Algorithmen und Programmierung IV: Nichtsequentielle Programmierung

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

Überblick

Überblick

- Verklemmungen
  - Erkennung
  - Auflösung
  - Sicherheit
  - Livelocks

4.3 Verklemmungen

- ... drohen überall, wo synchronisiert wird

- Def. 1 (1.3←):

- 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.

Verklemmungen

- **Def. 2:**
  Eine Verklemmung heißt *deterministisch*, wenn sie bei jedem möglichen Ablauf – mit gleichen Eingabedaten – eintritt (andernfalls *nichtdeterminstisch*).
- **Def. 3:**
  Eine Verklemmung heißt *total* oder *partiell,* je nachdem, ob alle oder nicht alle Prozesse verklemmt sind.
- 3 Alternativen für den Umgang mit Verklemmungen:
  1. **Erkennung** *(detection)* + Auflösung
  2. **Umgehung/Vermeidung** *(avoidance)*
  3. **Ausschluß** *(prevention)*

Verklemmungen

- Wichtigstes Anwendungsgebiet:
  Verklemmungen beim **Sperren von Ressourcen** (nichtdeterm.)
  - **Datenbanksysteme** (DB): 2-Phasen-Sperren
  - **Betriebssysteme** (BS): Betriebsmittelverwaltung

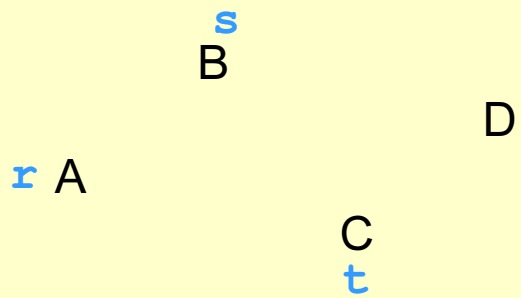
Anzahl Exemplare / Anzahl Typen	1	mehrere
1		1/N (BS)
mehrere	N/1 (DB,BS)	N/M (BS)

4.3.1 Erkennung am Beispiel N/1

- **Def.:**
  Anforderungsgraph ist *markierter gerichteter Graph*, der den Systemzustand modelliert:
  - **Ecke** bedeutet Prozess, ist markiert mit Menge der Ressourcen, die in seinem Besitz sind
  - **Kante** P→Q, markiert mit r, bedeutet:
    P wartet auf Ressource r, die im Besitz von Q ist.
- *Beachte:*
  Jede Ecke hat höchstens eine divergierende Kante.

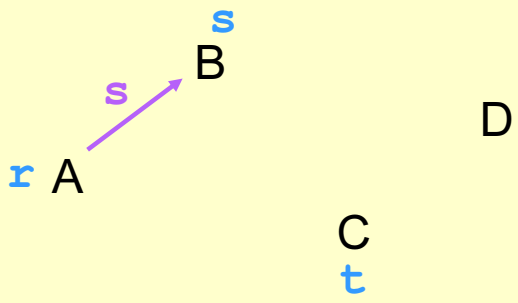
Bsp. mit 4 Prozessen A,B,C,D und 3 Ressourcen r,s,t

Zeit	A	B	C	D
0	`lock(r)`	`lock(s)`	`lock(t)`	



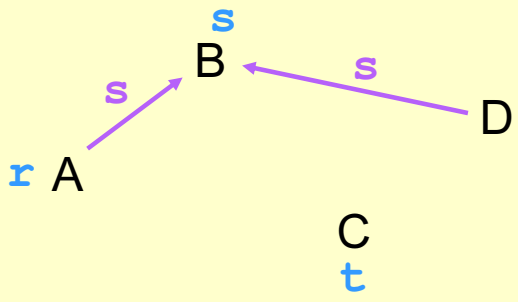
Bsp. mit 4 Prozessen A,B,C,D und 3 Ressourcen r,s,t

Zeit	A	B	C	D
0	lock(r)	lock(s)	lock(t)	
1	lock(s)			



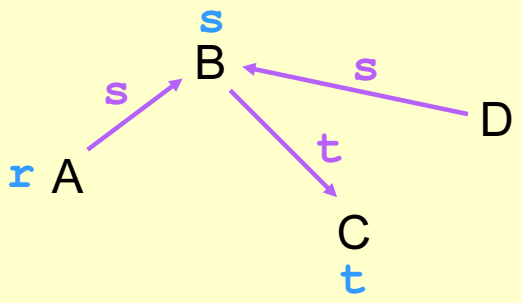
Bsp. mit 4 Prozessen A,B,C,D und 3 Ressourcen r,s,t

