Algorithmen und Programmierung IV: Nichtsequentielle Programmierung

Robert Tolksdorf
Basiert auf ALP IV, SS 2003
Klaus-Peter Löhr
Freie Universität Berlin

[1] © Peter Löhr, Robert Tolksdorf, Berlin

Inhalt

1. Organisation
2. Einführung
3. Nebenläufige Prozesse
4. Interaktion über Objekte
5. Ablaufsteuerung
6. Implementierung
7. (Interaktion über Nachrichten)

[2] © Peter Löhr, Robert Tolksdorf, Berlin

Überblick

[3] © Peter Löhr, Robert Tolksdorf, Berlin

Thema

- Organisation
- Einführung
  - Nichtsequentialität
    - Was ist Nichtsequentialität?
    - Warum kann man Nichtsequentialität nutzen?
    - Wie wird nichtsequentiell programmiert?
  - Nichtdeterminismus
    - Zeitschnitte
    - Unteilbare Anweisungen
  - Verklemmungen
  - Korrektheit nichtsequentieller Programme

[4] © Peter Löhr, Robert Tolksdorf, Berlin

Organisation

[5] © Peter Löhr, Robert Tolksdorf, Berlin

Rahmendaten

- *Veranstalter*
  Prof. Robert Tolksdorf
  Netzbasierte Informationssysteme
- *Übungen*:
  Katja Silligmann und Andrea Schuhmann
- Vorlesung mit Übung, 2+2 SWS
  - *Vorlesung:*
    Mi, 16:15-17:45, Hörsaal Informatik (003)
  - *Übungen:*
    Übung A Do 10:15-11:45, SR 055,(*für den 22.4. noch 053*) Katja Silligmann
    Übung B Do 12:15-13:45, SR 049, Andrea Schuhmann
    Übung C Do 12:15-13:45, SR 053, Katja Silligmann
    Übung D Do 14:15-15:45, SR 049, Andrea Schuhmann

[6] © Peter Löhr, Robert Tolksdorf, Berlin

Voraussetzungen

- ALP I-III
- Voraussetzungen für ein erfolgreiches Bestehen sind
  - Anwesenheit in den Tutorien (mit höchstens 2-maligem Fehlen)
  - Regelmäßiges Vortragen der Aufgabenlösungen im Tutorium
  - Von *n* ausgegebenen Aufgabenblättern müssen mindestens *n-1* erfolgreich bearbeitet werden. "Erfolgreich" heißt, dass mindestens 50% der erreichbaren Punktzahl des Aufgabenblatts erreicht werden.
  - Erfolgreiche Teilnahme an der Abschlussklausur, d.h. mindestens 50% der erreichbaren Punkte.
- Es werden bei erfolgreichem Bestehen nur benotete Scheine ausgegeben. Die Gesamtnote ergibt sich aus dem Mittel der Übungs- und Klausurleistungen.

[7] © Peter Löhr, Robert Tolksdorf, Berlin

Quellenhinweis

- Die Vorlesung im SoSe 2004 basiert auf der Vorlesung im SoSe 2003 von Prof. Peter Löhr

[8] © Peter Löhr, Robert Tolksdorf, Berlin

Anmeldung

- Für die Teilnahme an den Übungen ist eine *vorherige* Anmeldung im Netz erforderlich. Diese finden Sie über die Adresse *http://www.inf.fu-berlin.de/inst/ag-nbi/lehre/04/V_ALPIV/*
- Achtung Studierende zum Bachelor: Die FU-Satzung für allgemeine Prüfungsangelegenheiten schreibt in §13(4) eine *verbindliche Anmeldung* vor. Wir betrachten die hiesige Anmelden im Netz als in diesem Sinne verbindlich. Die Anmeldung kann bis zum 4. Vorlesungstermin zurückgenommen werden
- Eintrag in Mailingliste über http://lists.spline.inf.fu-berlin.de/mailman/listinfo/nbi_v_alpiv notwendig

1 Einführung

(Fachterminologie Englisch/Deutsch siehe auch http://www.babylonia.org.uk)

1.1 Nichtsequentielle Programme und Prozesse

- Was ist Nichtsequentialität?
- Warum Nichtsequentialität?
- Wie Nichtsequentialität?

