedeutet, dass die Datenelemente eines Objekts oder einer Klasse – dort sind es die Klassenvariablen – immer einen korrekten oder konsistenten Zustand aus der Sicht anderer Objekte oder Klassen haben, auch wenn von mehreren Threads gleichzeitig ***(shared)*** zugegriffen wird. (Lokale Variablen, Methodenparameter und Rückgabewerte sind davon nicht betroffen.)

Da Nebenläufigkeit ein wichtiges Merkmal von Java ist, muss man beim Design von Klassen grundsätzlich immer an Thread-Safety denken, da prinzipiell jedes Objekt oder jede Klasse in einer nebenläufigen Umgebung verwendet werden kann.

Thread-Safety kann folgendermaßen erreicht werden:

1. Kritische Bereiche erhalten ein `synchronized`.

   Alle Datenelemente, die inkonsistent werden können, werden `private` gemacht und erhalten eventuell `synchronized` Zugriffsmethoden. Ferner muss darauf geachtet werden, dass die Datenelemente nur über die Zugriffsmethoden verwendet werden. Für Konstanten gilt das nicht. ("inkonsistent werden können" bedeutet "durch mehr als einen Thread gleichzeitig veränderbar sein", ***mutable state variables***)

   `synchronized` ist notwendig, wenn mehrere Threads auf gemeinsame Daten (Objekte) zugreifen und mindestens ein Thread diese Daten (Objekte) verändert. Alle Methoden, die auf die Daten (Objekte) zugreifen, müssen als `synchronized` deklariert werden unabhängig davon, ob nur gelesen oder auch verändert wird.

Methoden-Aufrufe anderer Objekte sollten **nicht** in einer `synchronized` Umgebung erfolgen, weil das die Gefahr einer Verklemmung in sich birgt.

Der Zugriff auf einzelne Datenelemente vom Typ `boolean`, `char`, `byte`, `short`, `int` oder `float` ist unter Java Thread-sicher (nicht aber `long` und `double`).

2. Die Datenelemente werden unveränderbar (`final`) gemacht ***(immutable objects)***.

   Will man solche Objekte dennoch "ändern", dann muss man eventuell veränderte Kopien von ihnen anlegen.

3. Man benutzt einen Thread-sicheren Wrapper.

Warum kann man nicht einfach alles `synchronized` machen?

1. Synchronisierte Methoden-Aufrufe sind etwa 5 mal langsamer als nicht-synchronisierte Aufrufe.
2. Unnötige Synchronisationen (etwa bei Lesezugriffen) haben überflüssiges Blockieren von Threads zur Folge.
3. Immer besteht die Gefahr eines Deadlocks.
4. Auch wenn der einzelne Methodenaufruf Thread-sicher ist (`synchronized`), dann ist eine Kombination solcher Aufrufe trotzdem nicht Thread-sicher. Z.B. sind alle Methoden in der Klasse `Vector` `synchronized`, aber folgende Code-Folge ist nicht Thread-sicher:

   ```
   if (!vector.contains (element))
      {
      vector.add (element);
      }
   ```

   Um sie Thread-sicher zu machen, muss der ganze Block synchronisiert werden:

   ```
   synchronized (vector)
      {
      if (!vector.contains (element))
         {
         vector.add (element);
         }
      }
   ```

Die Verwendung von `synchronized` Blöcken hat den Nebeneffekt, dass Variablen, die eventuell im Cache waren, für andere Threads **sichtbar** werden. Wenn es allerdings nur darum geht, Variable für andere Threads sichtbar zu machen ***(sequential consistency)*** und nicht um Atomizität, dann kann – als mildere Form der Synchronisation – die Variable als `volatile` deklariert werden. Der Zugriff auf eine `volatile` Variable macht zu dem Zeitpunkt dann auch alle anderen Variablen für alle anderen Threads sichtbar.

Wenn Datenelemente nicht verändert werden ***(immutable)***, dann sind sie nach der vollständigen Objektkonstruktion Thread-sicher. Das kann auch noch für den Vorgang der Objektkonstruktion garantiert werden, wenn die Datenelemente `final` deklariert werden. Ferner kann auch kein Code erstellt werden, der diese Datenelemente verändert. Regel:

Alle Datenelemente sollten `final` deklariert werden, es sei denn, dass sie veränderlich sein sollen.

Bemerkung: Wenn Referenzen auf Objekte `final` deklariert werden, bedeutet das nicht, dass die referenzierten Objekte nicht verändert werden können, sondern nur, dass die `final` Referenz nicht ein anderes Objekt referenzieren kann.

8.2 Regeln zu `wait`, `notify` und `notifyAll`

8.2.1 Verwendung von `wait`, `notify` und `notifyAll`

Wenn durch einen Methodenaufruf der Zustand eines Objekts verändert wird **und** diese Änderung nur erfolgen darf, wenn eine gewisse **Wartebedingung** nicht mehr gegeben ist, dann hat diese Methode folgende Form:

```
synchronized void methode ()
   {
   while (Wartebedingung)
      {
      wait ();
      }

   // Mach Zustandsänderung.
   }
```

Das "`while`" ist notwendig, da nach Entlassung aus dem Wait-Set der Thread sich erst um das Lock bemühen muss, d.h. eventuell gewisse Zeit im Entry-Set verbringt, währenddessen ein anderer Thread die Wartebedingung wieder wahr machen könnte. Der zweite Grund sind spurious Wakeups (siehe unten).

Wenn eine Zustandsänderung dazu führt, dass eventuell wartende Threads weiterlaufen können, muss ein `notify` oder allgemeiner ein `notifyAll` aufgerufen werden:

```
synchronized void methode ()
   {
   // Mach Zustandsänderung.

   notifyAll ();
   }
```

Natürlich kann es sein, dass in einer Methode beide Fälle auftreten:

```
synchronized void methode ()
   {
   while (Wartebedingung)
      {
      wait ();
      }

   // Mach Zustandsänderung.

   notifyAll ();
   }
```

Nach **rein lesenden** Zugriffen muss grundsätzlich kein Thread geweckt werden (also **kein** `notify` oder `notifyAll`).

8.2.2 *Spurious Wakeup*

Es kann sein, dass ein Thread "aufwacht", ohne dass ein Timeout abgelaufen ist oder ein `notify`, `notifyAll` oder `interrupt` aufgerufen wurde. Dagegen muss man sich schützen. Das erreicht man i.A. dadurch, dass man die betroffenen `wait`s in eine `while`-Schleife bezüglich der Wartebedingung stellt (siehe Regel oben).

Spurious Wakeups kommen vor, weil z.B. das Multitasking des Host-Betriebssystems verwendet wird.

8.2.3 Timeout

Timeout "0" bedeutet bei `wait (timeout)` "kein Timeout", also unendlich langes Warten.

Das merkwürdige an der `wait (timeout)`-Methode ist, dass man nicht unterscheiden kann, ob der Timeout-Fall eingetreten ist, oder ob ein `notify` aufgerufen wurde. Man wird gezwungen ein Flag zu verwalten, das gesetzt wird, wenn ein `notify` aufgerufen wird, was nicht ganz trivial ist (wegen *spurious wakeup*).

8.2.4 Zusammenfassung `wait ()`, `notify ()`, `notifyAll ()`

Wir fassen die Verwendung von `wait ()`, `notify ()` und `notifyAll ()` in folgenden Regeln zusammen:

1. `wait`, `notify` und `notifyAll` können nur in einem Block aufgerufen werden, der für das aufrufende Objekt `synchronized` ist. Der Block kann natürlich auch eine `synchronized` Methode sein. (Die folgende Syntax bezieht sich auf diesen Fall.) Ferner verwenden wir **Sync** als Abkürzung für einen `synchronized` Block oder eine `synchronized` Methode.
2. `wait` gehört in eine While-Schleife:
   `while (Wartebedingung) wait ();`

3. Ein Sync muss ein `notify` oder `notifyAll` haben, wenn es mindestens ein `wait` gibt **und** wenn das Sync die `Wartebedingung` eines `wait`s möglicherweise `false` macht. (Ein Sync, das eine `Wartebedingung true` machen kann, braucht **kein** `notify` oder `notifyAll`.)

4. Ein `notify` genügt, wenn die Anwendung so ist, dass höchstens **ein** Thread die `wait`-Warteschlange verlassen soll und dass dieser Thread dann auch der "richtige" (der "gemeinte") ist.

5. In allen anderen Fällen muss ein `notifyAll` aufgerufen werden.

Wenn alle `wait`s in `while`-Schleifen verpackt sind, dann kann man immer ein `notifyAll` anstatt eines `notify` verwenden, ohne die Korrektheit des Programms zu beeinträchtigen. Auch überflüssige `notifyAll ()` schaden dann nicht. Das ist höchstens Performanz-schädlich.

8.3 `notify` oder `interrupt` ?

Mit `x.notify ()` wird **irgendein** Thread aus der `wait`-Warteschlange für `x` freigegeben.

Mit `t.interrupt ()` wird **genau** der Thread `t` aus einer `wait`-Warteschlange freigegeben.

`x.notify ()` kann nur in einer bezüglich `x synchronized` Umgebung aufgerufen werden. Abgesehen davon, dass man einen Monitor-Kontext braucht, ist `notify` performanter als `interrupt`, da keine Exception geworfen wird.

`notify` "verpufft" wirkungslos, wenn sich *kein* Thread in der `wait`-Warteschlange befindet.

`t.interrupt` weckt einen Thread `t` auch aus einem `sleep` oder `join`.

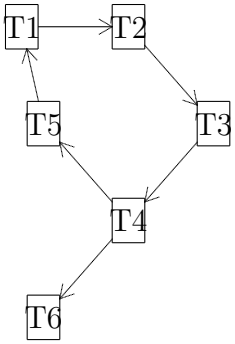
Bei `t.interrupt` wird eine `InterruptedException` geworfen.

`interrupt` setzt ein Interrupted-Flag, das dann bei Anlaufen eines `wait`, `sleep` oder `join` die `InterruptedException` auslöst, wobei dann das Flag zurückgesetzt wird. Ein `interrupt` "verpufft" also nicht. Mit `Thread.interrupted ()` kann das Flag vor dem Anlaufen eines `wait`, `sleep` oder `join` zurückgesetzt werden, um das Werfen einer Exception zu verhindern.

8.4 Verklemmungen

Vermeidung von **Verklemmungen**: Verklemmungen entstehen, wenn von verschiedenen Threads mehrere Betriebsmittel unter gegenseitigem Ausschluss angefordert werden. Solche Verklemmungen können detektiert werden, indem man einen **Vorranggraphen** erstellt, der aus Symbolen für Threads und Pfeilen besteht. Wenn ein Thread T_2 auf die Freigabe eines Betriebsmittels wartet, das ein anderer Thread T_1 benutzt, dann wird ein Pfeil $T_1 \longrightarrow T_2$ gezogen. Kommt es dabei zu einem Zyklus, dann liegt eine Verklemmung vor.

Achtung: Oft wird die Pfeilrichtung auch andersherum verwendet. Wie sie hier verwendet wird, erscheint uns am vernünftigsten und ist auch kompatibel mit anderen, ähnlichen Diagrammen, z.B. UML-Aktivitäts-Diagrammen.



Verklemmungen können folgendermaßen vermieden werden:

1. **Voranforderungsstrategie:** Ein Thread fordert alle Betriebsmittel, die er gerade benötigt, auf einen Schlag an.

   Oder: Solange ein Thread ein Betriebsmittel hat, darf er keine weiteren Betriebsmittel anfordern. Bezüglich des Vorranggraphen bedeutet das, dass ein Thread entweder nur eingehende oder nur ausgehende Pfeile hat. Damit kann es zu keinem Zyklus kommen.