Zeit	A	B	C	D
0	lock(r)	lock(s)	lock(t)	
1	lock(s)			
2				lock(s)



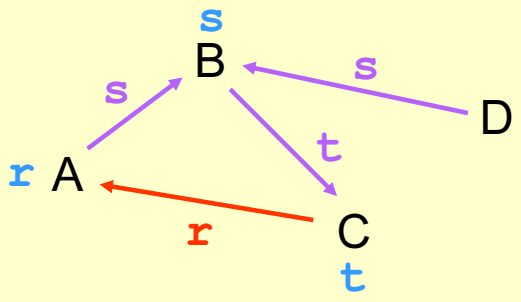
Bsp. mit 4 Prozessen A,B,C,D und 3 Ressourcen r,s,t

Zeit	A	B	C	D
0	lock(r)	lock(s)	lock(t)	
1	lock(s)			
2				lock(s)
3		lock(t)		



Bsp. mit 4 Prozessen A,B,C,D und 3 Ressourcen r,s,t

Zeit	A	B	C	D
0	lock(r)	lock(s)	lock(t)	
1	lock(s)			
2				lock(s)
3		lock(t)		
4			lock(r)	



Verklemmungserkennung

- Eine Verklemmung liegt genau dann vor, wenn der Anforderungsgraph einen Zyklus enthält.

- **Verklemmungs-Erkennung:**
  Ständige Überwachung des Anforderungsgraphen: bei jedem Hinzufügen einer Kante P→Q prüfen, ob damit ein Zyklus geschlossen wird, d.h. ob es einen *Weg von Q nach P* gibt.

- Ohne Backtracking, weil jede Ecke höchstens eine divergierende Kante hat !

Verklemmungserkennung

- Repräsentation des Anforderungsgraphen:
  - Prozesse durchnumeriert mit 1,...,processCount
  - Ressourcen durchnumeriert mit 1,...,resourceCount

- Ferner: me() liefert die Nummer des laufendes Prozesses

```
int[] owner = new int[resourceCount+1];
    // owner[i]: process that owns resource i
    // - if any, else 0

int[] requested = new int[processCount+1];
    // requested[i]: resource requested by
    // process i - if any, else 0
```

Verklemmungserkennung

- **Verklemmungs-Erkennung:** bei jeder Anforderung einer im Augenblick nicht verfügbaren Ressource res wird mit deadlock(int res) geprüft, ob ein Zyklus entsteht:

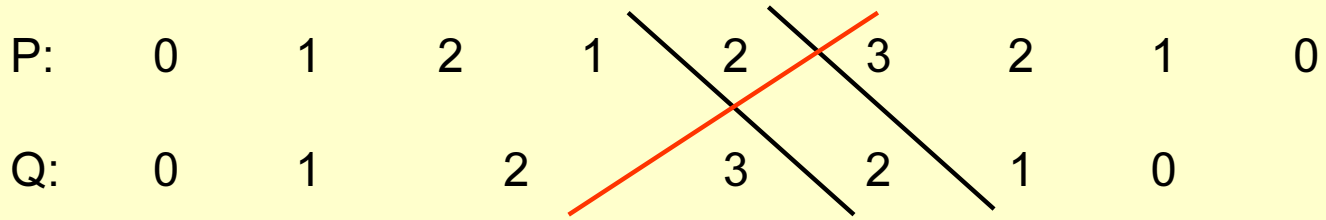
```
boolean deadlock(int res) {
    int p;                  // 1..processCount
  int r = res;            // 0..resourceCount
    do {  p = owner[r];    // != 0
     r = requested[p]; }
  while(r!=0);
  return p==me();
}
```


Verklemmungsauflösung

- **Verklemmungs-Auflösung** durch Wahl eines Opfers, dem gewaltsam die Ressource entzogen wird – mit der Konsequenz
  - (Datenbank:) Transaktion abbrechen und neu starten
  - (Checkpointing:) Prozess zurücksetzen zum letzten Checkpoint
  - (sonst:) Prozess abbrechen

4.3.2 Vermeidung am Beispiel 1/N

- Beispiel:
  - 2 Prozesse P,Q mit n = 4 Exemplaren eines Ressourcen-Typs
  - Vor.:
    Maximalbedarf jedes Prozesses ist bekannt und ist $\leq$ n.