- Sequentielle Programme

  versus

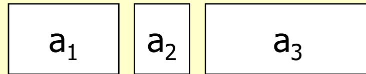
- Nichtsequentielle Programme

1.1.1 Was ist Nichtsequentialität?

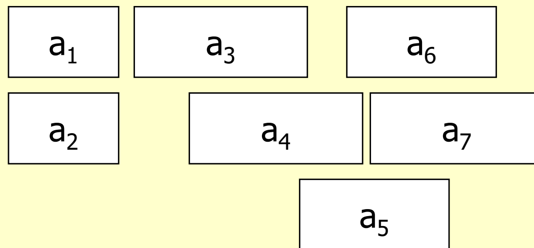
- Aktion =
  - Ausführung einer Programmanweisung,
  - Ausführung einer einer Maschineninstruktion,
  - Ausführung einer eines Mikroprogrammbefehls,
  - …
- Entscheidend ist betrachtete Abstraktionsebene

Programmablauf

- Programmablauf besteht aus Aktionen und heißt
  - sequentiell (sequential),
    - wenn er aus einer Folge von zeitlich nicht überlappenden Aktionen besteht,

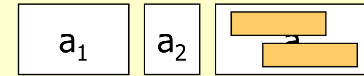


  - parallel (parallel),
    - wenn er nicht sequentiell ist



Programmablauf

- Ein und derselbe Ablauf kann
  - auf *einer* Abstraktionsebene sequentiell,
  - auf *einer anderen* parallel sein !



- Beispiel 1:

```
x = 3; y = 7;
{ x = x+1; y = 2*y; }
z = x+y;
```

- beschreibt einen sequentiellen Ablauf von 5 Aktionen (genauer: Zuweisungen)
- aber auf Hardware-Ebene könnten manche dieser Zuweisungen parallel ausgeführt werden!

Programmablauf

- Manche Programmiersprachen *erlauben explizit* parallele Abläufe, ohne sie vorzuschreiben, z.B.
- Beispiel 2:

```
x = 3; y = 7;
(x,y) = (x+1,2*y);
z = x+y;
```

- aber auf Hardware-Ebene könnte das hier gezeigte *multiple assignment* auch wie

```
x = x+1; y = 2*y;
```

ausgeführt werden (oder in umgekehrter Reihenfolge!)

Terminologie

- eine Programmiersprache heißt

  nichtsequentiell (nebenläufig, concurrent), [konkurrierend – falsche Übersetzung]

  wenn sie Konstrukte enthält, die explizit parallele Abläufe zulassen

- ein Programm oder Programmteil heißt nichtsequentiell, wenn es solche Konstrukte verwendet.

Terminologie

- Typische Nebenläufigkeitskonstrukte identifizieren Programmteile als Prozesse:
- Def.:
  Ein Prozess (process) ist ein Programmteil, der – sobald er gestartet wurde – unabhängig vom Rest des Programms ablaufen kann
- Ein Prozess kann sequentiell oder nichtsequentiell sein
- Häufig ist mit „Prozess" „sequentieller Prozess" gemeint

Nebenläufigkeitsanweisung

- Nebenläufigkeitsanweisung (concurrent statement):
- Beispielsyntax:
  - Statement = … | ConcurrentStatement .
  - ConcurrentStatement = CO Processes OC .
  - Processes = Process { || Process } .
  - Process = StatementSequence .
- Beispiel:

```
x = 3;  y = 7;
CO x = x+1; || y = 2*y; OC
z = x+y;
```

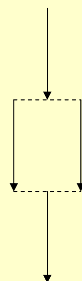

Nebenläufigkeitsanweisung

- Semantik:
  - Die Prozesse einer Nebenläufigkeitsanweisung werden unabhängig voneinander ausgeführt.
  - Die Nebenläufigkeitsanweisung ist beendet, wenn alle Prozesse beendet sind.

```
x = 3;
y = 7;

CO x = x+1; || y = 2*y; OC

z = x+y;
```



Gabelungsanweisung

- Gabelungsanweisung (fork statement)
- Beispielsyntax:
  - ForkStatement = FORK Process .
  - Process = Statement .
- Beispiel

```
x = 3;
y = 7;
FORK      x = x+1;
y = 2*y;
z = x+y;
```

