Systemprogrammierung

Verklemmungen

27. Januar 2010

Überblick

Parallele und Funktionale Programmierung, 2. Semester

Ergänzung, Verfeinerung bzw. Vertiefung von PFP

- Teil I, 5. Lebendigkeitsprobleme [1], speziell...
  - ✓ Verklemmung
  - ✓ Bedingungen
  - ✓ Gegenmaßnahmen

...bei Verwendung blockierender Synchronisation, genauer:

- in Semaphore verborgene „Untiefen“ in Maschinenprogrammen
- überlappungsfreie Ausführung kritischer Abschnitte auf Ebene 3
  - verdeutlicht durch Programme der Ebene 5 (C)

Stillstand von Prozessen

Verklemmung mit passivem Warten durch Blockade

dead·lock **1** a standstill resulting from the action of equal and opposed forces; stalemate **2** a tie between opponents in the course of a contest **3** DEADBOLT — to bring or come to a deadlock

Der Begriff bezeichnet (in der Informatik)

> *[. . .] einen Zustand, in dem die beteiligten Prozesse wechselseitig auf den Eintritt von Bedingungen warten, die nur durch andere Prozesse in dieser Gruppe selbst hergestellt werden können. [3]*

- das „geringere Übel“ (im Vergleich zum *livelock*), da dieser Zustand eindeutig erkennbar ist und so die Basis zur „Erholung“ gegeben ist
  - die verklemmten Prozesse sind im Einplanungszustand „blockiert“

Stillstand von Prozessen (Forts.)

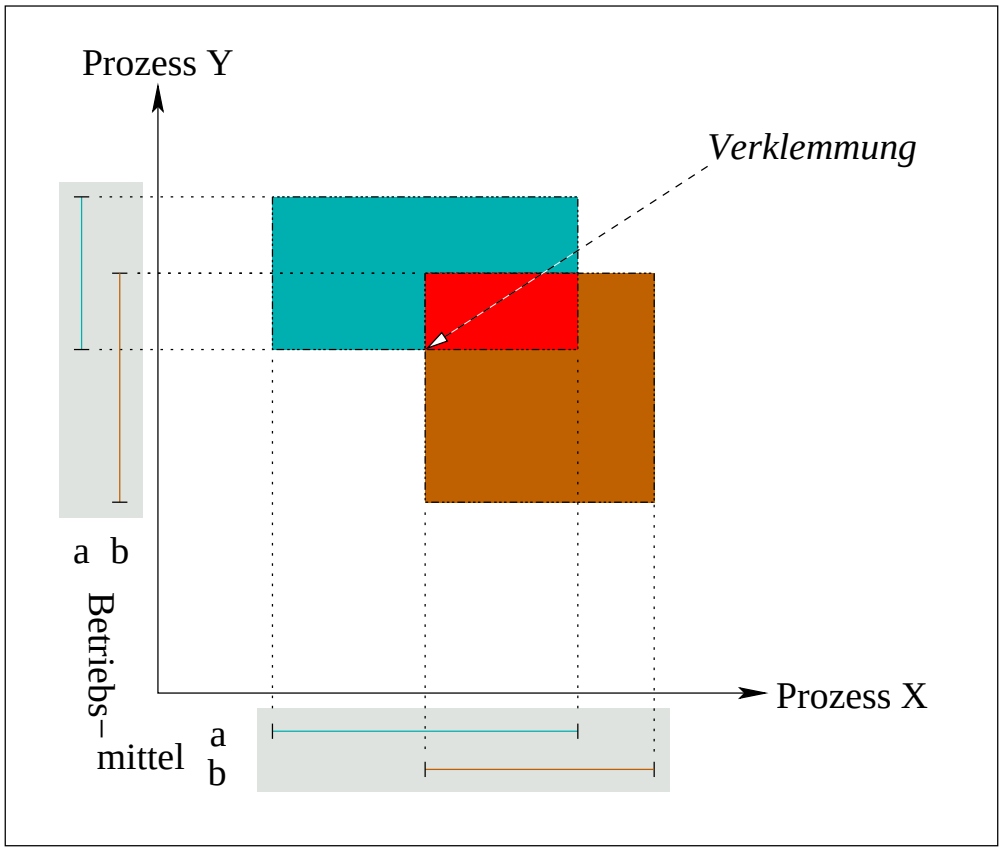
Verklemmung mit aktivem Warten

live·lock ist ...