   Diese Strategie vermindert – eventuell drastisch – die Nebenläufigkeit.

2. **Anforderung nach einer vorgegebenen Reihenfolge:** Mehrere Betriebsmittel dürfen nur in einer vorgegebenen Reihenfolge angefordert werden.

   Jedes Betriebsmittel könnte eine Nummer haben, nach der die Betriebsmittel – etwa aufsteigend – geordnet sind. Wenn ein Thread ein Betriebsmittel hat, dann kann er nur Betriebsmittel mit höherer Nummer anfordern. Das vermeidet Verklemmungen. Denn nehmen wir an, es gäbe im Vorranggraphen einen Zyklus. Wenn wir entlang des Zyklus gehen, dann muss die Nummer des jeweils angeforderten Betriebsmittels steigen. Damit ist aber kein Zyklus möglich.

   Das kann man i.A. nur über einen Betriebsmittel-Manager implementieren, der weiß, in welcher Reihenfolge die Betriebsmittel angefordert werden dürfen.

3. **Anforderung von Betriebsmitteln mit Bedarfsanalyse:** Jeder Thread gibt einem Betriebsmittel-Manager von vornherein bekannt, welche und wieviele Betriebsmittel er eventuell gleichzeitig benötigt. Der Manager teilt die Betriebsmittel nur dann zu, wenn dabei der **Zustand der Betriebsmittelbelegungen (Belegungszustand) sicher** bleibt. "Sicher" heißt, es kann zu keiner Verklemmung kommen. Oder genauer: Es gibt eine Folge von Belegungszuständen, so dass jeder Thread seine Restforderungen erfüllt bekommt, damit er zu Ende laufen kann und alle belegten Betriebsmittel schließlich freigibt.

   Wegen Einzelheiten des sogenannten **(Bankier-Algorithmus)** sei auf die Literatur [17] verwiesen.

Bei Datenbanksystemen nimmt man Verklemmungen von Transaktionen durchaus in Kauf, um möglichst hohe Nebenläufigkeit zu erreichen. Im Verklemmungsfall wird eben eine Transaktion zurückgesetzt (Rollback). Diese kann dann zu einem späteren Zeitpunkt wiederholt werden.

Reale Prozesse können aber oft nicht rückgängig gemacht werden. Man kann die Vermischung von zwei Flüssigkeiten nicht rückgängig machen. Zwei Züge, die bei der Reservierung von Gleisabschnitten in eine Deadlock-Situation geraten sind, könnte man eventuell wieder rückwärts fahren

lassen, aber das ist dann doch sehr peinlich. Daher muss man Verklemmungen bei Echtzeitsystemen unbedingt vermeiden.

8.5 Priorität

Faustregel:

Die Priorität ist umgekehrt proportional zur "Prozesszeit" zu vergeben.

D.h. ein Thread, der sehr schnell auf Input reagieren soll, sollte eine höhere Priorität haben, als ein langsam reagierender Thread.

Allgemein ist es günstig, **rechen-intensiven** Threads eine niedrigere Priorität zu geben als **I/O-intensiven** Threads.

Die Korrektheit eines Programms darf nicht von der Vergabe von Prioritäten abhängen.

Doug Lea[12] gibt folgende Empfehlungen:

Priorität	Thread-Typ
10	Krisen-Management
7 – 9	interaktiv, ereignisgesteuert
4 – 6	I/O-abhängig
2 – 3	Berechnung im Hintergrund
1	wenn wirklich nichts anderes ansteht

8.6 Komposition anstatt Vererbung/Realisierung

Das Erben der Klasse `Thread` hat zwei Nachteile. Man kann erstens nicht mehr von anderen Klassen erben, und zweitens – eventuell gravierender – stehen dem Anwender alle `Thread`-Methoden zur Verfügung. Auch wenn wir `Runnable` realisieren, besteht die Gefahr, dass der Anwender unsere Methode `run` überschreibt.

Daher mag es sinnvoll sein, den ganzen Thread-Mechanismus durch Komposition zu kapseln:

```
public class   MeinThread
   {
   private Thread task;

   MeinThread ()
      {
      task = new Thread (new MeinRunnable ());
      task.start ();
      }

   private class  MeinRunnable implements Runnable
      {
```

```
      public void run ()
         {
         // ...
         }
      }
   }
```

Oder eventuell noch kompakter:

```
public class   MeinThread
   {
   MeinThread ()
      {
      new Thread
         (
         new Runnable ()
            {
            public void run ()
               {
               // ...
               }
            }
         ).start ();
      }
   }
```

Bemerkung: Es ist **nicht** zu empfehlen, einen Thread in seinem Konstruktor zu starten, wie das hier gemacht wurde und obwohl das einen sehr eleganten Eindruck macht. (Siehe Diskussion unten.)

8.7 Vorzeitige Veröffentlichung von `this`

Das Starten eines Threads in seinem Konstruktor macht einen sehr eleganten Eindruck. Allerdings kann das zu ernsten Fehlern führen, weil das Objekt zu dem Zeitpunkt noch nicht vollständig konstruiert ist, insbesondere wenn der Thread weit oben in einer Vererbungshierarchie gestartet wird. Mit dem Thread-Start wurde der `this`-Zeiger vorzeitig veröffentlicht ***(escaped)***.

Das kann auch passieren, wenn innerhalb des Konstruktors das Objekt, etwa als Listener, irgendwo registriert wird.

Dieses Problem kann man eigentlich nur durch die Verwendung von protected Konstruktoren, publish-Methoden und öffentlichen Factory-Methoden lösen. Das sei an folgendem Beispiel für das Starten eines Threads gezeigt:

```
public class   AutoThreadStart
   {
   public static void   main (String[] arg)
      {
```

```
      BThread.newInstance ();
      }
   }
class AThread extends Thread
   {
   protected String  a;
   protected   AThread () { a = "Thread A"; }
   protected AThread publish ()
      {
      start ();   // oder registriere irgendwo
      return this;
      }
   public static AThread   newInstance ()
      {
      return new AThread ().publish ();
      }
   public void run ()
      {
      while (true)
         {
         System.out.println ("Jetzt läuft " + a);
         try { Thread.sleep (2000); } catch (InterruptedException e) {}
         }
      }
   }
class BThread extends AThread
   {
   protected   BThread () { a = "Thread B"; }
   public static AThread   newInstance ()
      {
      return new BThread ().publish ();
      }
   }
```

8.8 Zeitliche Auflösung

Einige Methoden haben als Argumente eine Zeitangabe (z.B. `sleep`, `wait`, `join`) oder geben eine Zeitangabe zurück (z.B. `System.currentTimeMillis`). Wie genau diese Zeitangabe eingehalten wird bzw. zurückgegeben wird, hängt leider vom zu Grunde liegenden Zeitsystem ab.

Die Zeitsysteme der verwendeten Betriebssysteme haben oft eine geringe Auflösung. Unter Windows liegt die Auflösung bei 10 bis 16 ms. Das bedeutet, dass `System.currentTimeMillis ()` Werte in Inkrementen von 10 bis 16 ms zurückliefert. Oder ein `Thread.sleep (5)` lässt den Thread 10 bis 16 ms schlafen.

Die Methode `System.nanoTime ()` benutzt i.A. ein genaueres Zeitsystem und liefert Nanosekunden zurück, die allerdings nur mit Werten aus anderen Aufrufen dieser Methode verglichen werden können. Daher kann diese Methode manchmal als Workaround verwendet werden.

8.9 Java Virtual Machine Profile Interface

Zeitanalysen werden oft mit Hilfe der Methoden

```
System.currentTimeMillis ()
```
oder besser
```
System.nanoTime ()
```

gemacht. Das reicht in vielen Fällen aus, ist aber eine sehr grobe Methode. Für genauere Analysen sei auf das JVMPI (Java Virtual Machine Profile Interface) und die entsprechende Literatur dazu hingewiesen.

8.10 Adhoc-Thread

Wenn innerhalb von sequentiellem Code eine Situation eintritt, in der plötzlich parallelisiert werden kann oder muss, dann kann an dieser Stelle im Code ein Thread als anonyme innere Klasse definiert, instanziiert und gestartet werden.

```
//... sequentiell
//... nun adhoc-Thread:
new Thread ()
   {
   public void run ()
      {
      //... paralleler Code
      }
   }.start ();
//... ab jetzt zwei Threads
```

Dazu folgendes Beispiel:

```
public class   CodeMitAdhocThread
   {
   public static void   main (String[] arg)
      {
      new CodeMitAdhocThread ().tuWas ();
      }
   public void tuWas ()
      {
      for (int i = 0; i < 3; i++)
         {
         System.out.println ("tuWas sequentiell (" + i + ").");
         }
      Thread   t = new Thread ()
         {
         public void run ()
            {
            for (int i = 0; i < 3; i++)
               {
               System.out.println ("                          "
                  + "paralleler adhoc-Code (" + i + ").");
               Thread.yield ();
               }
            System.out.println ("                         "
               + "paralleler adhoc-Code fertig.");
```

```
            }
         };
      t.start ();
      for (int i = 0; i < 7; i++)
         {
         if (t.isAlive ())
            System.out.println ("tuWas parallel (" + i + ").");
         else
            System.out.println ("tuWas sequentiell (" + i + ").");
         Thread.yield ();
         }
      }
   }
```

8.11 Thread-sichere API-Klassen

Die Klassen `java.util.Vector` und `java.util.Hashtable` sind Thread-sicher, d.h. gleichgültig, wieviele Threads gleichzeitig auf Objekte dieser Klassen zugreifen, ihr Zustand bleibt konsistent. Wenn der Anwender allerdings mehrere Methoden kombiniert, dann kann es zu Überraschungen kommen. Hier muss dann Client-seitiges Locking verwendet werden.

```
Vector<Object> v;
// ...
synchronized (v)  // Client-seitiges Locking
   {
   int   letzter = v.size () - 1;
   if (letzter >= 0)
      {
      v.remove (letzter);
      }
   }
// ...
```

Wenn man sowieso Client-seitig Sperren muss, dann empfiehlt sich aus Gründen der Effizienz die Verwendung der **nicht** Thread-sicheren Klassen `java.util.ArrayList` und `java.util.HashMap`.

Mit `java.util.Collections.synchronizedList (List<E> x)` kann man sich `synchronized` Varianten von allen möglichen Listen erzeugen.

Siehe auch Kapitel "Concurrent Utilities".

8.12 Übungen

8.12.1 StopAndGo

Schreiben Sie einen Thread, der von einem anderen Thread – etwa dem `main` angehalten, fortgesetzt und beendet werden kann.

(Man kann auch `Bewegung.java` nehmen und die mit "`// ???`" gekennzeichneten Methoden implementieren bzw. verändern. Möglicherweise müssen noch Datenelemente ergänzt werden. `StopAndGoGUI.java` ist dann eine Anwendung für die Klasse `Bewegung`.)

Kapitel 9

Concurrency Utilities

Die Programmierung paralleler Tasks ist schwierig und fehlerträchtig. Die Aufgabe kann wesentlich vereinfacht werden, wenn man eine Nebenläufigkeits-Bibliothek verwendet. Java bietet dafür seit J2SE 5.0 das Package `java.util.concurrent` an, mit dem wir uns in diesem Kapitel beschäftigen. Ferner werden noch einige fortgeschrittene Techniken der Parallel-Programmierung angesprochen.