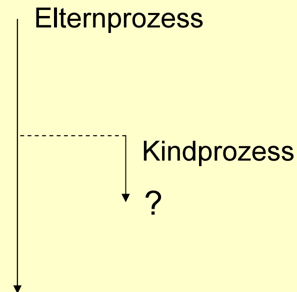
Gabelungsanweisung

- Semantik:
  - Die Gabelungsanweisung startet den angegebenen Prozess.
  - Wann dieser beendet ist, ist *nicht* bekannt!

```
x = 3;
y = 7;

FORK    x = x+1;

y = 2*y;
z = x+y;
```



Gabelungsanweisung

- Es wird grundsätzlich unterstellt, daß Prozeduren *eintrittsinvariant* implementiert sind;
- Eintrittsinvariant: Code- und Datenteil sind getrennt, für jede neue Nutzung wird neuer Datenteil reserviert
- somit ist folgendes *unproblematisch:*

```
...
FORK    p(x+1);
FORK    p(x+2);
p(x+3);
...
```


1.1.2 Warum nichtsequentielle Programmierung?

1. Effizienz eventuell verbesserbar

   - durch schnellere parallele Ausführung
   - sofern entsprechende Hardware vorhanden
   - sofern Hardware entsprechend genutzt wird

Warum nichtsequentielle Programmierung?

2. Problemlösungsstruktur im Programm gut sichtbar („Problemorientierung statt Maschinenorientierung")

- Maschinenorientiert: Wie soll ausgeführt werden?

  ~~...; x = x+1; y = 2*y; ...~~
  ~~...; y = 2*y; x = x+1; ...~~

- Problemorientiert: Was soll errechnet werden?

  ...; CO x = x+1; || y = 2*y; OC ...

Warum nichtsequentielle Programmierung?

3. Systemarchitektur besser abbildbar

- Mehrprozessbetrieb (multitasking) bei
  - Betriebssystemen,
  - Datenbanksystemen,
  - Echtzeitsystemen,
- motiviert durch
  - Effizienzüberlegungen und
  - konkurrierende Nutzung knapper Ressourcen

1.1.3 Wie wird nichtsequentiell programmiert?

- Zentrale Begriffe:
  - Prozess
  - Interaktion zwischen Prozessen
  - Synchronisation von Prozessen

1.1.3.1 Prozessbegriff

- „Prozess" ist ein schillernder Begriff – mit unterschiedlichsten Definitionen:
  - unabhängig ausführbarer Teil eines Programms
  - Programmablauf (!)
  - virtueller Prozessor (Betriebssystem!)
  - industrieller Prozess (Prozessrechner!)
  - u.a.

Prozessbegriff

- Manchmal muß auch zwischen „Prozess" und „Inkarnation eines Prozesses" unterschieden werden:
- „Prozess":
  PROCESS void p(int i) {.....}
- Benutzung:
  START p(x); **START** p(y);
- **START** p(x);
  erzeugt Inkarnation (Exemplar, *instance*)
  des Prozesses p
  [Instanz – falsche Übersetzung]
- Oder: „erzeugt Prozess, der p ausführt"

1.1.3.2 Interaktion zwischen Prozessen

- z.B. über gemeinsame Variable
- Schreiben/Lesen:

```
CO ... time = clock(); ...
  || ... write(time); ...
OC
```

- anders manipulieren:

```
CO ... decrement(seats); ...
  || ... increment(seats); ...
OC
```

 © Peter Löhr, Robert Tolksdorf, Berlin

Konflikt

- Def.:
  Zwei Prozesse, die nebenläufig auf eine gemeinsame Variable x zugreifen, stehen miteinander in Konflikt (conflict, interference) bezüglich x, wenn mindestens einer der Prozesse x verändert.
- Konflikte sind meistens unerwünscht (➔1.2)
- und werden durch Synchronisationsmaßnahmen unterbunden

 © Peter Löhr, Robert Tolksdorf, Berlin

1.1.3.3 Synchronisation

- Prozesse laufen grundsätzlich asynchron, d.h. unabhängig voneinander und mit nicht vorhersagbaren relativen Geschwindigkeiten
- Beispiel:
  `CO p(x); || q(y); OC   r(z);`
- Es bleibt offen, ob p oder q zuerst fertig ist.
- Und: „heute so, morgen so"!