- ein *deadlock*-ähnlicher Zustand, in dem die involvierten Prozesse zwar nicht blockieren, sie aber auch keine wirklichen Fortschritte in der weiteren Programmausführung erreichen
- wenn die an der Verklemmung beteiligten Prozesse wechselseitig aktiv auf die Bereitstellung von Betriebsmitteln warten:
  - ohne Prozessorabgabe $\mapsto$ *busy waiting*
  - mit Prozessorabgabe, in Laufbereitschaft bleibend $\mapsto$ *lazy waiting*
- das „größere Übel“, da dieser Zustand nicht eindeutig erkennbar ist und damit die Basis zur „Erholung“ fehlt
  - die verklemmten Prozesse haben die Einplanungszustände „laufend“ oder „bereit“ (d.h., jeden anderen außer „blockiert“)
  - die Unterscheidung von unverklemmten Prozessen ist kaum möglich

Entstehung von Verklemmungen

Überlappender Zugriff auf gemeinsame unteilbare Betriebsmittel



Alles hängt davon ab,

1. ob sich die Prozesse einander überlappen und
2. welche Auswirkung die Überlappung auf den Zugriff auf gemeinsame Betriebsmittel hat.

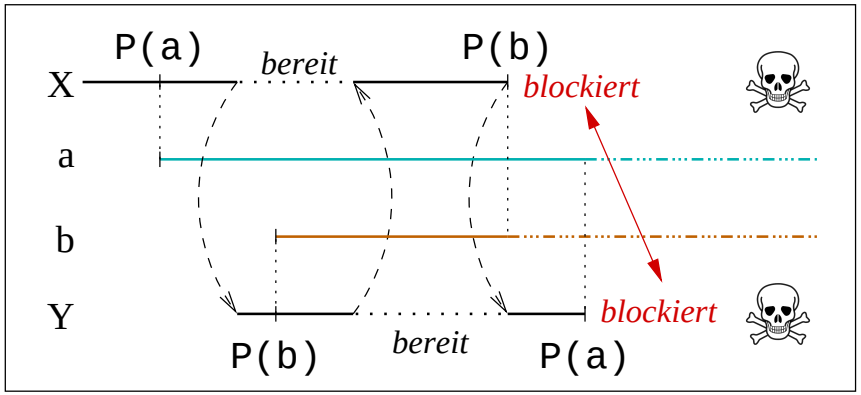
Keine Verklemmung, falls. . .

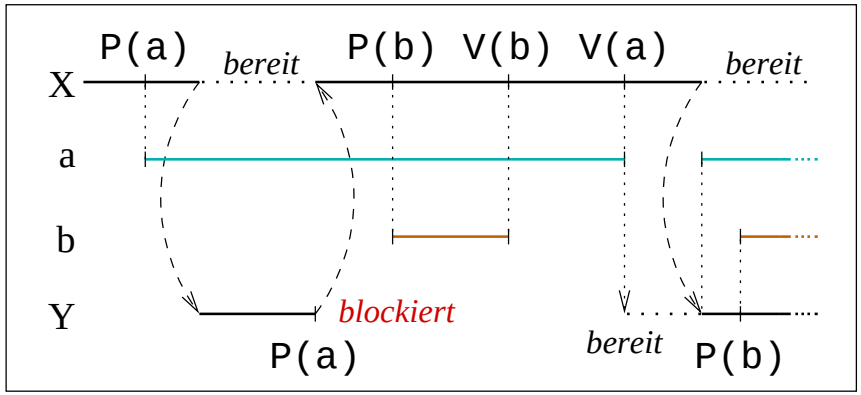
P_X B_a erst nach P_Y oder

P_Y B_b erst nach P_X belegt.