Beobachtungen

- Wie ein Prozess sich in der *Zukunft* verhalten wird, ist *unbekannt* (abgesehen vom Maximalbedarf);
- man muß daher stets mit dem Schlimmsten rechnen, d.h. daß er seinen *Maximalbedarf* fordert;
- wenn man die Ressourcen-Vergabe so steuert, daß jederzeit für mindestens einen nichtblockierten Prozess die Maximalforderung befriedigt werden kann, kann dieser garantiert zu Ende gebracht werden – und danach dann auch der andere.

Sicherheit

- **Def.:**
  Ein Zustand in einem Ablauf eines 2-Prozess-Systems heißt **sicher** *(safe),* wenn in ihm für mindestens einen nichtblockierten Prozess der Maximalbedarf befriedigt werden kann.

- **Feststellung 1:**
  Ein direkter Übergang von einem sicheren Zustand in einen Verklemmungszustand ist nicht möglich.
  - weil damit nach dem ersten Prozess auch der zweite Prozess blockieren müsste – was der Sicherheit widerspricht

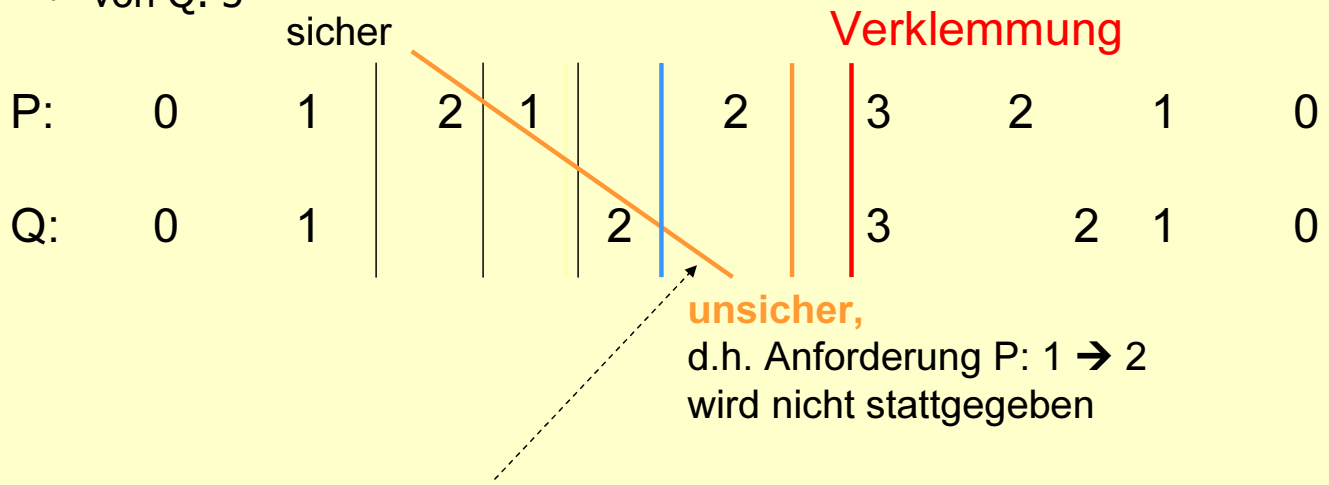
- **Feststellung 2:**
  Verklemmungen werden vermieden, wenn das System stets in einem sicheren Zustand gehalten wird.

Sicherheit

- **Feststellung 3:**
  Die Sicherheit beizubehalten *ist möglich:*
  - der Anfangszustand ist sicher;
  - in einem sicheren Zustand wird der Anforderung eines Prozesses nur dann stattgegeben, wenn der Folgezustand ebenfalls sicher ist (selbst wenn genügend freie Ressourcen vorhanden sein sollten!).
- Die eventuelle Blockierung des anfordernden Prozesses *kann keine Verklemmung verursachen,* da der Ausgangszustand sicher ist.
  - Die Annahme „Verklemmung" impliziert, daß der andere Prozess im Ausgangszustand blockiert war.
  - „Sicherheit" dieses Zustands impliziert, daß sogar die Maximalanforderung des anfordernden Prozesses befriedigt werden kann.
  - Dann kann aber seine aktuelle Anforderung auch befriedigt werden, ohne daß die Sicherheit beeinträchtigt wird – Widerspruch!
- **Feststellung 4:**
  Die Orientierung an der Sicherheit ist eine *konservative Strategie*: Sicherheit ist *hinreichend, aber nicht notwendig* für Verklemmungsfreiheit.

