Systemprogrammierung

Grundlagen von Betriebssystemen

Teil C – XI. Stillstand

Wolfgang Schröder-Preikschat

13. Dezember 2022

Agenda

Einführung

Betriebsmittel
- Systematik
- Verwaltung

Systemblockade
- Grundlagen
- Fallbeispiel
- Gegenmaßnahmen

Zusammenfassung

Gliederung

Lehrstoff

- über den **Stillstand** (*stalemate*) gekoppelter Prozesse, hervorgerufen durch fehlkonstruierte oder -geleitete Betriebsmittelzuteilung
  - überkreuzte Anforderungen von Betriebsmitteln
  - „verlorene Abgabe“ produzierter oder im Voraus erworbener Ressourcen
- eine **tödliche Umarmung** (*deadly embrace* [2, S. 73]) solcher direkt oder indirekt voneinander abhängigen Prozesse
  - bedingt durch **Entwurfsfehler**, zu beheben durch Entwurfsänderungen
  - der Schwerpunkt (der Vorlesung) liegt auf **konstruktive Maßnahmen**
- verschiedene Facetten von Verklemmungen, je nach Ausprägung des Wartezustands der gekoppelten Prozesse
  - **Totsperre** (*deadlock*) als kleineres Übel, da erkennbar
  - **Lebendsperre** (*livelock*) als größeres Übel, da nicht erkennbar
- **Gegenmaßnahmen** sind Vorbeugung, Vermeidung oder Erkennung und Erholung von Systemblockaden
  - wobei Vorbeugung als konstruktive Maßnahme verbreitet ist, die anderen (analytische Maßnahmen) dagegen nur bedingt umzusetzen sind

Verklemmung, Blockierung, Stillstand

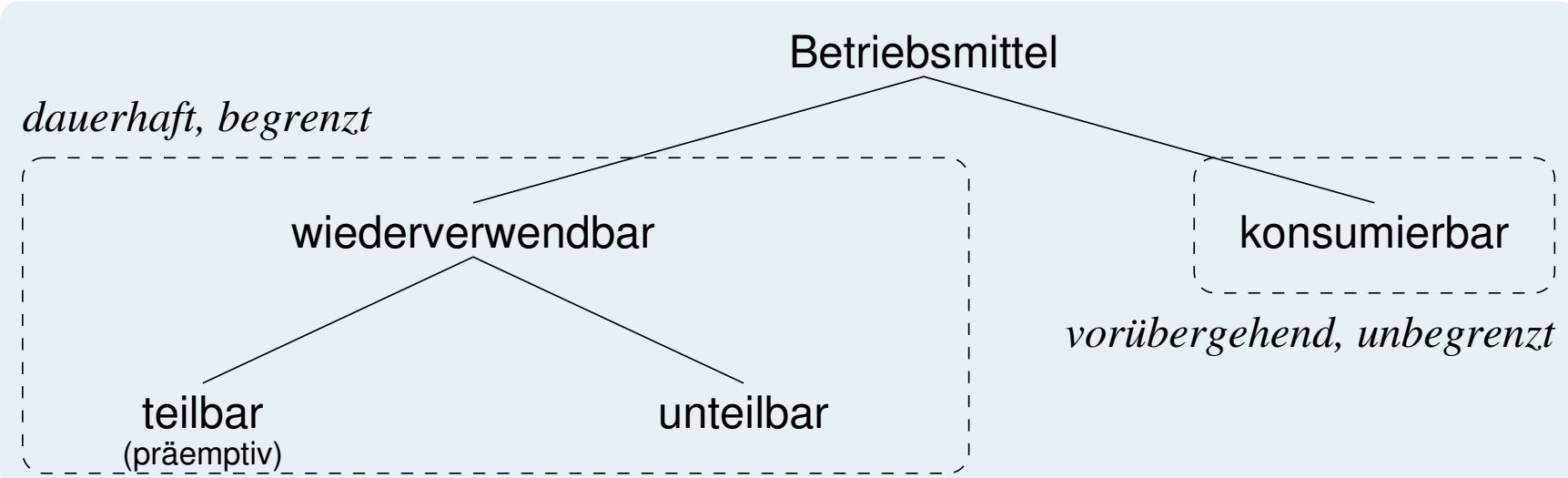
deadlock

source: National Geographic

Gliederung

Betriebsmittelklassifikation

nach [4, 5]