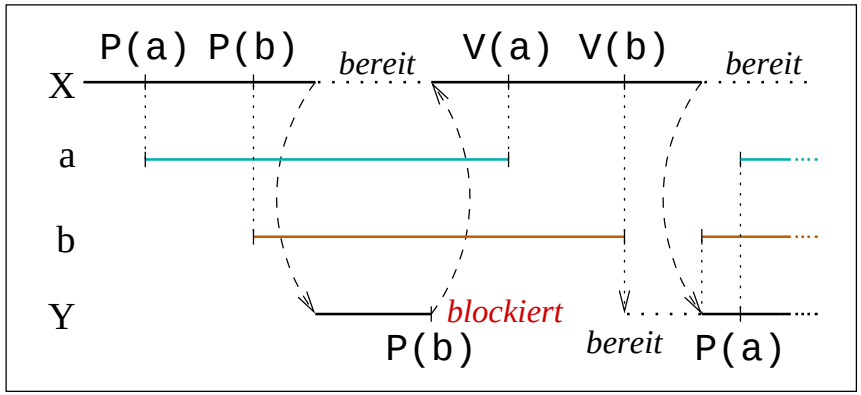
Entstehung von Verklemmungen (Forts.)

Nicht-deterministische Prozessabläufe und Betriebsmittelanforderungen





Verklemmungen sind durch geschickte Prozesseinplanung verhinderbar — vorausgesetzt, die Prozesse wie auch ihre Betriebsmittelanforderungen sind alle bekannt.



☞ kooperative Ausführung von P_X und $P_Y \rightsquigarrow$ Verklemmungsfreiheit

Voraussetzungen für Verklemmungen

Notwendige und hinreichende Bedingungen

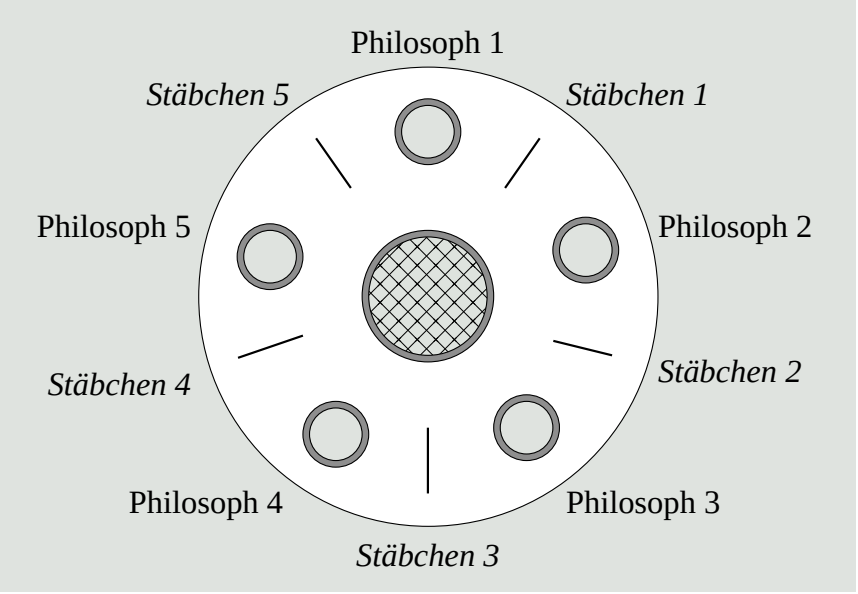
notwendige Bedingungen (müssen erfüllt sein, damit die Aussage zutreffen kann)