Sicherheit

- Maximalforderung
  - von P: 3
  - von Q: 3



- Bereits dieser Zustand würde nicht zugelassen werden, d.h. der Anforderung von Q (1 → 2) würde nicht stattgegeben werden
  – obwohl nicht notwendig Verklemmung eintritt.

Bankiers-Algorithmus (banker's algorithm) [Dijkstra 65]

- beantwortet die Frage nach der Sicherheit für *beliebig viele* Prozesse:
- Ein Zustand heißt **sicher**,
  - wenn in ihm mindestens ein Prozeß unter Gewährung seines Maximalbedarfs zum Abschluß gebracht werden kann
  - **und** mit den damit freiwerdenden Ressourcen ein weiterer Prozeß zum Abschluß gebracht werden kann
  - **und** mit den damit freiwerdenden .....
  - **usw.**
- *Präzisierung:*

Bankiers-Algorithmus

- Setze B auf Anzahl der Betriebsmittel;
- setze F auf Anzahl der freien Betriebsmittel;
- setze P auf Menge der Prozesse.

- Wiederhole:
  - suche Prozess p∈P, dessen Maximalbedarf befriedigt werden kann;
  - wenn nicht vorhanden, **Unsicher**;
  - erhöhe F um aktuellen Betriebsmittelbesitz von p;
  - wenn F = B, **Sicher**;
  - streiche p aus P.

- Dieser Algorithmus wird entweder mit der Meldung **Unsicher** oder mit der Meldung **Sicher** beendet.

- *Weitere Präzisierung:*

Bankiers-Algorithmus

```
int resCount              // Anzahl der Betriebsmittel
int available             // freie Betriebsmittel
int procCount             // Anzahl der Prozesse
int[] maxclaim            // Maximalforderungen der Prozesse
int[] allocated           // vergebene Betriebsmittel
boolean safe() {
    boolean[] terminated = new boolean[procCount];
    int avail = available;
    int p = 0;
    while(p < procCount) {
          if(terminated[p] || maxclaim[p] – allocated[p] > avail)
                    p++;
          else {terminated[p] = true;
          avail += allocated[p];
          p = 0; }
     }
    return avail == total; }
```

Bemerkungen

- Effizienzverbesserungen möglich, z.B. Prozesse in der Reihenfolge abfallender maximaler Zusatzforderung betrachten.
- Gegebenenfalls Maßnahmen für die Gewährleistung von Fairness treffen !
- Der Bankiers-Algorithmus kann für den Fall N/M verallgemeinert werden („mehrere Währungen“) – was aber eher von akademischem Interesse ist.

4.3.3 Ausschluß am Beispiel N/1

- Zur Erinnerung: *Anforderungsgraph* (4.3.1←)
- **Behauptung:**
  - Die Betriebsmittel seien linear geordnet, und die Prozesse tätigen ihre Anforderungen nur in der Reihenfolge dieser Ordnung
    (Prozess im Besitz von r fordert s nur dann, wenn s > r).
  - *Dann bleibt der Anforderungsgraph zyklenfrei.*

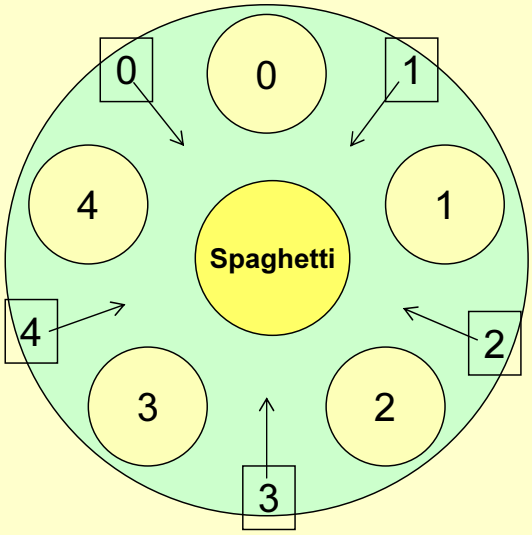
Beweis (indirekt)

- Es liege ein Zyklus vor.
  - Dann gibt es
    - n Prozesse P_i, $n \geq 2$, $i \in [0,n-1]$, und
    - n Betriebsmittel r_i
    - derart, daß
      - P_i besitzt r_i und
      - wartet auf $r_i \oplus 1$. ($\oplus n$ = mod n)
  - Nach Voraussetzung gilt
    $r_i < r_i \oplus 1$ für alle i,
    also
    $r_0 < r_1 < r_2 < \ldots . r_{n-1} < r_0$,
  - was zum ungültigen $r_0 < r_0$
  - und damit zum Widerspruch führt.