9.1 Timer

Zunächst wollen wir auf eine Klasse hinweisen, die es schon in `java.util` gibt und mit der man Tasks zeitlich für einmalige oder wiederholte Ausführung einplanen kann:

```
public class   Timer
   {
   public   Timer ();
   public   Timer (boolean isDaemon);
   public   Timer (String name);
   public   Timer (String name, boolean isDaemon);
   public void cancel ();  // Beendet den Timer. Eingeplante Tasks werden ausgeplant.
   public int  purge ();   // Entfernt alle ausgeplanten TimerTasks.
   public void schedule (TimerTask task, Date time);
   public void schedule (TimerTask task, Date firstTime, long period);
   public void schedule (TimerTask task, long delay);
   public void schedule (TimerTask task, long delay, long period);
   public void scheduleAtFixedRate (TimerTask task, Date firstTime, long period);
   public void scheduleAtFixedRate (TimerTask task, long delay, long period);
   }

public abstract class   TimerTask implements Runnable
   {
   public   TimerTask ();
```

```
public boolean cancel ();  // Task wird ausgeplant.
                           // Wenn die Task gerade läuft, läuft sie zu Ende.
public abstract void run ();
public long scheduledExecutionTime ();
}
```

Für weitere Erläuterungen sei auf die Java-API-Dokumentation verwiesen.

9.2 Thread-Pools

Die Erzeugung und der Start eines Threads ist wesentlich aufwändiger als der Aufruf einer Prozedur oder Methode.

Anstatt für jede Task einen neuen Thread zu erzeugen und zu starten, kann man einen Thread aus einem Pool von Threads verwenden und ihm die Task zur Ausführung geben. Wenn die Task beendet wird, geht der Thread wieder in den Pool zurück.

Eine weitere wichtige Anwendungsmöglichkeit von Threadpools ist die Begrenzung der Anzahl der Threads, die gestartet werden können. Denn wenn die von Java oder dem zugrundeliegenden Betriebssystem gesetzte Grenze überschritten wird, kommt es zum Absturz. Typisches Beispiel ist ein multithreaded (Web-)Server, der unter hoher Last (zu viele Client-Threads) zusammenbricht.

Threadpools können dann am besten eingesetzt werden, wenn die Tasks **homogen** und **unabhängig** sind. Eine Mischung von langlaufenden und kurzen Tasks führt leicht zu einer Verstopfung des Threadpools, so dass die Ausführung kurzer Tasks unerwartet lange dauert. Bei voneinander abhängigen Tasks riskiert man ein Deadlock, indem z.B. eine Task unendlich lange im Threadpool verbleibt, weil sie auf das Ergenis einer anderen Task wartet, die aber nicht laufen kann, weil der Threadpool voll ist.

Zentral ist die Schnittstelle `Executor`:

```
public interface  Executor
   {
   void  execute (Runnable task);
   }
```

Die Klasse `ThreadPoolExecutor` ist eine Realisierung dieser Schnittstelle und bietet verschiedene Möglichkeiten für Pool-Operationen, auf die wir hier nicht eingehen.

Die einfachste Möglichkeit zu einem Threadpool zu kommen, besteht im Aufruf der `static` Methode `newFixedThreadPool (int anzThreads)` der Klasse `Executors`. Diese Methode gibt ein Objekt vom Typ `ExecutorService` zurück, der die Schnittstelle `Executor` mit der Methode `execute (Runnable)` realisiert. Die Einzelheiten werden im folgenden Beispiel gezeigt.

Zunächst eine Task, die Start- und Endezeit ausgibt und zwischendurch etwas wartet:

```
public class   Task
```

```
   implements  Runnable
   {
   private String name;
   private int wartezeit;
   public   Task (String name, int wartezeit)
      {
      this.name = name;
      this.wartezeit = wartezeit;
      }
   public void run ()
      {
      System.out.println ("Task " + name + " beginnt um "
         + System.currentTimeMillis () + ".");
      try
         {
         Thread.sleep (wartezeit);
         }
         catch (InterruptedException e ) {}
      System.out.println ("Task " + name + " ist fertig um "
         + System.currentTimeMillis () + ".");
      }
   }
```

Um diese Task auszuführen, benützen wir einen Thread-Pool. In dem folgenden Programm erzeugen wir einen Pool der Größe 3 und lassen 10 Tasks durchführen. Schließlich warten wir, bis alle Tasks zu Ende gekommen sind.

```
import   java.util.concurrent.*;
public class   ExecuteTask
   {
   public static void   main (String[] arg)
      {
      ExecutorService executor
         =  Executors.newFixedThreadPool (3);
         // Wir erzeugen einen Thread-Pool mit 3 Threads.
      int   sumWartezeit = 1000;
         // Wir summieren alle Wartezeiten, damit wir
         // wissen, wann wir den executor schließen können.
      java.util.Random  zufall = new java.util.Random ();
      for (int i = 0; i < 10; i++)
         {
         String   name = "" + i;
         int   wartezeit = zufall.nextInt (1000);
         sumWartezeit = sumWartezeit + wartezeit;
         Runnable task = new Task (name, wartezeit);
            // Tasks werden erzeugt
         System.out.println ("Task " + name +
            " mit Wartezeit " + wartezeit
            + " wird an den Threadpool übergeben.");
         executor.execute (task);
            // Tasks werden dem Executor übergeben.
         }
      try
         {
         Thread.sleep (sumWartezeit);
         executor.shutdown ();
         executor.awaitTermination (sumWartezeit,
            TimeUnit.MILLISECONDS);
         }
         catch (InterruptedException e) {}
      }
   }
```

Die Ausgabe dieses Programms ist:

```
Task 0 mit Wartezeit 406 wird hinzugefügt.
Task 1 mit Wartezeit 935 wird hinzugefügt.
Task 2 mit Wartezeit 247 wird hinzugefügt.
Task 3 mit Wartezeit 993 wird hinzugefügt.
Task 4 mit Wartezeit 402 wird hinzugefügt.
Task 5 mit Wartezeit 839 wird hinzugefügt.
Task 6 mit Wartezeit 14 wird hinzugefügt.
Task 7 mit Wartezeit 131 wird hinzugefügt.
Task 8 mit Wartezeit 613 wird hinzugefügt.
Task 9 mit Wartezeit 229 wird hinzugefügt.
Task 0 beginnt um 1103561602040.
Task 1 beginnt um 1103561602040.
Task 2 beginnt um 1103561602040.
Task 2 ist fertig um 1103561602280.
Task 3 beginnt um 1103561602280.
Task 0 ist fertig um 1103561602441.
Task 4 beginnt um 1103561602441.
Task 4 ist fertig um 1103561602841.
Task 5 beginnt um 1103561602841.
Task 1 ist fertig um 1103561602971.
Task 6 beginnt um 1103561602971.
Task 6 ist fertig um 1103561602981.
Task 7 beginnt um 1103561602981.
Task 7 ist fertig um 1103561603122.
Task 8 beginnt um 1103561603122.
Task 3 ist fertig um 1103561603272.
Task 9 beginnt um 1103561603272.
Task 9 ist fertig um 1103561603502.
Task 5 ist fertig um 1103561603682.
Task 8 ist fertig um 1103561603732.
```

Dabei ist zu bemerken, dass die ersten drei Tasks sehr schnell beginnen, während weitere Tasks warten müssen, bis wieder ein Thread zur Verfügung steht.

Da die JVM solange läuft, bis alle Tasks zu Ende gelaufen sind, ein Threadpool aber i.A. immer irgendwelche Tasks am laufen hat, bietet `ExecuteService` Methoden an, mit denen ein Threadpool ordentlich abgeschaltet werden kann. Nach einem `shutdown ()` wird der Threadpool nicht verwendete Threads abschalten und keine neuen Threads zum Laufen bringen. Allerdings muss er warten, bis laufende Threads zu Ende gekommen sind. Das kann eine gewisse Zeit dauern. Daher gibt es die Methode

```
awaitTermination (long timeout, TimeUnit timeunit),
```

um auf die Beendingung zu warten.

Das Threadpool-Framework bietet noch wesentlich mehr. Dazu sei auf die Dokumentation zu J2SE 5.0 verwiesen.

9.3 Semaphore

Das Package `java.util.concurrent` bietet eine Klasse `Semaphore` an, die einen **allgemeinen** oder ***counting*** Semaphor repräsentiert. Allgemeine Semaphore werden verwendet, um den Zugang zu einem Betriebsmittel zahlenmäßig zu begrenzen oder um eine Sende-Warteauf-Ereignis-Situation zu realisieren.

Die Klasse `Semaphore` hat zwei Konstruktoren:

```
Semaphore (int permits)
Semaphore (int permits, boolean fair)
```

Eine FIFO-Strategie für die Warteschlange wird nur garantiert, wenn `fair` auf `true` gesetzt wird. Um eine Aussperrung zu verhindern, sollte man i.A. `fair` auf `true` setzen.

Mit den Methoden

```
void  acquire ()
void  acquire (int permits)
```

kann ein Thread ein oder mehrere Permits beantragen. Wenn die Permits nicht sofort erhältlich sind, wartet der Thread, bis er die Permits erhält oder durch `interrupt` unterbrochen wird.

Bei den nächsten beiden Methoden kann der Thread nicht unterbrochen werden. Allerdings kann nach Erhalt der Permits festgestellt werden, ob ein Unterbrechungsversuch stattgefunden hat.

```
void  acquireUninterruptibly ()
void  acquireUninterruptibly (int permits)
```

Die folgenden Methoden blockieren nicht oder höchstens eine gewisse Zeit:

```
boolean  tryAcquire ()
boolean  tryAcquire (int permits)
boolean  tryAcquire (long timeout, TimeUnit unit)
boolean  tryAcquire (int permits, long timeout, TimeUnit unit)
```

Die Permits werden mit der Methode

```
void release ()
```

wieder freigegeben.

Die Permits entsprechen der Semaphor-Variablen eines allgemeinen oder counting Semaphors und können daher beliebige Werte, also auch Werte über dem Initialisierungswert annehmen. Insbesondere können die Semaphore auch mit 0 oder negativen Werten für die Permits initialisiert werden. In dem Fall sind dann ein oder mehrere `release`-Operationen vor dem ersten erfolgreichen `acquire` nötig.