1. exklusive Belegung von Betriebsmitteln („*mutual exclusion*“)
   - die umstrittenen Betriebsmittel sind nur unteilbar nutzbar
2. Nachforderung von Betriebsmitteln („*hold and wait*“)
   - die umstrittenen Betriebsmittel sind nur schrittweise belegbar
3. kein Entzug von Betriebsmitteln („*no preemption*“)
   - die umstrittenen Betriebsmittel sind nicht rückforderbar

hinreichende Bedingung (muss erfüllt sein, damit die Aussage zutrifft bzw. „bewiesen“ ist)

4. zirkulares Warten (engl. *circular wait*)
   - Existenz einer geschlossenen Kette wechselseitig wartender Prozesse

Speisende Philosophen
Szenario



Fünf Philosophen, die nichts anderes zu tun haben, als zu denken und zu essen, sitzen an einem runden Tisch. Denken macht hungrig — also wird jeder Philosoph auch essen. Dazu benötigt ein Philosoph jedoch stets beide neben seinem Teller liegenden Stäbchen.

Prozess $\mapsto$ Philosoph

Betriebsmittel $\mapsto$ Stäbchen

- unteilbar

Verklemmungsfall. . .

Die Philosophen nehmen zugleich das eine Stäbchen auf und greifen anschließend auf das andere zu.

Speisende Philosophen (Forts.)

Umsetzung als nebenläufiges Programm

```
void phil (int who) {
    for (;;) {
        think();
        grab(who);
        eat();
        drop(who);
    }
}

void think () {}
void eat   () {}
```

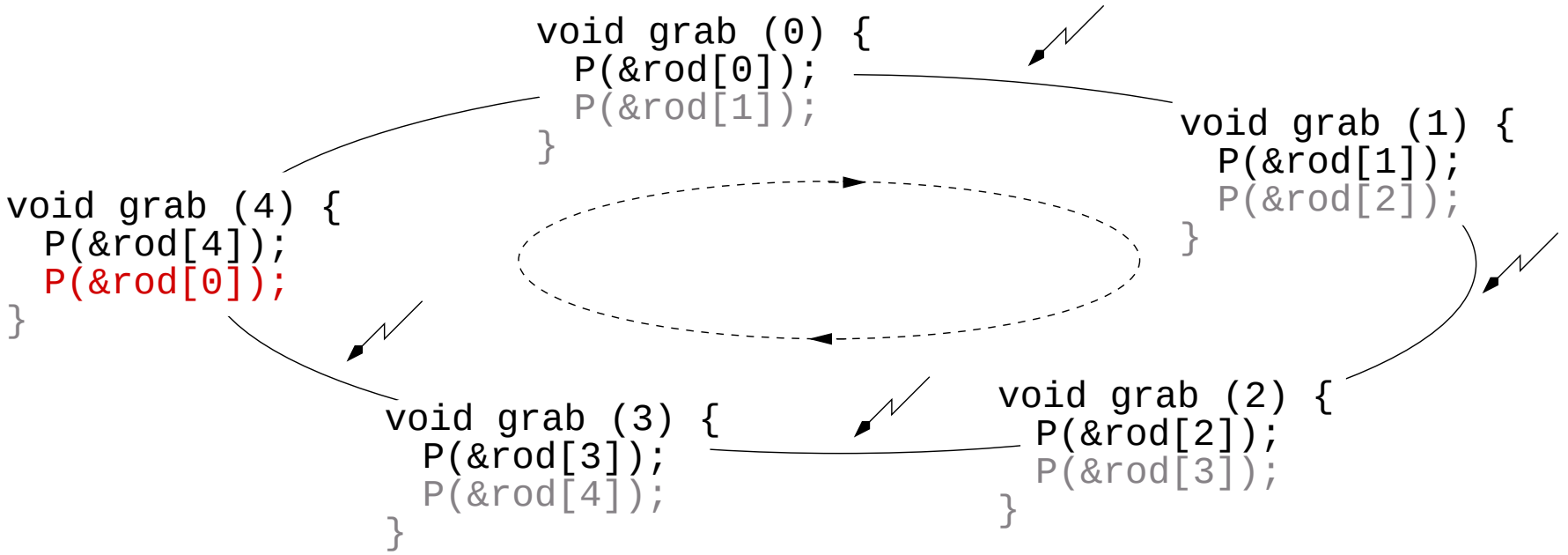
```
semaphore rod[5] = {
    {1, 0}, {1, 0}, {1, 0}, {1, 0}, {1, 0}
};

void grab (int who) {
    P(&rod[who]);
    P(&rod[(who + 1) % NPHIL]);
}

void drop (int who) {
    V(&rod[who]);
    V(&rod[(who + 1) % NPHIL]);
}
```