- alle Betriebsmittel werden angefordert, zugeteilt, belegt, benutzt und freigegeben, jedoch wird dabei wie folgt unterschieden:

  wiederverwendbar
  - sie sind (ggf. zeitweilig) **persistent**, nicht flüchtig
  - Anforderung durch mehrseitige Synchronisation
  - Freigabe ermöglicht ihre Wiederverwendung

  konsumierbar
  - sie sind **transient**, flüchtig
  - Anforderung durch einseitige Synchronisation
  - Freigabe führt zu ihrer Entsorgung (Zerstörung)

- entsprechend gestaltet sich der Wettbewerb um sie (*resource contention*)

Ziele

- konfliktfreie Abwicklung der anstehenden Aufträge
- korrekte Bearbeitung der Aufträge in endlicher Zeit
- gleichmäßige, maximierte Auslastung der Betriebsmittel
- hoher Durchsatz, kurze Durchlaufzeit, hohe Ausfallsicherheit

⋮

- **Betriebsmittelanforderung** frei von Verhungern/Verklemmung

  Verhungern
  - andauernde (zeitweilige) Benachteiligung von Prozessen
  - das Prozesssystem macht Fortschritt, steht nicht still

  Verklemmung
  - irreversible gegenseitige Blockierung von Prozessen
  - das Prozesssystem macht keinen Fortschritt, steht still

  - gemeinsames Merkmal ist der **Wettbewerb** um Betriebsmittelzuteilung

- allgemein:
  - Durchsetzung der vorgegebenen Betriebsstrategie
  - eine optimale Realisierung in Bezug auf relevante Kriterien

Aufgaben

- **Buchführung** über die im Rechensystem vorhandenen Betriebsmittel
  - Art, Klasse
  - Zugriffsrechte, Prozesszuordnung, Nutzungszustand und -dauer
- **Steuerung** der Verarbeitung von Betriebsmittelanforderungen
  - Entgegennahme, Überprüfung (z.B. der Zugriffsrechte)
  - Einplanung der Nutzung angeforderter Betriebsmittel durch Prozesse
  - Einlastung (Zuteilung) von Betriebsmittel
  - Entzug oder Freigabe von Prozessen benutzter Betriebsmittel

- **Betriebsmittelentzug**
  - Zurücknahme (*revocation*) der Betriebsmittel, die von einem „aus dem Ruder geratenen“ Prozess belegt werden
  - bei **Virtualisierung** zusätzlich:
    - Rückforderung und Neuzuteilung eines realen Betriebsmittels
    - wobei das zugehörige virtuelle Betriebsmittel dem Prozess zugeteilt bleibt

Verfahrensweisen

- **statisch**
  - vor Laufzeit oder vor einem Laufzeitabschnitt
  - Anforderung aller (im Abschnitt) benötigten Betriebsmittel
  - Zuteilung der Betriebsmittel erfolgt ggf. lange vor ihrer eigentlichen Benutzung
  - Freigabe aller belegten Betriebsmittel mit Laufzeit(abschnitt)ende

↪ Risiko einer nur suboptimalen Auslastung der Betriebsmittel

- **dynamisch**
  - zur Laufzeit, in beliebigen Laufzeitabschnitten
  - Anforderung des jeweils benötigten Betriebsmittels bei Bedarf
  - Zuteilung des jeweiligen Betriebsmittels erfolgt „im Moment“ seiner Benutzung
  - Freigabe eines belegten Betriebsmittels, sobald kein Bedarf mehr besteht

↪ Risiko der **Verklemmung** von abhängigen Prozessen

Gliederung

Stillstand von Prozessen

Definition (deadly embrace)

Eine Situation, in der gekoppelte Prozesse gegenseitig die Aufhebung einer Wartebedingung entgegensehen, diese aber durch Prozesse eben dieses Systems selbst aufgehoben werden müsste.