Beispiel: Wir haben drei Postschalter, bei denen die Abfertigung ein bis zwei Sekunden dauert. Die Kunden warten höchstens zwei Sekunden, bis sie dran kommen, dann geben sie auf.

```
import   java.util.concurrent.*;
import   java.util.*;
public class   Postschalter
   {
   private static final int ANZ_SCHALTER = 3;
   private static final Semaphore   sema
      = new Semaphore (ANZ_SCHALTER, false);
      //= new Semaphore (ANZ_SCHALTER, true);   // Das funktioniert nicht!!
   private static final int ANZ_KUNDEN = 20;
   private static final int KUNDEN_INTERVALL = 300;   // Millisekunden
   private static final int MAX_WARTEZEIT = 2000;  // Millisekunden
```

```
private static final int MIN_ABFERTIGUNGSZEIT = 1000; // Millisekunden
private static final int MAX_ABFERTIGUNGSZEIT = 2000; // Millisekunden
private static final Random   zufall = new Random ();
static class   KundIn extends Thread
   {
   int   kdnr;
   public   KundIn (int kdnr)
      {
      this.kdnr = kdnr;
      }
   public void run ()
      {
      System.out.println ("Kunde " + kdnr + " wartet.");
      long  w = System.currentTimeMillis ();
      try
         {
         if (sema.tryAcquire (MAX_WARTEZEIT, TimeUnit.MILLISECONDS))
            {
            System.out.println ("   Kunde " + kdnr);
            int   z = MIN_ABFERTIGUNGSZEIT
               + zufall.nextInt (
                  MAX_ABFERTIGUNGSZEIT - MIN_ABFERTIGUNGSZEIT);
            System.out.println ("   Kunde " + kdnr);
            try { sleep (z); }
               catch (InterruptedException e) { e.printStackTrace (); }
            w = System.currentTimeMillis () - w;
            System.out.println ("Kunde " + kdnr + " wurde in " + z +
               " Millisekunden abgefertigt"
               + " nach insgesamt " + w + " Millisekunden.");
            }
         else
            {
            w = System.currentTimeMillis () - w;
            System.out.println ("Kunde " + kdnr
               + " hat aufgegeben nach " + w + " Millisekunden.");
            }
         }
         catch (InterruptedException e) { e.printStackTrace (); }
         finally
            {
            sema.release ();
            }
      }  // end run
   }
public static void   main (String[] arg)
   {
   ArrayList<KundIn> ka = new ArrayList<KundIn> ();
   for (int i = 0; i < ANZ_KUNDEN; i++)
      {
      KundIn   k = new KundIn (i);
      ka.add (k);
      k.start ();
      try { Thread.sleep (KUNDEN_INTERVALL); }
         catch (InterruptedException e) { e.printStackTrace (); }
      }
   while (true)
      {
      try
         {
         Thread.sleep (5000);
         }
         catch (InterruptedException e) { e.printStackTrace (); }
      System.out.print ("Die Kunden");
      for (KundIn k :ka)
         {
         if (k.isAlive ()) System.out.print ( " " + k.kdnr);
         }
      System.out.println (" sind immer noch aktiv!");
      }
```

```
    }
  }
```

9.4 Locks

Der `synchronized` Block ist die elementarste Methode, ein Lock für ein Codestück zu beantragen. `Lock`-Klassen im Paket

```
java.util.concurrent.locks
```

bieten darüber hinaus Timeouts, mehrwertige Bedingungsvariable, Read-Write-Locks und die Möglichkeit, das Warten auf ein Lock abzubrechen.

Das Benutzungmuster sieht folgendermaßen aus:

```
Lock  s = ...; // Siehe später.
s.lock ();     // Das Lock wird beantragt.
try
   {
   // Verwendung des geschützten Betriebsmittels
   }
   finally
      {
      s.unlock ();
      }
```

Da ein Lock auf jeden Fall wieder freigegeben werden muss, ist der `finally`-Block notwendig.

`s.lock ()` lässt den Thread warten, bis er das beantragte Lock bekommt. Der Thread kann dabei nicht unterbrochen werden. Dies aber erlaubt `s.lockInterruptibly`:

```
Lock  s = ...;
try
   {
   s.lockInterruptibly ();
   try
      {
      // Verwendung des geschützten Betriebsmittels
      }
      finally
         {
         s.unlock ();
         }
   }
   catch (InterruptedException e)
      {
```

```
      System.err.println ("Thread wurde beim Warten auf ein Lock unterbrochen.");
      }
```

Wenn für den Thread `interrupt ()` aufgerufen wird, dann wird das `s.lockInterruptibly ()` abgebrochen und eine `InterruptedException` geworfen. Das wird i.A. so programmiert, dass bei einem Interrupt der geschützte Code nicht durchlaufen wird. Da der Thread kein Lock bekommen hat, darf das Lock auch nicht freigegeben werden. Daher wird der geschachtelte `try-try-finally-catch`-Konstrukt benötigt.

Mit `tryLock (...) :boolean` kann man Timeouts setzen.

```
Lock  s = ...; // Siehe später.
if (s.tryLock (200, TimeUnit.MILLISECONDS))
   {
   try
      {
      // Verwendung des geschützten Betriebsmittels
      }
      finally
         {
         s.unlock ();
         }
   }
else
   {
   // Tu was anderes ...
   }
```

`tryLock ()` ohne Argumente kommt sofort mit `false` zurück, wenn das Lock nicht erhalten wird, mit Argumenten erst nach der angegebenen Zeit.

Das Interface `Lock` sieht folgendermaßen aus:

```
public interface  Lock
   {
   void  lock ();               // Beantragt das Lock. Blockiert.
                                // Erhöht den hold count.

   void  lockInterruptibly ();  // Beantragt das Lock. Kann durch interrupt ()
                                // abgebrochen werden. Blockiert.
                                // Erhöht den hold count, wenn das Lock gegeben
                                // wird.

   Condition   newCondition ();  // Siehe Bemerkung unten.

   boolean  tryLock ();         // true: Wenn das Lock frei ist, sonst false.
                                // Blockiert nicht.
```

```
boolean  tryLock (long time, TimeUnit unit);
                              // true: Wenn das Lock innerhalb time frei ist,
                              // sonst false.
                              // Blockiert höchstens time.

void  unlock ();              // Erniedrigt hold count. Gibt das Lock bei
                              // hold count == 0 frei.
}
```

Ein Lock bezieht sich immer auf einen Thread, d.h. ein Thread besitzt das Lock oder eben nicht. Wenn ein Thread ein Lock hat, dann bekommt er dasselbe Lock sofort wieder, wenn er es noch einmal beantragt. Dabei wird der *hold count* um eins erhöht. Bei `unlock ()` wird der *hold count* um eins erniedrigt. Wenn Null erreicht wird, wird das Lock freigegeben.

Das Interface `Lock` wird durch folgende Klassen realisiert:

```
ReentrantLock
ReentrantReadWriteLock.ReadLock
ReentrantReadWriteLock.WriteLock
```

Üblicherweise wird für den gegenseitigen Ausschluss die Klasse `ReentrantLock` verwendet. Objekte der anderen beiden Klassen sind über die Klasse `ReentranReadWriteLock` zugänglich und werden zur Lösung des Reader-Writer-Problems verwendet (ähnlich einer Bolt-Variablen):

```
ReadWriteLock  rwl = new ReentrantReadWriteLock ();
Lock  readLock = rwl.readLock ();
Lock  writeLock = rwl.writeLock ();
```

Die Leser müssen dann das `readLock` wie ein `ReentrantLock` zum Schützen des Lesebereichs verwenden. Die Schreiber verwenden das `writeLock` zum Schützen des Schreibbereichs. Das `writeLock` ist ein **exklusives** Lock, während das `readLock` ein **shared** Lock ist, deren Verhalten wir mit folgender Kompatibilitäts-Matrix darstellen:

	`readLock`	`writeLock`
`readLock`	ja	nein
`writeLock`	nein	nein

D.h.: Wenn ein Thread ein `readLock` hat, dann kann ein zweiter (oder n-ter) Thread auch ein `readLock` bekommen, aber kein `writeLock`.

Offenbar sind die Klassen so programmiert, dass die Schreiber Vorrang haben. D.h., wenn ein Schreiber schreiben möchte, dann wird kein weiterer Leser zugelassen.

Die Methode `newCondition ()` wird von `ReadLock` und `WriteLock` *nicht* unterstützt.

Bemerkungen:

1. `newCondition` gibt ein Bedingungsobjekt vom Typ `Condition` zurück, das an das Lock gebunden ist und das die Methode `await ()` anbietet. Der Mechanismus ist mit dem `wait` vergleichbar: `await ()` kann nur aufgerufen werden, wenn man das zugehörige Lock hat. Wenn `await ()` aufgerufen wird, wird der Thread in eine Warteschlange bezüglich des verwendeten Bedingungsobjekts befördert. Dabei wird das Lock freigegeben. Wenn für das Bedingungsobjekt `signal ()` oder `signalAll ()` aufgerufen wird, dann wird ein bzw. werden alle Threads aus der Warteschlange entfernt. Diese Threads müssen sich dann wieder um das Lock bemühen.

   Ein Lock kann mehrere unterschiedliche `Condition`-Objekte liefern. Damit kann man unterschiedliche Wait-Sets verwalten.

   Für das `await` gibt es Varianten mit Timeout oder eine Variante, die nicht unterbrechbar ist.

9.5 Barrieren

Eine Barriere ***(barrier)*** ist ein Synchronisationspunkt ***(barrier point)***, an dem mehrere Threads aufeinander warten, bevor sie mit der Ausführung ihres Codes weitermachen. Die Klasse `CyclicBarrier` bietet Methoden an, solch einen Treffpunkt zu definieren. Die Barriere ist zyklisch, weil sie wiederverwendet werden kann, nachdem sich die Threads getroffen haben.

Mit den beiden Konstruktoren

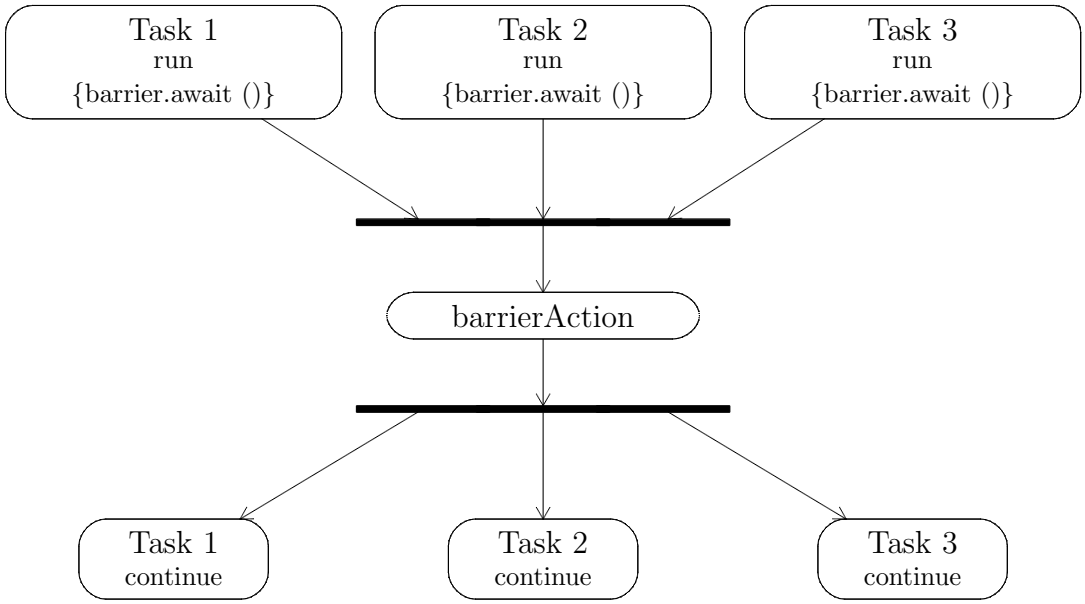
```
CyclicBarrier (int parties)
CyclicBarrier (int parties, Runnable barrierAction)
```

kann man die Anzahl von Teilnehmern am Treffpunkt definieren. Außerdem ist es möglich, eine Aktion als `Runnable` anzugeben, die nach dem Eintreffen der Teilnehmer gestartet wird und zu Ende läuft, bevor die Teilnehmer wieder freigegeben werden.

Die Teilnehmer müssen am Synchronisationspunkt für die Barriere

```
void  await ()
void  await (long timeout, TimeUnit unit)
```

aufrufen. Dabei wird die Exception `BrokenBarrierException` geworfen, wenn ein Teilnehmer den Treffpunkt vorzeitig (etwa durch einen Interrupt) verlässt.



Übung: Darstellung der Barriere in der Ablaufdiagramm-Notation.