 © Peter Löhr, Robert Tolksdorf, Berlin

Nichtverhersagbarer Ablauf und Synchronisation

- 1) 2) 3) ...

1) p(x) q(y) r(z)
2) p(x) q(y) r(z)
3) p(x) q(y) r(z)

- Die schließende Klammer OC synchronisiert die beiden Prozesse, bevor mit r fortgefahren wird

 © Peter Löhr, Robert Tolksdorf, Berlin

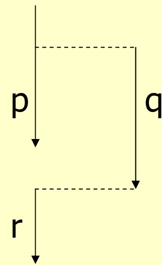
Synchronisation

- Synchronisation bei Gabelungsanweisung:

```
FORK p(x);

q(y); JOIN;

r(z);
```



- JOIN-Anweisung bewirkt Synchronisation des Elternprozesses mit einem Kindprozess durch eventuelles Warten (auf die Beendigung des Kindprozesses)

[33] © Peter Löhr, Robert Tolksdorf, Berlin

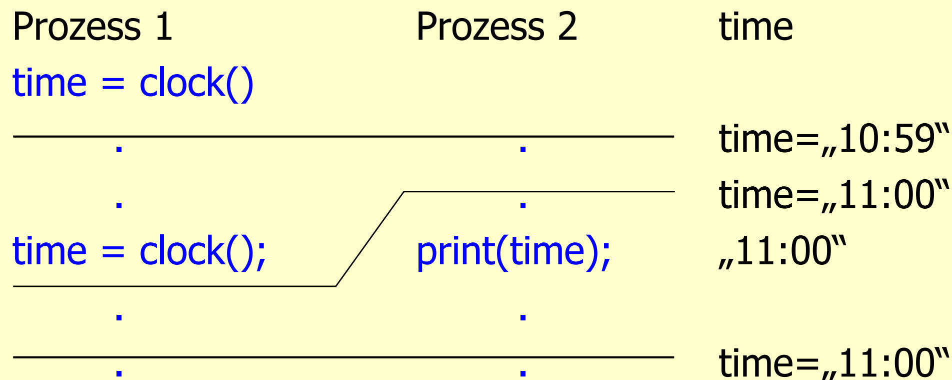
1.2 Nichtdeterminismus

- Der Ablauf eines sequentiellen Programms ist immer deterministisch
  - d.h. gleiche Eingaben führen zu gleichen Zustandsfolgen
- Das Ergebnis ist determiniert
  - d.h. gleiche Eingaben liefern gleiche Ausgaben (sofern nicht explizit mit Zufallszahlen gearbeitet wird)
- Asynchronie
  - führt in der Regel zu nichtdeterministischen Abläufen
  - möglicherweise aber trotzdem zu determinierten Ergebnissen.

[34] © Peter Löhr, Robert Tolksdorf, Berlin

1.2.1 Zeitschnitte

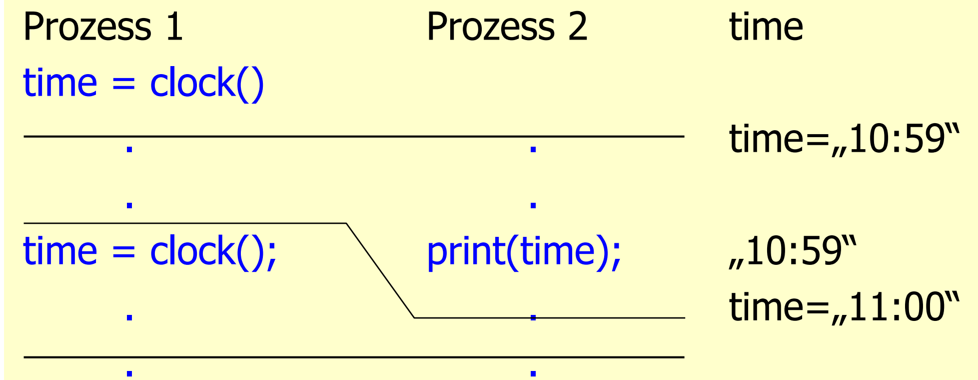
- dienen der Veranschaulichung nichtsequentieller, insbesondere nichtdeterministischer Abläufe:



[35] © Peter Löhr, Robert Tolksdorf, Berlin

1.2.1 Zeitschnitte

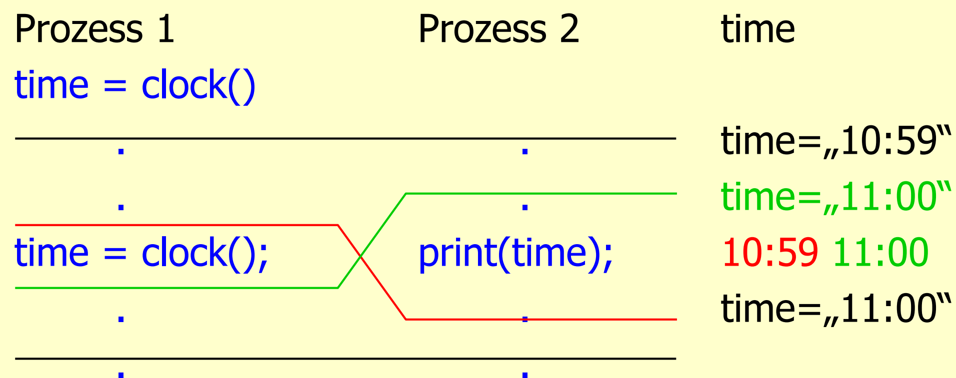
- dienen der Veranschaulichung nichtsequentieller, insbesondere nichtdeterministischer Abläufe:



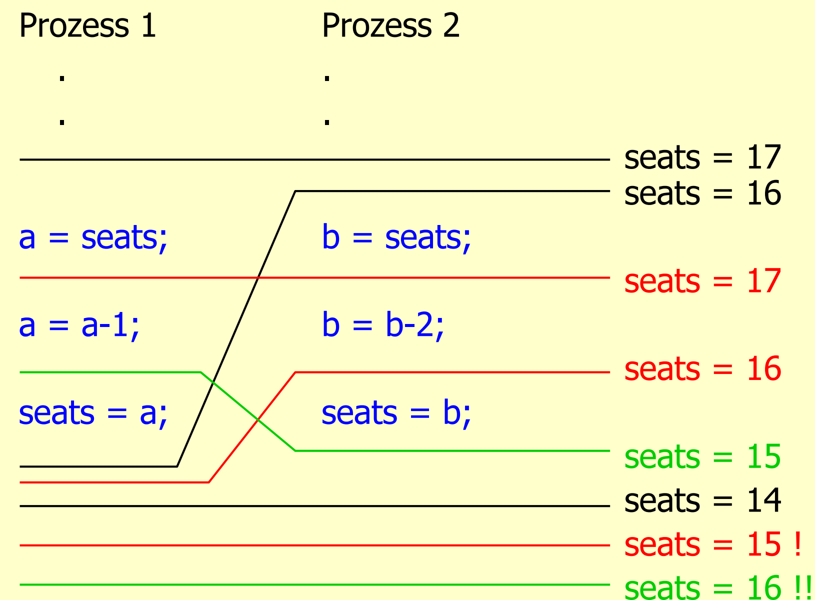
[36] © Peter Löhr, Robert Tolksdorf, Berlin

1.2.1 Zeitschnitte

- dienen der Veranschaulichung nichtsequentieller, insbesondere nichtdeterministischer Abläufe:



Zeitschnitte



Ablaufdeterminismus/Ergebnisdeterminiertheit

Ergebnis / Ablauf	determiniert	nicht determiniert
deterministisch	immer	-
nichtdeterministisch	möglich	meistens

1.2.2 Unteilbare Anweisungen

- verhalten sich so, als würden sie „zeitlos" ausgeführt
- Es können also keine „Zwischenzustände" beobachtet werden
- (statt „unteilbar" auch atomar, *indivisible, atomic*)

- *(Beachte:* in 1.2.1 wurde stillschweigend vorausgesetzt, daß die dortigen Anweisungen unteilbar sind.)

1.2.2 Unteilbare Anweisungen

- Vorsicht!
- Annahmen über die Unteilbarkeit selbst einfachster Anweisungen sind häufig nicht gerechtfertigt.
- Beispiel: a=10L in Java
- Sprachbeschreibungen sind häufig unpräzise in Bezug auf die Unteilbarkeit.
- Implementierungen können auf subtile Weise die Unteilbarkeit verletzen.
- Im Zweifelsfall konservative Betrachtungsweise: Das Lesen bzw. Schreiben einer Arbeitsspeicherzelle durch eine Maschineninstruktion ist unteilbar.