`P()` fordert zu einem Zeitpunkt nur ein Stäbchen (engl. *rod*) an

`V()` gibt ein Stäbchen frei

Speisende Philosophen (Forts.)

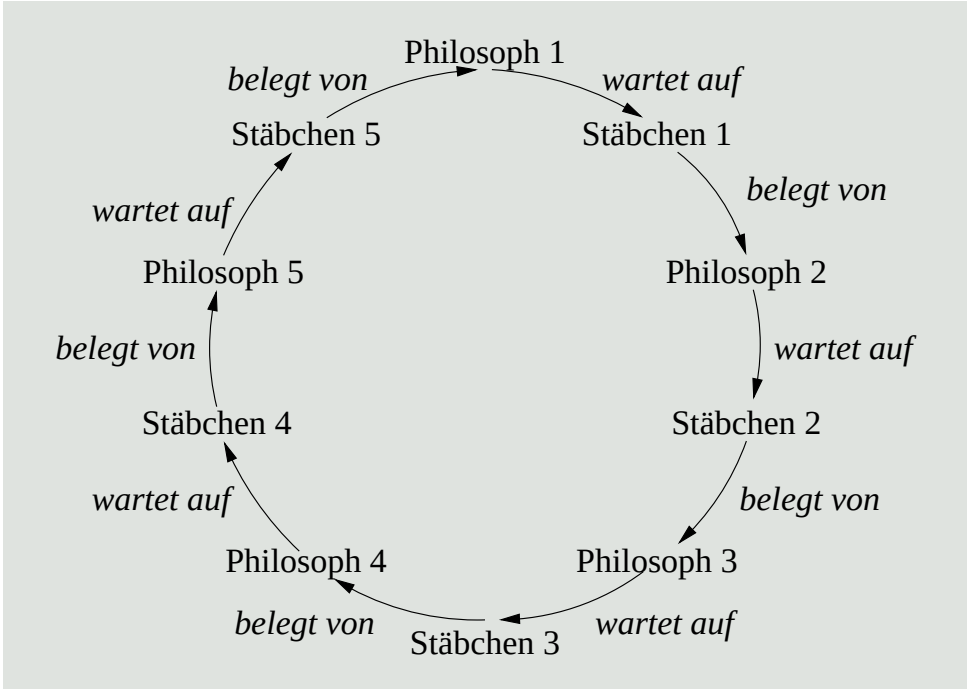
„Ende einer Dienstfahrt“: `grab` $(i-1) \models$ Philosoph$_i$, `rod`$[i-1] \models$ Stäbchen$_i$



1. Philosoph$_1$ nimmt Stäbchen$_1$ auf
2. Philosoph$_2$ überlappt Philosoph$_1$, nimmt Stäbchen$_2$ auf
3. Philosoph$_3$ überlappt Philosoph$_2$, nimmt Stäbchen$_3$ auf
4. Philosoph$_4$ überlappt Philosoph$_3$, nimmt Stäbchen$_4$ auf
5. Philosoph$_5$ überlappt Philosoph$_4$, nimmt Stäbchen$_5$ auf, fordert Stäbchen$_1$ an

Speisende Philosophen (Forts.)