Beispiel: Wir haben drei Threads, die je eine Roboterachse steuern. Bei jedem Positionspunkt sollen die Threads aufeinander warten, da der Mainthread einen neuen Positionspunkt bestimmen muss.

```
import   java.util.concurrent.*;
import   java.util.*;
public class   RobAchsen
   {
   private static final int   ANZ_ACHSEN = 3;
   private static final int   FAHRZEIT = 1000;
   private static final Random   zufall = new Random ();
   //private static CyclicBarrier   barrier;
   //private static MyCyclicBarrier barrier;
   private static MyCyclicBarrier2  barrier;
   private static ArrayList<Achse>  achse = new ArrayList<Achse> ();
   private static class Achse extends Thread
      {
      int   achsNr;
      int   posNr;
      String   s = "";
      public   Achse (int achsNr)
         {
         this.achsNr = achsNr;
         for (int i = 0; i < achsNr; i++) s = s + "\t";
         }
      public void run ()
         {
         while (true)
            {
            System.out.println (s + "Achse " + achsNr
               + " fährt nach Pos " + posNr + ".");
            vertueEtwasZeit ();
            System.out.println (s + "Achse " + achsNr
               + " hat Pos " + posNr + " erreicht und wartet an Barriere.");
```

```
            try
               {
               barrier.await ();
               }
               catch (InterruptedException e) { e.printStackTrace (); }
               catch (BrokenBarrierException e ) { e.printStackTrace (); }
            vertueEtwasZeit ();
            System.out.println (s + "Achse " + achsNr
               + " nimmt neue Pos " + posNr + " auf.");
            vertueEtwasZeit ();
            }
         }
      }
   private static class Position
      implements  Runnable
      {
      int   pos = 0;
      public void run ()
         {
         pos = pos + 1;
         System.out.println ("Barrier-Action-" + Thread.currentThread ()
            + " Position setzt neue Positionsnummer " + pos + ".");
         for (Achse a : achse)
            {
            a.posNr = pos;
            }
         }
      }
   public static void   main (String[] arg)
      {
      for (int i = 0; i < ANZ_ACHSEN; i++)
         {
         Achse a = new Achse (i + 1);
         achse.add (a);
         }
      Position position = new Position ();
      //barrier = new CyclicBarrier (ANZ_ACHSEN, position);
      //barrier = new MyCyclicBarrier (ANZ_ACHSEN, position);
      barrier = new MyCyclicBarrier2 (ANZ_ACHSEN, position);
      position.run ();  // Nur um die erste Position zu setzen.
      for (Achse a :achse) a.start ();
      }
   public static void   vertueEtwasZeit ()
      {
      try
         {
         Thread.sleep (zufall.nextInt (FAHRZEIT));
         }
         catch (InterruptedException e) { e.printStackTrace (); }
      }
   }
```

9.6 Latches

Die Klasse `CountDownLatch` hat eine ähnliche Funktion wie `CyclicBarrier` – nämlich die Synchronisation mehrerer Tasks – mit dem Unterschied, dass die wartenden Tasks auf ein Signal warten, dass durch andere Tasks gegeben wird, indem diese einen Zähler herunterzählen. Wenn der Zähler Null erreicht, dann wird eben dieses Signal generiert, dass die wartenden Tasks freigibt.

Die Klasse `CountDownLatch` kann mit einer ganzen Zahl ***(count)*** (anfänglicher Zählerstand) initialisiert werden. Konstruktor und Operationen:

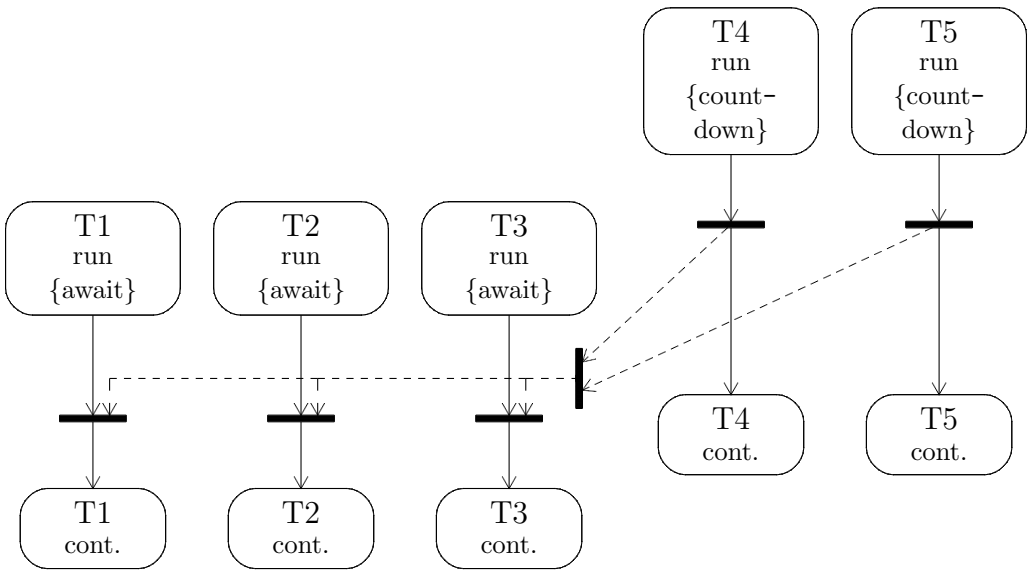
```
CountDownLatch (int count)

void  await ()
void  await (long timeout, TimeUnit unit)

void  countDown ()
int   getCount ()
```

Die Anwendung ist typisch für solche Situationen, in denen eine oder mehrere Tasks warten (`latch.await ()`) müssen, bis eine oder mehrere Operationen – typischerweise in anderen Tasks – erledigt (`latch.countDown ()`) sind.

Wenn Null einmal errreicht ist, bleiben weitere `await`-Aufrufe wirkungslos. Einen `CountDownLatch` kann man nur einmal verwenden *(one-shot-device)*.



Beispiel: Wir haben fünf Arbeiter, die anfangen zu arbeiten, wenn die Werkzeuge bereitliegen, wenn der Chef seinen Kaffee getrunken hat, und wenn es 8:15 Uhr geschlagen hat. Einen zweiten Latch verwenden wir für das Ende des Arbeitstages: Der Chef kann erst nach Hause gehen, wenn jeder der fünf Arbeiter seine Arbeit erledigt hat.

```
import   java.util.*;
import   java.util.concurrent.*;
public class   ArbeiterChef
   {
   public static CountDownLatch  startSignal;
   public static CountDownLatch  endeSignal;
   //public static MyCountDownLatch startSignal;
```

```
   //public static MyCountDownLatch endeSignal;
   public static void   main (String[] arg)
      {
      int   anzArbeiter = 5;
      Arbeiter[]  arbeiter = new Arbeiter[anzArbeiter];
      startSignal = new CountDownLatch (3);
      endeSignal = new CountDownLatch (anzArbeiter);
      //startSignal = new MyCountDownLatch (3);
      //endeSignal = new MyCountDownLatch (anzArbeiter);
      for (int i = 0; i < anzArbeiter; i++)
         {
         arbeiter[i] = new Arbeiter (i + 1);
         }
      Chef  chef = new Chef ();
      Werkzeuge   werkzeuge = new Werkzeuge ();
      Uhr   uhr = new Uhr ();
      }  // end main
   static Random  random = new Random ();
   static void tue (int maxzeit, String was)
      {
      System.out.println (was);
      try
         {
         Thread.sleep (random.nextInt (maxzeit));
         }
         catch (InterruptedException e) { e.printStackTrace (); }
      }
   }  // end class ArbeiterChef
class Arbeiter extends Thread
   {
   int   nummer;
   public   Arbeiter (int nummer)
      {
      this.nummer = nummer;
      start ();
      }
   public void run ()
      {
      ArbeiterChef.tue (100, "Arbeiter " + nummer
         + " wartet auf Beginn der Arbeit.");
      try
         {
         ArbeiterChef.startSignal.await ();
         }
         catch (InterruptedException e) { e.printStackTrace (); }
      ArbeiterChef.tue (1000, "Arbeiter " + nummer + " tut seine Arbeit.");
      ArbeiterChef.tue (1, "Arbeiter " + nummer + " ist fertig.");
      ArbeiterChef.endeSignal.countDown ();
      }
   }  // end class Arbeiter
class Chef extends Thread
   {
   public   Chef ()
      {
      start ();
      }
   public void run ()
      {
      ArbeiterChef.tue (1000, "Chef trinkt seinen Kaffee.");
      ArbeiterChef.tue (1, "Chef ist fertig mit Kaffeetrinken.");
      ArbeiterChef.startSignal.countDown ();
      ArbeiterChef.tue (1, "Chef wartet verantwortungsvoll "
         + "bis die Arbeiter fertig sind.");
      try
         {
         ArbeiterChef.endeSignal.await ();
         }
```

```
         catch (InterruptedException e) { e.printStackTrace (); }
      ArbeiterChef.tue (1, "Chef geht nun auch nach Hause.");
      }
   }  // end class Chef
class Werkzeuge extends Thread
   {
   public   Werkzeuge ()
      {
      start ();
      }
   public void run ()
      {
      ArbeiterChef.tue (1000, "Werkzeuge werden bereitgelegt.");
      ArbeiterChef.tue (1, "Werkzeuge sind bereit.");
      ArbeiterChef.startSignal.countDown ();
      }
   }  // end class Werkzeuge
class Uhr extends Thread
   {
   public   Uhr ()
      {
      start ();
      }
   public void run ()
      {
      ArbeiterChef.tue (1000, "Uhr wartet bis es 8:15 Uhr ist.");
      ArbeiterChef.tue (1, "Es ist 8:15 Uhr!.");
      ArbeiterChef.startSignal.countDown ();
      }
   }  // end class Uhr
```

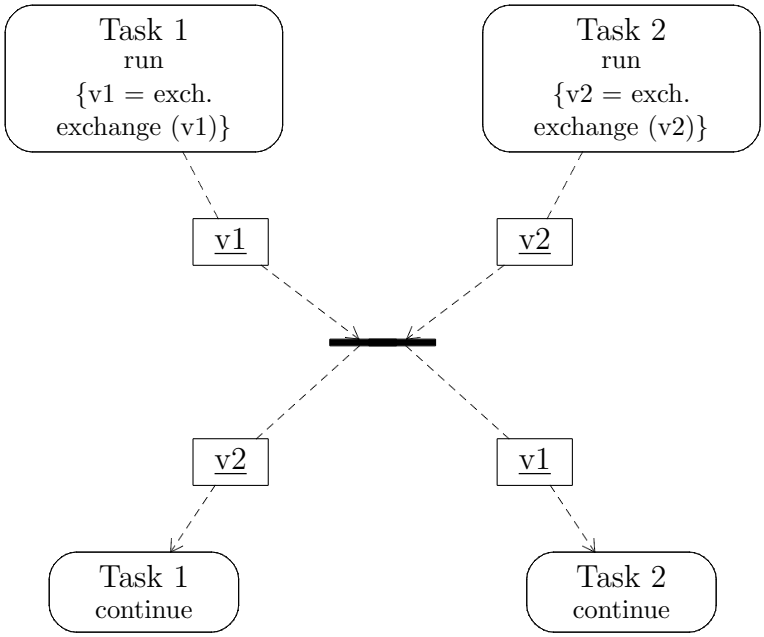
9.7 Austauscher

Mit der Klasse `Exchanger<V>` können Datenstrukturen `V` zwischen zwei Tasks ausgetauscht werden. Jede Task ruft einfach nur die Methode `exchange` auf mit dem auszutauschenden Objekt als Argument. Zurück bekommt sie das getauschte Objekt.