[41] © Peter Löhr, Robert Tolksdorf, Berlin

1.2.2 Unteilbare Anweisungen

- Beispiel:
  wenn Zeichenfelder keine Objekte sind, ist das Lesen/Schreiben einer entsprechenden Variablen nicht unteilbar:

Prozess 1	Prozess 2	
.	.	clock =„11:00"
.	.	time = „10:59"
time = clock;		
//time[0]=clock[0];		time = „10:59"
//time[1]=clock[1];		time = „11:59"
	write(time);	„11:59" !
//time[2]=clock[2];		time = „11:59"
//time[3]=clock[3];		time = „11:09"
//time[4]=clock[4];		time = „11:00"

[42] © Peter Löhr, Robert Tolksdorf, Berlin

1.2.2 Unteilbare Anweisungen

- Klassifikation von Variablen in höheren Programmiersprachen:
  - einfache Variable (*simple variable*):
  - wird von der Hardware unteilbar gelesen/geschrieben
  - private Variable (*private variable*) eines Prozesses:
  - wird nur von diesem Prozess benutzt
  - gemeinsame Variable (*shared variable*) mehrerer Prozesse:
  - nicht privat

[43] © Peter Löhr, Robert Tolksdorf, Berlin

1.2.2 Unteilbare Anweisungen

- Präzisierung der Unteilbarkeit
  (wenn wegen lückenhafter Beschreibung der Sprache und/oder ihrer Implementierung erforderlich):
- Ein Ausdruck ist unteilbar, wenn seine Auswertung
  - keine Nebenwirkungen hat und
  - während der Auswertung
    - höchstens eine
    - – zudem einfache – gemeinsame Variable
    - höchstens einmal gelesen wird.
- Eine Zuweisung v = A ist unteilbar, wenn entweder
  - v privat und A unteilbar ist oder
  - v einfach ist und A sich nur auf private Variable bezieht

[44] © Peter Löhr, Robert Tolksdorf, Berlin

1.2.2 Unteilbare Anweisungen

- Angesichts oft ungesicherter Verhältnisse wünschenswertes Sprachkonstrukt: Klammerung mit ⟨ ⟩
- Beispielsyntax:
  - AtomicExpression = ⟨ Expression ⟩
  - AtomicStatement = ⟨ Statement ⟩
- Semantik:
  - Während ein Prozess einen atomaren Ausdruck auswertet bzw. eine atomare Anweisung ausführt, ruhen alle anderen Prozesse (Synchronisation!)
  - (Aber: Die Praktikabilität von ⟨ ⟩ lässt zu wünschen übrig! →3.1)

1.2.2 Unteilbare Anweisungen

- Beispiele:

```
CO ... ⟨ seats = seats-1; ⟩ ...
|| ... ⟨ seats = seats-2; ⟩ ...
OC

void get(int n){
 ...
  ⟨ seats = seats – n; ⟩
 ...
}
...
CO get(1); || get(2); || get(3); OC
...
```


1.3 Verklemmungen

- Synchronisation löst Probleme
- Ist aber auch Gefahrenquelle:
  - Prozess wartet an Synchronisationspunkt
  - ist gesichert, daß er irgendwann fortfahren kann ?

1.3 Verklemmungen

- Beispiel 1:
  FORK-Variante (vgl. 1.1.3.3), die Verweis auf erzeugten Prozess liefert

main	procP	procQ
PROCESS p,q;		
...		
p = FORK procP();	…	
q = FORK procQ();	…	…
	JOIN q;	JOIN p;
	…	…