Zirkulares Warten



Betriebsmittelgraph zeigt für jedes Betriebsmittel, welcher Prozess es belegt

Wartegraph verbucht für jeden Prozess, auf welches Betriebsmittel er wartet

- ein geschlossener Kreis (im Wartegraphen) erfasst all die Prozesse, die sich zusammen im *Deadlock* befinden.
- es muss sichergestellt sein, dass ein solcher Kreis entweder nicht entstehen oder dass er erkannt und „durchbrochen“ werden kann

Speisende Philosophen (Forts.)

Kritischer Abschnitt — Verklemmungsfreiheit bei minimaler Nebenläufigkeit

```
semaphore mutex = {1, 0};

void grab (int who) {
    P(&mutex);
    P(&rod[who]);
    P(&rod[(who + 1) % NPHIL]);
    V(&mutex);
}
```

Philosoph$_{who}$ greift auf die benötigten Stäbchen unteilbar zu

- ein binärer Semaphor (`mutex`) zum gegenseitigen Ausschluss
- „*hold and wait*“ vorgebeugt

Problemfall: Verdrängung in `eat()`, Überlappung mit `grab(int)`

- Philosoph$_{who+1}$ muss auf Stäbchen$_{who+1}$ (rechte Hand) warten
  - blockiert im kritischen Abschnitt, ohne diesen freizugeben
- Philosoph$_{who+2}$ muss auf Freigabe des kritischen Abschnitts warten
  - ebenso ergeht es den beiden anderen Philosophen
- schlimmstenfalls kann immer nur ein Philosoph essen

Speisende Philosophen (Forts.)

Synchronisationsaspekte [4]

mehrseitige Synchronisation gegenseitiger Ausschluss beim Gebrauch wiederverwendbarer Betriebsmittel

- keine zwei benachbarten Philosophen können gleichzeitig dasselbe Stäbchen gemeinsam benutzen

einseitige Synchronisation

- ein Philosoph muss warten, bis seine beiden Nachbarn ihm ein Stäbchen zur Verfügung gestellt haben

Randbedingungen

- jeder Philosoph fordert die Stäbchen nacheinander an
  - kein Philosoph legt ein Stäbchen zurück, wenn er feststellt, dass das andere bereits vom Nachbarn aufgenommen worden ist
- ein Philosoph kann seinem Nachbarn ein bereits aufgenommenes Stäbchen nicht entreissen

Speisende Philosophen (Forts.)

Ein in der Praxis nicht selten anzufindendes Problem...

- auf mehreren Magnetbändern vorliegende Daten sortieren
- einen kontinuierlichen Strom kodierter Informationen umkodieren
- Nachrichten von einem Eingangsport auf einen Ausgangsport leiten
- Pakete auf einem Ringnetz zur übernächsten Station durchschleusen
- Daten von dem einen *Backup Medium* auf ein anderes transferieren

- überall dort, wo eine von mehreren Prozessen benutzte Routine (mindestens) zwei wiederverwendbare Betriebsmittel zugleich benötigt, diese aber nur nacheinander angefordert werden können

Verklemmungsvorbeugung
(engl. *deadlock prevention*)

indirekte Methoden entkräften eine der Bedingungen 1–3

1. nicht-blockierende Verfahren verwenden
2. Betriebsmittelanforderungen unteilbar (atomar) auslegen
3. Betriebsmittelentzug durch Virtualisierung ermöglichen
   - virtueller Speicher, virtuelle Geräte, virtuelle Prozessoren

direkte Methoden entkräften Bedingung 4

4. lineare/totale Ordnung von Betriebsmittelklassen einführen:
   - Betriebsmittel B_i ist nur dann erfolgreich vor B_j belegbar, wenn i linear vor j angeordnet ist (d.h. $i < j$).