Die beiden Tasks **warten aufeinander** am `exchange`.

```
Exchanger<V> exch = new Exchanger<V> ();
```

Task 1	Task 2
`// ...`	`// ...`
`// ...`	`V v2 = ...`
`V v1 = ...`	`//  ...`
`v1 = exch.exchange (v1);`	`v2 = exch.exchange (v2);`
`// v1 ist nun das v2`	`// v2 ist nun das v1`
`//  von rechts.`	`//  von links.`



Beispiel: Ein Drucker leert eine Tonerkassette, die dann mit einer vollen Kassette ausgetauscht wird, die die Wartung bereitstellt. Die Wartung füllt die leere Kassette wieder.

```
import   java.util.*;
import   java.util.concurrent.*;
public class   Toner
   {
   public static void   main (String[] arg)
      {
      Exchanger<Tonerkassette>   exch = new Exchanger<Tonerkassette> ();
      Drucker  drucker = new Drucker (exch);
      Wartung  wartung = new Wartung (exch);
      }  // end main
   static Random  random = new Random ();
   static void tue (int maxzeit, String was)
      {
      System.out.println (was);
      try
         {
         Thread.sleep (random.nextInt (maxzeit));
         }
         catch (InterruptedException e) { e.printStackTrace (); }
      }
   }  // end class Toner
class Drucker extends Thread
   {
   Exchanger<Tonerkassette>   exch;
   public   Drucker (Exchanger<Tonerkassette> exch)
      {
      this.exch = exch;
      start ();
      }
   public void run ()
```

```
      {
      Tonerkassette  t = new Tonerkassette ("Nr. 1");
      while (true)
         {
         Toner.tue (1, "Drucker will Tonerkassette "
            + t.name + " tauschen.");
         try
            {
            t = exch.exchange (t);
            }
            catch (InterruptedException e) { e.printStackTrace (); }
         Toner.tue (3000, "Drucker hat neue Tonerkassette "
            + t.name + " und druckt.");
         }
      }
   }  // end class Drucker
class Wartung extends Thread
   {
   Exchanger<Tonerkassette>   exch;
   public   Wartung (Exchanger<Tonerkassette> exch)
      {
      this.exch = exch;
      start ();
      }
   public void run ()
      {
      Tonerkassette  t = new Tonerkassette ("Nr. 2");
      while (true)
         {
         t.füllen ();
         Toner.tue (1, "Wartung bietet Tonerkassette "
            + t.name + " zum Tausch an.");
         try
            {
            t = exch.exchange (t);
            }
            catch (InterruptedException e) { e.printStackTrace (); }
         }
      }
   }  // end class Wartung

class Tonerkassette
   {
   String   name;
   public   Tonerkassette (String name)
      {
      this.name = name;
      }
   public void füllen ()
      {
      Toner.tue (3000, "Tonerkassette " + name + " wird gefüllt.");
      }
   }  // end class Tonerkassette
```

Übung: Exchanger

Entwickeln Sie eine Implementierung der Klasse `Exchanger`, indem Sie die Methode `exchange` implementieren. (Falls Sie mit Generics wenig Erfahrung haben, lassen Sie sich nicht stören: `V` ist einfach der Typ der Objekte, die ausgetauscht werden können. Sie können den Typ `V` in der Klasse `Exchanger` einfach verwenden.)

Für die Übung heißt die Klasse `AufgabeAustauscher` und realisiert die Schnittstelle `Austauscher`, die die Methode `exchange` deklariert.

```
public class   AufgabeAustauscher<V> implements Austauscher {
   public         V  exchange (V v) throws InterruptedException {
      // ... Ihr Code
      return v;
   }
}  // end class
```

Der Code befindet sich im Verzeichnis `code`. Das Testprogramm wird mit der Klasse `Anwendung` gestartet und läuft eventuell mit dem Exchanger der Java-API (`JavaExchangerAustauscher`). Dort müssen Sie dann eventuell Ihren Austauscher

```
     = new AufgabeAustauscher<Bierkasten> ();
```

einkommentieren.

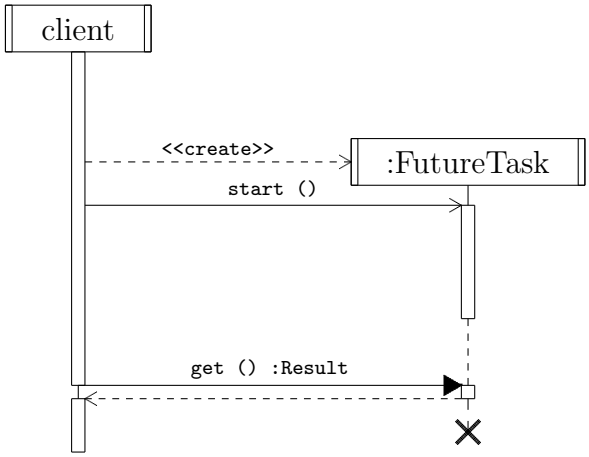
9.8 *Future*

Wenn das Ergebnis einer lang dauernden Aktivität, z.B. einer längeren Berechnung, erst zu einem späteren Zeitpunkt benötigt wird, dann kann es sinnvoll sein, diese Aktivität möglichst frühzeitig anzustoßen und das Ergebnis später, wenn es schließlich benötigt wird, abzuholen. Die Klasse `FutureTask` erlaubt eine einfache Lösung dieses Problems.

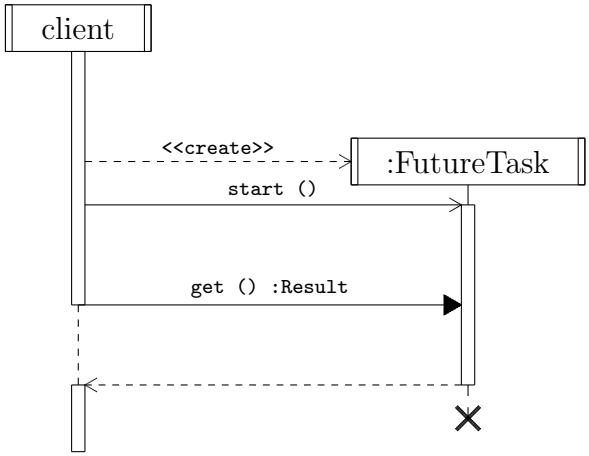
Der Kontruktor von `FutureTask<Resultat>` bekommt als Argument ein `Callable`-Objekt mit, dessen `call`-Methode so implementiert wird, dass die lang dauernde Aktivität aufgerufen wird und dass deren Resultat zurückgegeben wird. `Resultat` ist eine Klasse des Anwenders, deren Objekte das Ergebnis der Aktivität repräsentieren.

Die Objekte von `FutureTask` sind `Runnable`s und können als Thread gestartet werden. Der Mechanismus ist so, dass dabei `call ()` aufgerufen wird, also die lang dauernde Aktivität gestartet wird.

Das Ergebnis der Aktivität wird schließlich mit `get ()` des `FutureTask`-Objekts geholt.



Dieses `get ()` blockiert, falls die Aktivität noch nicht zu Ende gelaufen ist.



Folgendes kommentierte Beispiel dürfte die Einzelheiten des Mechanismus hinreichend erläutern:

```
import   java.util.concurrent.*;
public class   Vorausberechnung
   {
   public static class  Resultat
      // Objekte dieser Klasse repräsentieren das Resultat
      // einer langandauernden Berechnung.
      {
      double[] ergebnis = new double[10];
      }
   public Resultat   berechne ()
      // Dies ist die lange dauernde Aktivität.
      {
      Resultat r = new Resultat ();
      double   x;
```

```
   for (int i = 0; i < 10; i++)
      {
      r.ergebnis[i] = Math.random ();
      System.out.println ("                    Berechnung: " + i);
      try { Thread.sleep (100); }
         catch (InterruptedException e)   { e.printStackTrace (); }
      }
   return r;
   }

// Nun geben wir diese Aktivität (Berechnung)
// als ein Callable einem FuturTask-Objekt future:
public FutureTask<Resultat> future
   = new FutureTask<Resultat>
      (
      new Callable<Resultat> ()
         {
         public Resultat   call ()
            {
            return berechne ();
            }
         }
      );
public static void   main (String[] arg)
   {
   // Jetzt wird das Ganze angewendet:
   Vorausberechnung  vb = new Vorausberechnung ();
   System.out.println ("Berechnung wird als Future gestartet.");
   new Thread (vb.future).start ();
      // Damit wird die Aktivität (Berechnung) gestartet.
   // Wir brauchen das Ergebnis noch nicht.
   // Wir haben noch anderes zu tun:
   long  d = 0;
   System.out.println ("Main tut was anderes.");
   try { Thread.sleep (d = (long) (Math.random () * 1500)); }
      catch (InterruptedException e)   { e.printStackTrace (); }
   System.out.println ("Main braucht dafür " + d + " ms.");
   // Aber jetzt brauchen wir das Ergebnis:
   System.out.println ("Main holt jetzt das Resultat,");
   System.out.println ("   muss aber eventuell auf restliche");
   System.out.println ("   Berechnungen warten.");
   try
      {
      Resultat r = vb.future.get ();
         // Hier wird das Resultat geholt und wir müssen
         // an dieser Stelle eventuell warten, bis das Resultat
         // fertig ist.
      System.out.print ("Resultat: ");
      for (int i = 0; i < 10; i++)
         {
         System.out.print (r.ergebnis[i] + " ");
         }
      System.out.println ();
      }
      catch (ExecutionException e) { e.printStackTrace (); }
      catch (InterruptedException e) { e.printStackTrace (); }
   }
}
```

Bemerkung: Unter C# gibt es das sogenannte Task-based Asnychronous Pattern (TAP), das Ähnliches leistet.

9.9 *Message Queues*

Für Botschaften-Kanäle können verschiedene Klassen verwendet werden, die die Schnittstellen `BlockingQueue` und/oder `Queue` implementieren.

Die Schnittstelle `Queue<E>` hat folgende Methoden:

```
boolean  add (E e);
   // Fügt das Element e in die Queue ein.
   // Wirft eine IllegalStateException, wenn das nicht sofort
   // möglich ist.

E  element ();
   // Gibt den Kopf der Queue zurück ohne ihn zu entfernen.
   // Wirft eine NoSuchElementException, wenn die Queue leer ist.

boolean  offer (E e);
   // Fügt e am Ende der Queue ein, wenn das möglich ist.

E  peek ();
   // Gibt den Kopf der Queue zurück ohne ihn zu entfernen.
   // Gibt null zurück, wenn die Queue leer ist.

E  poll ();
   // Gibt den Kopf der Queue zurück und entfernt ihn.
   // Gibt null zurück, wenn die Queue leer ist.

E  remove ();
   // Gibt den Kopf der Queue zurück und entfernt ihn.
   // Wirft eine NoSuchElementException, wenn die Queue leer ist.
```

Die Meisten Methoden werfen noch weitere Exceptions. Wegen Einzelheiten sei auf die API-Dokumentation verwiesen.

Die Schnittstelle `BlockingQueue<E>` erbt alle Methoden von `Queue<E>` und hat u.a. folgende zusätzliche Methoden:

```
boolean  offer (E e, long  time, TimeUnit unit);
   // Fügt e am Ende der Queue ein, wenn das möglich ist.
   // Blockiert höchstens time.

E  poll (long  time, TimeUnit unit);
   // Gibt den Kopf der Queue zurück und entfernt ihn.
   // Blockiert höchstens time.
   // Gibt null zurück, wenn die Queue leer ist.

void  put (E e);
   // Fügt e am Ende der Queue ein, wenn das möglich ist.
```

```
   // Blockiert, bis es möglich ist.

E  take ();
   // Gibt den Kopf der Queue zurück und entfernt ihn, wenn es möglich ist.
   // Blockiert, bis es möglich ist.
```

Wie bei der Schnittstelle `Queue<E>` werfen die meisten Methoden auch Exceptions. Wegen Einzelheiten sei auf die API-Dokumentation verwiesen.