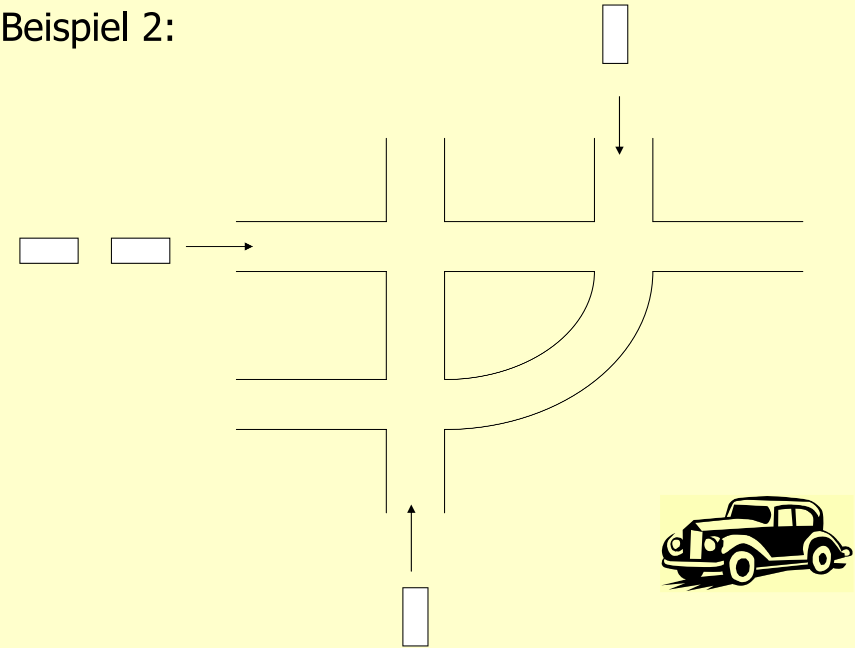
- p wartet auf Beendigung von q
- q wartet auf Beendigung von p.
- Ursache: Programmfehler !

1.3 Verklemmungen

- Def.:
  Beim Ablauf eines nichtsequentiellen Programms liegt eine Verklemmung (deadlock) vor, wenn es Prozesse im *Wartezustand* gibt, die durch *keine mögliche Fortsetzung* des Programms daraus erlöst werden können.
- Bemerkung 1:
  Typischerweise entsteht eine Verklemmung dadurch, daß zwei oder mehr Prozesse *zyklisch* aufeinander warten.
- Bemerkung 2:
  Wenn *kein* zyklisches Warten vorliegt, spricht man häufig von *hangup* statt von *deadlock*.
  - Beispiel: Elternprozess bleibt bei JOIN hängen, weil Kindprozess nicht terminiert.

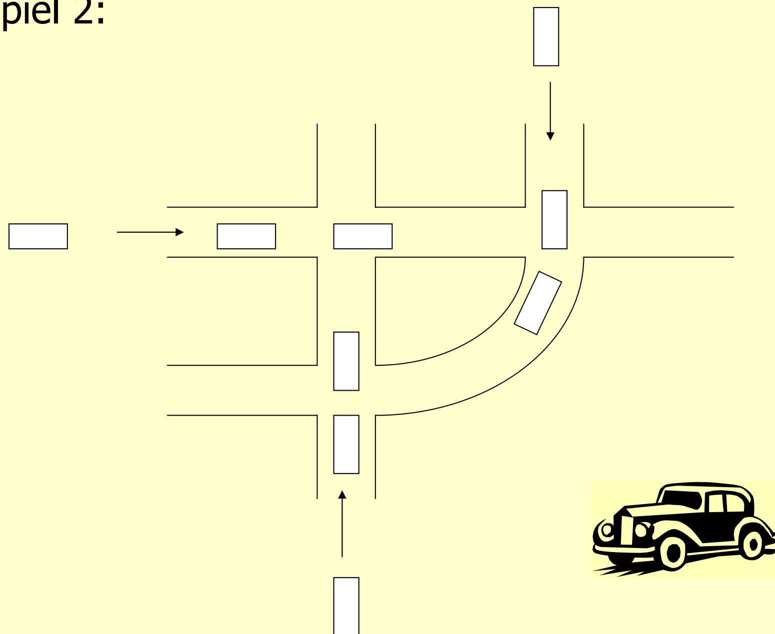
1.3 Verklemmungen

- Beispiel 2:



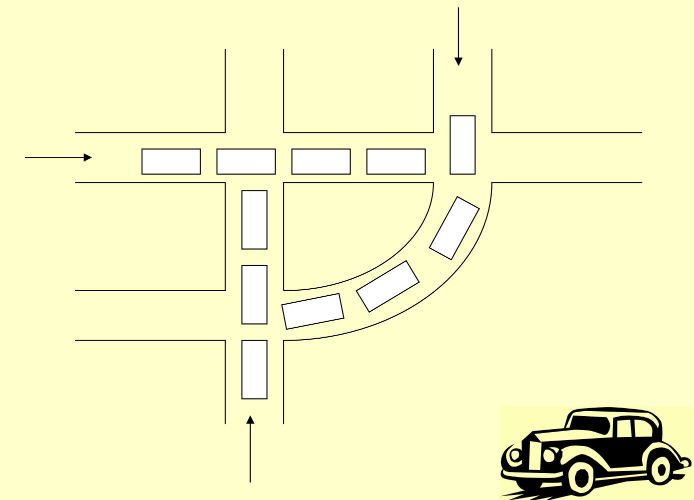
1.3 Verklemmungen

- Beispiel 2:



1.3 Verklemmungen

- Beispiel 2:



Diese Verklemmung tritt nichtdeterministisch ein !

1.4 Korrektheit nichtsequentieller Programme

- Zur Erinnerung:
  Korrektheit sequentieller Programme bezüglich vorgegebener Spezifikation:
- partielle Korrektheit:
  - wenn das Programm terminiert,
  - liefert es das spezifizierte Ergebnis
- totale Korrektheit:
  - partielle Korrektheit +
  - garantierte Termination

1.4 Korrektheit nichtsequentieller Programme

- Zusätzliche Fehlerquellen im nichtsequentiellen Fall:
  - Nichtdeterminismus
  - Verklemmungen, deterministisch oder nichtdeterministisch
- Bezug zur Korrektheit:
  - Nichtdeterminismus bedroht partielle Korrektheit
  - Verklemmungen gefährden Termination

1.4 Korrektheit nichtsequentieller Programme

- Achtung:
  Es gibt Programme, die nicht terminieren sollen – aber natürlich auch verklemmungsfrei sein sollen
- Daher: Erweiterte Definition nötig
- Def.: Korrektheit = Sicherheit + Lebendigkeit
- Sicherheit (*safety*) eines Programms:
  - das Programm produziert nichts Falsches
    (↔ partielle Korrektheit, Determiniertheit)
- Lebendigkeit (*liveness*) eines Programms:
  - das Programm bleibt nicht hängen
    (↔ Termination, Verklemmungsfreiheit)

1.4 Korrektheit nichtsequentieller Programme

- Merke:
- Korrektheit durch Testen nachweisen zu wollen ist von vornherein sinnlos (wegen des Nichtdeterminismus)
- Formale Verifikation ist möglich, aber schwieriger als bei sequentiellen Programmen

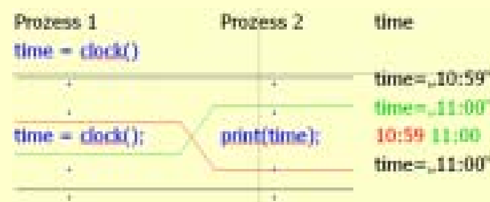
Zusammenfassung

Zusammenfassung Einleitung

- Nichtsequentialität
  - Was ist Nichtsequentialität?
    - Aktionen, parallele Abläufe, Prozesse
    - Nebenläufigkeitsanweiseung (CO a || b OC)
    - Gabelungsanweisung (FORK)
  - Warum kann man Nichtsequentialität nutzen?
    - Effizienz verbesserbar
    - Problemorientierung sichtbar
    - Systemarchitektur abbildbar
  - Wie wird nichtsequentiell progroammiert?
    - Prozess
    - Interaktion, Konflikt
    - Synchronisation (OC-Klammer, JOIN)

Zusammenfassung Einleitung

- Nichtdeterminismus
  - Ablaufdeterminismus
  - Ergebnisdeterminiertheit
  - Zeitschnitte
  - Unteilbare Anweisungen
    - Variablenarten
    - Atomare Ausdrücke
- Verklemmungen
  - Prozesszustand wartend kann auf keinen Fall verlassen werden



Zusammenfassung Einleitung

- Korrektheit nichtsequentieller Programme
  - Fehlerquellen: Nichtdeterminismus, Verklemmungen
  - Sicherheit – partielle Korrektheit
  - Lebendigkeit – Termination
  - Testen sinnlos, Verifikation schwerer