- Regeln, die das Eintreten von Verklemmungen verhindern
- Methoden, die zur Entwurfs- bzw. Implementierungszeit greifen

Verklemmungsvermeidung
(engl. *deadlock avoidance*)

Verhinderung von Wartezyklen durch strategische Maßnahmen:

- keine der drei notwendigen Bedingungen wird entkräftet
- fortlaufende Bedarfsanalyse schließt zirkulares Warten aus

Prozesse und ihre Betriebsmittelanforderungen sind zu steuern:

- das System wird (laufend) auf „unsichere Zustände“ hin überprüft
- Zuteilungsablehnung im Falle ungedeckten Betriebsmittelbedarfs
- anfordernde Prozesse nicht bedienen bzw. frühzeitig suspendieren
- Betriebsmittelnutzung einschränken $\leadsto$ „sichere Zustände“

☞ *à priori* Wissen erforderlich: maximaler Betriebsmittelbedarf

Sicherer/Unsicherer Zustand

Speisende Philosophen

Ausgangspunkt fünf Stäbchen sind insgesamt vorhanden

- jeder der fünf Philosophen braucht zwei Stäbchen zum Essen

Situation P_1, P_2 und P_3 haben je ein Stäbchen; zwei Stäbchen sind frei

- P_4 fordert ein Stäbchen an $\rightarrow$ ein Stäbchen wäre dann noch frei
  - sicherer Zustand: einer von drei Philosophen könnte essen
  - die Anforderung von P_4 wird akzeptiert
- P_5 fordert ein Stäbchen an $\rightarrow$ kein Stäbchen wäre dann mehr frei
  - unsicherer Zustand: keiner der Philosophen könnte essen
  - die Anforderung von P_5 wird abgelehnt, P_5 muss warten
- haben vier Philosophen je ein Stäbchen, wird der fünfte gestoppt

Sicherer/Unsicherer Zustand (Forts.)

Leitungsvermittlung

Ausgangspunkt ein Vermittlungsrechner mit 12 Kommunikationskanälen

- Prozess P_1 benötigt max. 10 Kanäle, P_2 vier und P_3 neun

Situation: P_1 belegt fünf Kanäle, P_2 und P_3 je zwei; drei Kanäle sind frei

- P_3 fordert einen Kanal an, zwei blieben frei $\rightarrow$ unsicherer Zustand
  - P_3 könnte noch sechs Kanäle anfordern: $6 > 2$
  - die Anforderung von P_3 wird abgelehnt, P_3 muss warten
- P_1 fordert zwei Kanäle an, einer bliebe frei $\rightarrow$ unsicherer Zustand
  - P_1 könnte noch drei Kanäle anfordern: $3 > 1$
  - die Anforderung von P_1 wird abgelehnt, P_1 muss warten
- sichere Prozessfolge: $P_2 \rightarrow P_1 \rightarrow P_3$

Verklemmungsfreiheit

Verhinderung unsicherer Zustände

sicherer Zustand ist, wenn eine Folge der Verarbeitung vorhandener Prozesse existiert, in der alle Betriebsmittelanforderungen erfüllbar sind

unsicherer Zustand ist, wenn eine solche Folge nicht existiert; Erkennung dieses Zustands z.B. durch:

- Betriebsmittelbelegungsgraph (engl. *resource allocation graph*)
  - damit Vorhersage über das Eintreten von Zyklen treffen $\leadsto O(n^2)$
  - bei jeder Betriebsmittelanforderung den Graphen überprüfen
- Bankiersalgorithmus (engl. *banker's algorithm* [5])
  1. Prozesse beenden ihre Operationen in endlicher Zeit
  2. Betriebsmittelbedarf aller Prozesse übersteigt Gesamtvorrat nicht
  3. Prozesse definieren einen verbindlichen Kreditrahmen
  4. Betriebsmittelzuteilung erfolgt variabel innerhalb dieses Rahmens
- die Verfahrensweisen führen Prozesse dem *long-term scheduling* zu

Verklemmungserkennung
(engl. *deadlock detection*)

Verklemmungen werden (stillschweigend) in Kauf genommen...