Diese Schnittstelle wird von folgenden Klassen realisiert:

```
ArrayBlockingQueue
DelayQueue
LinkedBlockingQueue
PriorityBlockingQueue
SynchronousQueue
```

Am häufigsten dürfte `ArrayBlockingQueue<E>` eingesetzt werden. Sie hat eine begrenzte Kapazität und ihre Elemente sind FIFO-geordnet. Ihre wichtigsten Konstruktoren und Methoden sind:

```
public ArrayBlockingQueue<E> (int capacity)
public ArrayBlockingQueue<E> (int capacity, boolean fair)
void  clear ()
```

(`fair = true` gibt wartenden Erzeugern und Verbrauchern in FIFO-Reihenfolge Zutritt zur Queue.)

`ArrayBlockingQueue:` Begrenzte Kapazität.

`DelayQueue:` Objekte müssen eine gewisse Zeit in der Queue verbringen, ehe sie daraus entfernt werden können.

`LinkedBlockingQueue:` "Unendliche" Kapazität.

`PriorityBlockingQueue:` Die Objekte können verglichen werden. Die kleinsten oder größten Objekte werden zuerst aus der Queue entfernt.

`SynchronousQueue:` Enthält keine Objekte. D.h. ein Erzeuger muss mit dem Ablegen eines Objekts so lange warten, bis ein Verbraucher das Objekt aus der Queue entnehmen will. Und umgekehrt. D.h. eine `SynchronousQueue` wird nur dazu verwendet, um zwei Threads aufeinander warten zu lassen (Rendezvous).

9.9.1 *Deques*

`Deque` and `BlockingDeque` sind ***double ended queues,*** die die Entnahme und Ablage von Objekten an beiden Enden erlauben.

9.10 Atomare Variable

Auch Operationen von Standardtypen sind im Prinzip nicht Thread-sicher. Da das Anlegen einer Thread-sicheren Umgebung oft aufwendig ist, gibt es im Paket `java.util.concurrent.atomic` für die Standardtypen spezielle Klassen wie z.B.

```
AtomicInteger
AtomicLong
AtomicReference<V>
...
```

mit denen atomare Operationen wie z.B.

```
int   addAndGet (int delta)
int   getAndAdd (int delta)
int   decrementAndGet ()
int   getAndDecrement ()
int   incrementAndGet ()
int   getAndIncrement ()
int   getAndSet (int newValue)
boolean  compareAndSet (int expect, int update)
```

durchgeführt werden können.

9.11 Collections

9.11.1 Thread-sichere Collections

`Vector` und `Hashtable` sind Thread-sicher, nicht aber `ArrayList`, `HashMap` und andere Collections. Man kann allerdings Thread-sichere Collections erhalten mit:

```
List<E>  liste = Collections.synchronizedList (new ArrayList<E> ());
Map<K, E>   map = Collections.synchronizedMap (new HashMap<K, E> ());
```

Die Methoden von `liste` und `map` sind dann alle Thread-sicher.

9.11.2 Schwach Thread-sichere Collections

Die Thread-sicheren Collections haben den Nachteil, dass sie eine Collection oft sehr lange sperren und damit die Parallelität stark einschränken. Aus diesem Grund wurden sogenannte ***Concurrent Collections*** entwickelt, die mehr Parallelität erlauben, dafür aber nur **schwach konsistent** ***(weakly consistent)*** sind. Als Sperr-Strategie wird das sogenannte ***lock striping*** verwendet. Dabei werden nur Teile einer Collection gesperrt. In der Klasse `ConcurrentHashMap` werden nur die gerade benutzten Hash-Eimer gesperrt.

Bei `ConcurrentHashMap` sind die wichtigen Methoden

```
get
put
containsKey
remove
putIfAbsent
replace
```

Thread-sicher, nicht aber die weniger wichtigen Methoden wie:

```
size
isEmpty
```

Auch die Iteratoren dieser Klasse sind nur schwach konsistent.

`CopyOnWriteArrayList` ist ein concurrent Ersatz für ein synchronized `ArrayList`. `CopyOnWriteArraySet` ist ein concurrent Ersatz für ein synchronized `ArraySet`. Die Collections werden in diesem Fall als unveränderliche Objekte veröffentlicht und im Fall einer Veränderung kopiert. Da das Kopieren bei großen Collections aufwendig ist, eignen sich diese Klassen insbesondere, wenn anstatt von Modifikation Iteration weitaus häufiger ist.

9.12 GUI-Frameworks

Die meisten GUI-Frameworks sind ***single-threaded***. D.h. es gibt einen ***event dipatch thread*** **(EDT)**, der alle GUI-Ereignisse **sequentiell** abarbeitet. Multithreaded Frameworks hatten immer Probleme mit gegenseitigem Ausschluss. Insbesondere bestand die Gefahr eines Deadlocks.

Die sequentielle Behandlung von GUI-Ereignissen bedeutet, dass die individuelle Behandlung sehr schnell abgeschlossen sein muss, um die Antwortgeschwindigkeit *(responsiveness)* einer Anwendung zu gewährleisten. Daher muss eine lang andauernde Ereignisbehandlung in einen eigenen Thread ausgelagert werden. Die Probleme des Multithreadings werden also auf den Anwender geschoben.

9.13 Übungen

9.13.1 Concurrency Utilities

Als eine sehr gute Übung empfiehlt sich eine eigene Implementierung einiger Klassen der Pakete `java.util.concurrent` und `java.util.concurrent.locks`.

9.13.2 Pipes, Queues

Schreiben Sie zwei Threads, die über eine Pipe oder Queue miteinander kommunizieren. Als Grundlage könnte man das Herr-und-Knecht-Beispiel nehmen.

9.13.3 Pipes and Queues

Viele Herren schreiben Befehle in die gleiche Queue. Viele Knechte entnehmen diese Befehle dieser Queue und bearbeiten sie. Ein Befehl wird von einem Knecht bearbeitet. Nach der Bearbeitung sendet der Knecht eine Antwort über eine Pipe an den Herren, der den Befehl gegeben hat. Das bedeutet, dass ein Befehls-Objekt Informationen über den Herren, etwa die Pipe oder den Input-Pipe-Kopf enthalten muss. Der Herr liest die Antwort des Knechtes. Herren und Knechte haben Namen.

Anhang A

Übung zum Warmwerden in Java

Bisher haben wir die Java-Programme immer als **Anwendung** ***(application)*** gestartet. Das folgende Programm kann auch innerhalb eines **Applet-Viewers** ***(applet viewer)*** oder Web-Browsers gestartet werden, da wir eine Klasse definiert haben, die von der Klasse `JApplet` erbt. Diese und alle anderen Klassen die mit "`J`" beginnen, befinden sich in dem Paket `javax.swing`, das alle Komponenten der graphischen Benutzeroberfläche enthält.

```
import   java.awt.*;
import   java.awt.event.*;
import   javax.swing.*;

public class   Schmier extends JApplet
   implements ActionListener, ItemListener, MouseMotionListener
   {
   private Container behälter;
   private int altesX = 0;
   private int altesY = 0;
   private Color  aktuelleFarbe = Color.black;
   private JButton   loeschKnopf;
   private JComboBox farbWahl;
   private JButton   endeKnopf;

   private static final String   schwarz = "Schwarz";
   private static final String   rot = "Rot";
   private static final String   gelb = "Gelb";
   private static final String   grün = "Grün";
   private static final String   blau = "Blau";

   public void init ()
      {
      behälter = this.getContentPane ();
      behälter.setLayout (new FlowLayout ());
      behälter.setBackground (Color.gray);
```

```
      loeschKnopf = new JButton ("Löschen");
      loeschKnopf.addActionListener (this);
      loeschKnopf.setForeground (Color.black);
      loeschKnopf.setBackground (Color.lightGray);
      behälter.add (loeschKnopf);

      farbWahl = new JComboBox ();
      farbWahl.addItemListener (this);
      farbWahl.addItem (schwarz);
      farbWahl.addItem (rot);
      farbWahl.addItem (gelb);
      farbWahl.addItem (grün);
      farbWahl.addItem (blau);
      farbWahl.setForeground (Color.black);
      farbWahl.setBackground (Color.lightGray);
      behälter.add (new JLabel ("Farbe: "));
      behälter.add (farbWahl);

      behälter.addMouseMotionListener (this);
      }

   public void actionPerformed (ActionEvent ereignis)
      {
      Object   ereignisQuelle = ereignis.getSource ();
      if (ereignisQuelle == loeschKnopf)
         {
         repaint ();
         }
      else if (ereignisQuelle == endeKnopf)
         {
         System.exit (0);  // Nur bei Applikationen erlaubt,
                           // nicht erlaubt bei Applets
         }
      }

   public void itemStateChanged (ItemEvent e)
      {
      if (e.getItem () == schwarz) aktuelleFarbe = Color.black;
      else if (e.getItem () == rot) aktuelleFarbe = Color.red;
      else if (e.getItem () == gelb) aktuelleFarbe = Color.yellow;
      else if (e.getItem () == grün) aktuelleFarbe = Color.green;
      else if (e.getItem () == blau) aktuelleFarbe = Color.blue;
      }

   public void mouseDragged (MouseEvent e)
      {
      Graphics g = behälter.getGraphics ();
      g.setColor (aktuelleFarbe);
```

```
   g.drawLine (altesX, altesY, e.getX (), e.getY ());
   altesX = e.getX ();
   altesY = e.getY ();
   }

public void mouseMoved (MouseEvent e)
   {
   altesX = e.getX ();
   altesY = e.getY ();
   }

public void addiere (JComponent komponente)
   {
   behälter.add (komponente);
   }

public static void   main (String[] argument)
   {
   Schmier  s = new Schmier ();
   s.init ();
   s.endeKnopf = new JButton ("Ende");
   s.endeKnopf.addActionListener (s);
   s.endeKnopf.setForeground (Color.black);
   s.endeKnopf.setBackground (Color.lightGray);
   s.addiere (s.endeKnopf);

   JFrame   f = new JFrame ("Schmier");
   f.pack ();  // Trick: Erzeugt Peer-Frame

   f.getContentPane ().add (s, BorderLayout.CENTER);
   f.setSize (600, 400);
   s.start ();
   f.setVisible (true);
   }

}
```

Um auf Ereignisse wie das Drücken der Maus oder das Anklicken eines Knopfs reagieren zu können, werden von der Klasse `Schmier` verschiedene *Ereignis-Empfänger* implementiert: `ActionListener`, `ItemListener` und `MouseMotionListener`. D.h. diese Schnittstellen geben an, welche Methoden implementiert werden müssen, damit das empfangene Ereignis behandelt werden kann. Das sind beim `ActionListener` die Methode `actionPerformed`, bei `ItemListener` die Methode `itemStateChanged` und bei `MouseMotionListener` die Methoden `mouseDragged` und `mouseMoved`.

Die Ereignisse werden von den verschiedenen GUI-Komponenten `JButtons`, `JComboBox` und dem Behälter `JApplet` bei entsprechenden Aktionen des Benutzers erzeugt.

Die Methoden

```
XXXEreignisErzeuger.addXXXListener (XXXEreignisEmpfänger)
```

sagen dem Ereignis-Erzeuger, wer das Ereignis empfängt und behandelt.

Damit die Ereignis-Klassen bekannt sind, muss das Statement

```
import java.awt.event.*;
```

gegeben werden.

Damit das Programm als Applet laufen kann, muss die Klasse `Schmier` von der Klasse `JApplet` erben:

```
public class   Schmier extends JApplet
```

Der Name der Klasse `JApplet` wird durch das Statement

```
import javax.swing.*;
```

bekannt gemacht.