Dining Philosophers, [Dijkstra 71]

- Beispiel:
  Die Philosophen bei Tisch
- 5 Teller – aber auch nur 5 Gabeln !
- Zum Essen braucht man 2 Gabeln

➔ 2 benachbarte Philosophen können nicht gleichzeitig essen



Dining Philosophers

- Jeder Philosoph lebt wie folgt:
  Wiederhole:
  - denken;
  - linke und rechte Gabel greifen;
  - essen;
  - linke und rechte Gabel ablegen.

- N/1 Problem mit 5 Ressourcen und 5 Prozessen

```
void philosopher(int i) { // i=0,1,2,3,4
   while(true){ think();
     getForks(i);
     eat();
     putForks(i); }
}
```


Dining Philosophers

- getForks/putForks
  z.B. als Semaphor-Operationen präzisieren (3.3.1←):
  - getForks(i) = P(fork[i],fork[i⊕1]);
  - putForks(i) = V(fork[i],fork[i⊕1]);

- **Verklemmungen?**
  Keine Gefahr.

- **Fairness?**
  - Nicht gegeben !
  - Z.B. 1 und 3 können sich verabreden, 2 auszuhungern *(starvation)*

- *Daher Alternative:*

Dining Philosophers

- getForks(i) = fork[i].P(); fork[i⊕1].P();
- putForks(i) = fork[i].V(); fork[i⊕1].V();

- **Fairness?**
  Ja, wenn P/V fair.

- **Verklemmung?**
  - tritt ein, wenn alle gleichzeitig ihre erste Gabel greifen.

- *Daher Alternative:*

Dining Philosophers

- Gabeln zwar einzeln greifen, aber nur in der Reihenfolge ihrer Ordnung:

→ 4 muß sich anders verhalten:

  - getForks(4) = fork[0].P(); fork[4].P();

  → Verklemmungsfreiheit **und** Fairness

- Übung:
  getForks/putForks als Operationen eines Monitors

4.3.4 Verklemmungsbehandlung in Java

- gibt es **nicht** !
- ... sofern man sie nicht selbst programmiert.

- Manche Systeme praktizieren wenigstens *Erkennung*: wenn z.B. alle Prozesse blockiert sind – aber keiner wegen Wartens auf Eingabe –, kann dies als Verklemmung gemeldet werden.

Verklemmungsbehandlung in Java

- Hinweise auf Verklemmung:
  - Programm terminiert nicht, reagiert nicht auf Eingabe und verbraucht keine Rechenzeit:
    - Totale Verklemmung
  - Programm terminiert nicht, verbraucht Rechenzeit, erbringt aber nur Teilleistung:
    - Partielle Verklemmung
    - **oder** nicht abbrechende Schleife(n)
- **Beachte:**
  Ein System.exit(); – unabhängig vom ausführenden Thread – beendet stets das gesamte Programm.

4.3.5 Livelocks

- (Kunstwort, aus *deadlock* abgeleitet, schwer übersetzbar)
- ... ähneln Deadlocks, daher der Name, sind aber keine!
- ... treten typischerweise bei elementaren Versuchen zur Realisierung von Sperrsynchronisation auf

Beispiel

```
volatile boolean lock1 = false;
volatile boolean lock2 = false;
```

Prozess 1	Prozess 2
do	**do**
lock1 = true;	lock2 = true;
lock1 = !lock2;	lock2 = !lock1;
while(!lock1);	**while**(!lock2);
kritischer Abschnitt	kritischer Abschnitt
lock1 = false;	lock2 = false;

- *„busy waiting"* (normalerweise verpönt)

Livelocks

- Ausschluss garantiert?
  Ja !
- Verklemmungsfreiheit?
  Ja !
- Aber bei zufällig *streng synchronem* Ablauf:
  Livelock, d.h. beide Prozesse sagen gleichzeitig „bitte nach Ihnen“ und wiederholen das unbeschränkt lange.
- *Lösung:* Asynchronie einbauen.

Zusammenfassung

Zusammenfassung

- Verklemmungen
  - Erkennung
  - Auflösung
  - Sicherheit
  - Livelocks