- nichts im System verhindert das Auftreten von Wartezyklen
- keine der vier Bedingungen wird entkräftet

Ansatz: Wartegraph erstellen und auf Zyklen hin untersuchen $\leadsto O(n^2)$

- zu häufige Überprüfung verschwendet Betriebsmittel/Rechenleistung
- zu seltene Überprüfung lässt Betriebsmittel brach liegen

Zyklensuche geschieht zumeist in großen Zeitabständen, wenn...

- Betriebsmittelanforderungen zu lange andauern
- die Auslastung der CPU trotz Prozesszunahme sinkt
- die CPU bereits über einen sehr langen Zeitraum untätig ist

Verklemmungsauflösung

Erholungsphase nach der Erkennungsphase

Prozesse abbrechen und dadurch Betriebsmittel frei bekommen

- verklemmte Prozesse schrittweise abbrechen (gr. Aufwand)
  - mit dem „effektivsten Opfer“ (?) beginnen
- alle verklemmten Prozesse terminieren (gr. Schaden)

Betriebsmittel entziehen und mit dem „effektivsten Opfer“ (?) beginnen

- der betreffende Prozess ist zurückzufahren/wieder aufzusetzen
  - Transaktionen, *checkpointing/recovery* (gr. Aufwand)
- ein Aushungern der zurückgefahrenen Prozesse ist zu vermeiden

Gratwanderung zwischen Schaden und Aufwand:

- Schäden sind unvermeidbar und die Frage ist, wie sie sich auswirken

Nachlese ...

Verfahren zum Vermeiden/Erkennen sind eher praxisirrelevant

- sie sind kaum umzusetzen, zu aufwändig und damit nicht einsetzbar
- zudem macht die Vorherrschaft sequentieller Programmierung diese Verfahren wenig notwendig

Verklemmungsgefahr ist lösbar durch Virtualisierung von Betriebsmitteln

- Prozesse beanspruchen/belegen nur logische Betriebsmittel
- der Trick besteht darin, in kritischen Momenten den Prozessen (ohne ihr Wissen) physische Betriebsmittel entziehen zu können
- dadurch wird die Bedingung der Nichtentziehbarkeit entkräftet

☞ eher praxisrelevant/verbreitet sind die Vorbeugungsmaßnahmen

Zusammenfassung

- Verklemmung bedeutet „*deadlock*“ oder „*livelock*“
  - „[...] einen Zustand, in dem die beteiligten Prozesse wechselseitig auf den Eintritt von Bedingungen warten, die nur durch andere Prozesse in dieser Gruppe selbst hergestellt werden können“ [3]
  - dabei ist der *Lifelock* das größere Problem beider Verklemmungsarten
- für eine Verklemmung müssen vier Bedingungen gleichzeitig gelten
  - exklusive Belegung, Nachforderung, kein Entzug von Betriebsmitteln
  - zirkulares Warten der die Betriebsmittel beanspruchenden Prozesse
- Verklemmungsbekämpfung meint: Vorbeugen, Vermeiden, Erkennen
  - die Verfahren können im Mix zum Einsatz kommen

Literaturverzeichnis

[1] Michael Philippsen.
Parallele und Funktionale Programmierung.
http://www2.cs.fau.de/Lehre/SS2008/PFP/material/, 2008.
Lecture Notes.

[2] Victoria E. Neufeld, editor.
Webster's New World Dictionary.
Simon & Schuster, Inc., third college edition, 1988.

[3] Jürgen Nehmer and Peter Sturm.
Systemsoftware: Grundlagen moderner Betriebssysteme.
dpunkt.Verlag GmbH, zweite edition, 2001.

[4] Edsger Wybe Dijkstra.
Hierarchical ordering of sequential processes.
Acta Informatica, 1(2):115–138, June 1971.

Literaturverzeichnis (Forts.)

[5] Edsger Wybe Dijkstra.
Cooperating sequential processes.
Technical report, Technische Universiteit Eindhoven, Eindhoven, The Netherlands, 1965.
(Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New York, NY, 1996).