Um `Schmier` auch als Anwendung laufen lassen zu können, haben wir die Methode `main` definiert, die die graphischen Voraussetzungen schafft und die Methoden `init` und `start` aufruft. (Bem.: Nicht mit jedem Applet – insbesondere wenn Parameter übergeben werden – geht das so einfach.)

Um in einem `JApplet` oder einem `JFrame` GUI-Komponenten unterbringen zu können, muss man sich von den Objekten dieser Klassen mit der Methode `getContentPane ()` einen GUI-Komponenten-Behälter (Klasse `Container`) holen.

Übersetzt wird dieses Programm wieder mit:

```
$ javac Schmier.java
```

Als Anwendung kann es laufen mit:

```
$ java Schmier
```

Um es aber als Applet laufen zu lassen, müssen wir es in einer HTML-Datei referenzieren. Ein Beispiel dafür ist die Datei `Schmier.html`:

```
<HTML>
<HEAD>
<TITLE>Das Schmier-Applet</TITLE>
</HEAD>
<BODY>
Wenn Sie einen Java-fähigen Browser benutzen,
bitte schmieren Sie im unten angegebenen Applet rum.
Ansonsten haben Sie Pech gehabt.
```

```
<P>
<APPLET code="Schmier.class" width=600 height=400>
</APPLET>
</BODY>
</HTML>
```

Mit dem Applet-Viewer `appletviewer` können wir nun das Applet anschauen:

```
$ appletviewer Schmier.html
```

Alternativ kann man die Datei `Schmier.html` auch mit einem Web-Browser anschauen.

```
$ netscape file:‘pwd‘/Schmier.html
```

Übung Schmier: Schreiben Sie die Klasse `Schmier` ab. Übersetzen Sie das Programm und lassen Sie es als Anwendung und als Applet laufen.

1. Erweitern Sie das Programm um die Farben Zyan (`Color.cyan`) und Magenta (`Color.magenta`).

2. Beim Farbwahlmenü soll die Hintergrund-Farbe die aktuelle Schmier-Farbe sein. Die Vordergrund-Farbe soll so sein, dass man sie sehen kann, d.h. Weiß oder Schwarz oder vielleicht etwas noch Intelligenteres.

3. Wenn das Fenster überdeckt, vergrößert oder verkleinert wird, dann bleibt die Zeichnung i.A. nicht erhalten, weil wir nicht dafür gesorgt haben, dass in diesen Fällen die Zeichnung wiederhergestellt wird. In dieser Übung wollen wir diesen Mangel beheben.

   Unsere Zeichnung besteht aus einer großen Anzahl kurzer Striche, die wir uns in einem Objekt der Klasse `java.util.ArrayList` merken. Die Methoden dieser Klasse finden sich am Ende von Kapitel "Datentypen" und im Anhang. Dazu ist es aber notwendig für die Striche eine eigene Klasse zu definieren, z.B. `Strich`.

   Ferner müssen wir unsere graphische Benutzeroberfläche etwas erweitern. Insbesondere benötigen wir ein `JPanel`, auf das wir unsere Zeichnung malen.

   Folgende Schritte sind nacheinander durchzuführen:

   (a) In der Klasse `Schmier` bekommt `behälter` ein `BorderLayout`.

   (b) Definiere ein Datenelement vom Typ `JPanel` z.B mit Namen `menue`. Dieses wird instanziiert und mit

   ```
   behälter.add (menue, BorderLayout.NORTH)
   ```

   oben in `behälter` eingefügt.

   (c) Nun werden alle GUI-Komponenten anstatt in `behälter` in `menue` eingefügt.

   (d) Schreiben Sie eine Klasse `Zeichnung`, die von `JPanel` erbt. Sie hat ein Datenelement vom Typ `ArrayList`, das dann alle Strichobjekte aufnehmen soll. Dies muss instanziiert werden.

(e) Schreiben Sie in der Klasse `Zeichnung` eine Methode `loesche`, die die `ArrayList` neu instanziiert. Und dann die Methode `repaint ()` aufruft.

(f) Definieren Sie in der Klasse `Schmier` ein Datenelement vom Typ `Zeichnung`, instanziieren Sie es und fügen Sie es in `behälter` an der Stelle `BorderLayout.CENTER` ein.

(g) Rufen Sie in der Methode `init` (letzte Zeile) die Methode `addMouseMotionListener` anstatt für `behälter` für das `Zeichnung`-Objekt auf.

(h) In der Methode `actionPerformed` wird, wenn der Knopf "Löschen" gedrückt wurde, für das `Zeichnung`-Objekt nur die Methode `loesche ()` aufgerufen.

(i) In der Methode `mouseDragged` wird das `Graphics`-Objekt anstatt vom `behälter` von dem `Zeichnung`-Objekt geholt.

(j) Das Programm sollte jetzt wieder getestet werden.

(k) Schreiben Sie die Klasse `Strich`.

(l) Schreiben Sie in der Klasse `Zeichnung` eine Methode `addiere`, um Objekte vom Typ `Strich` an die `ArrayList` zu addieren.

(m) In der Methode `mouseDragged` wird ein `Strich`-Objekt angelegt und mit `addiere` der Zeichnung hinzugefügt.

(n) Für eine `JComponente` – und das ist unsere Klasse `Zeichnung` – wird die Methode `paintComponent (Graphics g)` aufgerufen, wenn die Komponente neu dargestellt werden soll. Also müssen wir in `Zeichnung` die Methode

```
public void paintComponent (Graphics g)
   {
   super.paintComponent (g);
   ...
   ...
   }
```

definieren und so überschreiben, dass der Inhalt des `ArrayList`-Objekts neu gezeichnet wird.

4. Wahrscheinlich kommt bei Ihnen die Codesequenz

```
g.setColor (...);
g.drawLine (...);
```

mindestens zweimal vor. Solche Codewiederholungen sind schwer wartbar und fehleranfällig. Der Code gehört eigentlich in die Klasse `Strich`. Definieren Sie also in dieser Klasse eine Methode

```
public void zeichne (Graphics g) ...,
```

die einen Strich zeichnet. Verwenden Sie diese Methode anstatt der oben erwähnten Codesequenz.

5. Schreiben Sie eine Klasse `DoppelStrich`, die von der Klasse `Strich` erbt und die jeden Strich doppelt zeichnet (z.B. um drei Pixel versetzt). Wenden Sie diese Klasse an, wobei nur in der Methode `mouseDragged` an einer einzigen Stelle der Klassenname `Strich` durch `DoppelStrich` ersetzt werden muss.

6. Bieten Sie für die beiden Stricharten ein Auswahlmenü an.

7. Bieten Sie weitere "Strich"-arten an, d.h. irgendwelche interessante Verbindungen zwischen zwei Punkten (Rechtecke, Kreise usw.). Die Methoden der Klasse `Graphics` können Sie da auf Ideen bringen.

8. Und nun wird es richtig schön objektorientiert gemacht: Wahrscheinlich haben Sie zur Unterscheidung der Stricharten in der Methode `mouseDragged` einen `if-else`-Konstrukt oder ein `switch` verwendet. Solche Konstrukte sind schwer zu warten. Ferner ist Ihre interessante Verbindung vielleicht so, dass es nicht vernünftig ist, sie von `Strich` erben zu lassen. Wir wollen diese Probleme unter Ausnutzung des Polymorphismus folgendermaßen lösen:

   (a) Definieren Sie in einer eigenen Datei eine Schnittstelle, die alle möglichen Verbindungen repräsentiert und folgende Methoden deklariert:

   ```
   public interface  Verbindung
      void  zeichne (Graphics g)
      Verbindung  newVerbindung (...)
      String   toString ()
   ```

   (b) Lassen Sie `Strich` und alle anderen Verbindungstypen diese Schnittstelle implementieren.

   Dabei gibt `newVerbindung` ein neues `Strich`-Objekt bzw. entsprechend andere Objekte zurück.

   `toString` gibt einfach den Klassennamen zurück, also etwa `Strich`.

   (c) Ersetzen Sie in `paintComponent ()` die Klasse `Strich` durch die Schnittstelle `Verbindung`.

   (d) Definieren Sie für jeden Verbindungstyp ein konstantes Objekt. Diese Objekte addieren Sie anstatt der Strings an die Auswahl-Combobox für die Verbindungstypen. (`JComboBox` ruft für jedes Item `toString` auf, um es darzustellen.)

   (e) Falls nicht schon geschehen, definieren Sie ein Klassenelement vom Typ `Verbindung` etwa mit Namen `aktuelleVerbindung`. In `itemStateChanged` belegen Sie dieses Klassenelement einfach mit `e.getItem ()`, wenn `itemStateChanged` von der entsprechenden Combobox aufgerufen wird.

   (f) Ersetzen Sie in `mouseDragged` das `if-else`- oder `switch`-Konstrukt durch einen `newVerbindung`-Aufruf für die aktuelle Verbindung.

9. Verwenden Sie für die Ereignisbehandlung dedizierte innere Klassen.

10. Erstellen Sie eine Verbindung, die blinkt (etwa `BlinkStrich`).

11. Erstellen Sie einen Thread, der jede Sekunde die Anzahl der Verbindungen ausgibt.

12. Erstellen Sie weitere Verbindungen, die aktiv sind. Versuchen Sie dabei das Threading durch Komposition zu realisieren, um Code-Wiederholungen zu vermeiden.

Literaturverzeichnis

[1] M. Ben-Ari, "Grundlagen der Parallel-Programmierung", Hanser 1985

[2] Joshua Bloch, "Effective Java Programming Language Guide", Addison-Wesley

[3] David Flanagan, "Java in a Nutshell", 3. Auflage, O'Reilly & Associates 1999

[4] David Flanagan, Jim Farley, William Crawford und Kris Magnusson, "Java Enterprise in a Nutshell", O'Reilly & Associates 1999

[5] Naran Gehani und William D. Roome, "The Concurrent C Programming Language", Silicon Press 1989

[6] Brian Goetz, "Java Concurrency in Practice", Addison-Wesley Longman

[7] Hassan Gomaa, "Designing Concurrent, Distributed, and Real-Time Applications with UML", Addison-Wesley

[8] C. A. R. Hoare, "Communicating Sequential Processes", Prentice Hall 1985

[9] Cay S. Horstmann und Gary Cornell, "Core Java. Band 1 – Grundlagen", Prentice Hall 1999

[10] Cay S. Horstmann und Gary Cornell, "Core Java. Band 2 – Expertenwissen", Prentice Hall 2000

[11] Heinz Kredel und Akitoshi Yoshida, "Thread- und Netzwerkprogrammierung mit Java", dpunkt.verlag

[12] Doug Lea, "Concurrent Programming in Java: Design Principles and Patterns", Addison-Wesley

[13] Edward. L. Lamie, "Real-Time Embedded Multithreading Using ThreadX and MIPS", Newnes

[14] Christoph Marscholik, "Anforderungen an einen Echtzeitkern", Wind River Systems 1995

[15] Timothy G. Mattson, Beverly A. Sanders und Berna L. Massingill, "Patterns for Parallel Programming", Addison-Wesley

[16] Scott Oaks und Henry Wong, "Java Threads", OŽReilly

[17] Rainer Oechsle, "Parallele und verteilte Anwendungen in Java", Hanser 2007

[18] Shangping Ren und Gul A. Agha, "A Modular Approach for Programming Embedded Systems", Lecture Notes in Computer Science, 1996

[19] B. Rosenstengel und U. Winand, "Petri-Netze: Eine anwendungsorientierte Einführung", Vieweg

[20] Lothar Piepmeyer, Vorlesungsskriptum "Java Threads", lothar.piepmeyer@hs-furtwangen.de