- die Bedingung sagt etwas zur Verfügbarkeit eines Betriebsmittels aus
  - unabhängig von der Art des Betriebsmittels erwarten gekoppelte Prozesse die Versorgung durch entsprechende Aktionen gleichgestellter Prozesse
  - da jedoch alle Prozesse so handeln, wird kein Betriebsmittel verfügbar
- nach [1] kann die „tödliche Umarmung“ von Prozessen entstehen:
  - i obwohl kein einziger Prozess mehr als die insgesamt verfügbare Menge von Betriebsmitteln benötigt und
  - ii unabhängig davon, ob Betriebsmittelzuteilung in der Verantwortlichkeit des Betriebssystems oder des Anwendungsprogramms selbst liegt
- das Warten kann **inaktiv** (*deadlock*) oder **aktiv** (*livelock*) geschehen
  - d.h., mit oder ohne Abgabe des Betriebsmittels „Prozessor“

Wartezustand „inaktiv“

Definition ([7, S. 235] ↦ zeitabhängiges Fehlverhalten)

Mit **Verklemmung** (*deadlock*) bezeichnet man 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.

- die „tödliche Umarmung“ gekoppelter Prozesse im **Schlafzustand**
  - die Befehlszähler der verklemmten Prozesse bleiben (für die meiste Zeit) konstant und warten meint für den einzelnen Prozess:

tief
  - er ist im Zustand „blockiert“, das erwartete Ereignis ist definiert
  - er gibt den Prozessor zu Gunsten anderer Prozesse ab
    - mit Ausnahme eines „blockiert“ fortschreitenden Leerlaufprozesses
    - der Prozessor ist in Wartestellung bis ein Prozess „bereit“ wird

- **gutartig**, das kleinere von zwei (inaktiv, aktiv) Übeln
  - wenn nicht vorgebeugt oder vermieden, so kann das erkannt werden
  - sofern das Ereignis, worauf ein Prozess wartet, bekannt/definiert ist
  - Abgrenzung von sich in Bewegung befindlichen Prozessen ist machbar

Wartezustand „aktiv“

livelock

Definition (Lebendsperre)

Eine der Verklemmung sehr ähnliche Situation, in der gekoppelte Prozess zwar nicht im Zustand „blockiert“ sind, sie jedoch dennoch keinen Fortschritt bei der Programmausführung erzielen.

- die „tödliche Umarmung“ gekoppelter Prozesse im **Wachzustand**
  - die Befehlszähler der verklemmten Prozesse verändern sich fortwährend und warten meint für den einzelnen Prozess:

beschäftigt
- er bleibt im Zustand „laufend“, hält den Prozessor

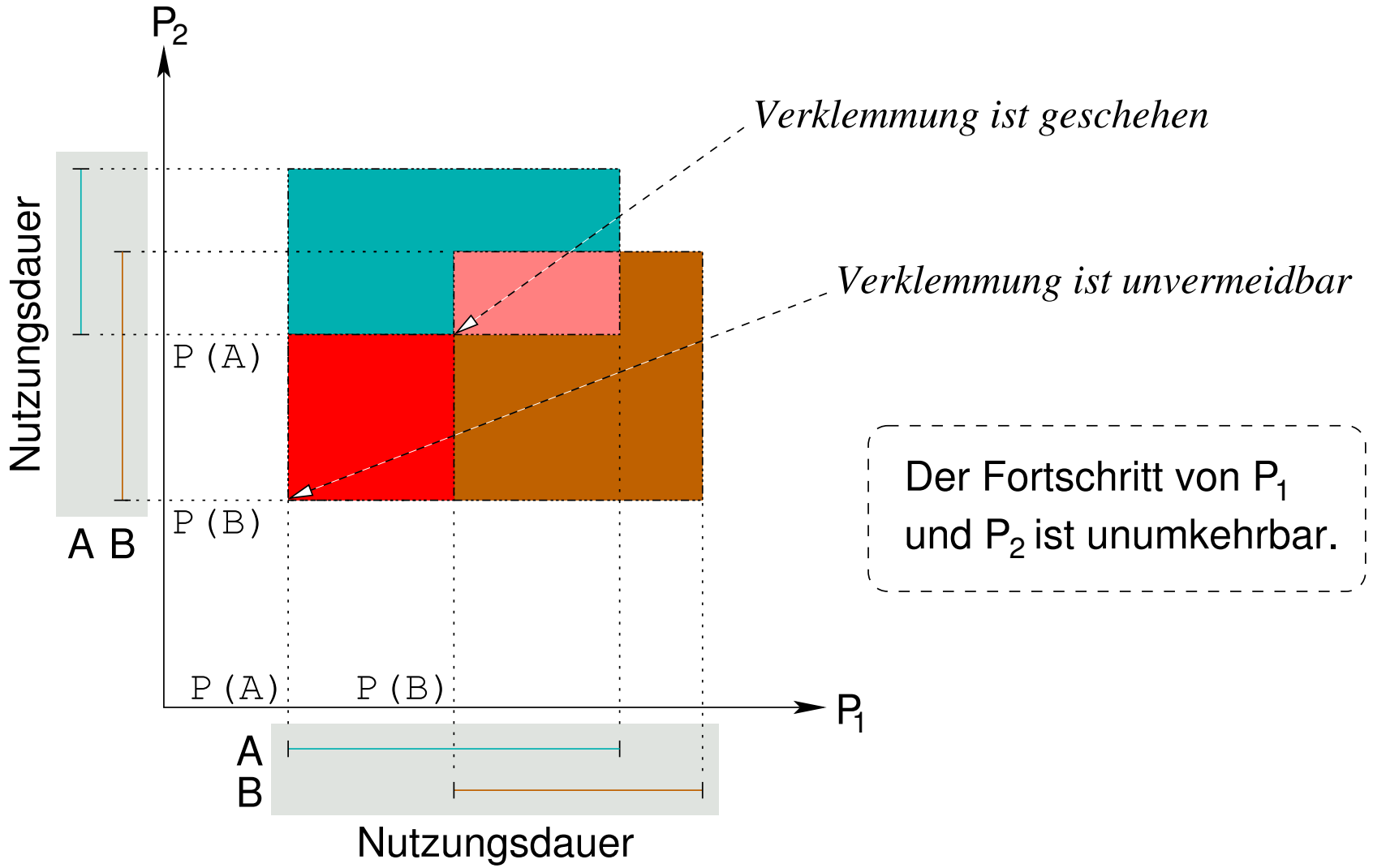
träge
- er alterniert zwischen den Zuständen „laufend“ und „bereit“
- er gibt den Prozessor zu Gunsten anderer Prozesse ab

- **bösartig**, das größere von zwei (inaktiv, aktiv) Übeln
  - wenn nicht vorgebeugt oder vermieden, so kann das nicht erkannt werden
  - keine Abgrenzung von normalen, voranschreitenden Prozessen[1]

[1] Wie häufig und lange sollte etwa geprüft werden, dass sich Befehlszähler von Prozessen in welchen Wertebereichen bewegen?

Zustandsraum

nach [1, p. 70/71]



Der Fortschritt von P_1 und P_2 ist unumkehrbar.

Kontoführung

- **Banküberweisung** (*bank transfer*) eines gewissen Betrags von einem Konto auf ein anderes Konto

```
1  void transfer(account_t *from, account_t *to, double amount) {
2      claim(from, to);          /* acquire account data records */
3      from->value -= amount;    /* withdraw from one account */
4      to->value += amount;      /* credit the other account */
5      clear(from, to);          /* release account data records */
6  }
```

  - dabei seien die Konten **wiederverwendbare Betriebsmittel** (Software)
  - die vom Überweisungsprozess zeitweilig **unteilbar** benutzt werden müssen

- für die Überweisungsoperation sei folgender **Datensatz** (*data record*) eines Kontos angenommen:

```
7  typedef struct account {
8      double value;          /* actual balance */
9      semaphore_t count;     /* safeguard for account management */
10 } account_t;
```

  - **wechelseitiger Ausschluss** sichert den Überweisungsprozess ab
  - hierzu wird ein **zählender Semaphor** mit Initialwert 1 verwendet

Betriebsmittelverwaltung

- **Inanspruchnahme** der für den Transfer benötigten Datensatzobjekte:

```
1  void claim(account_t *from, account_t *to) {
2      P(&from->count);      /* acquire source account */
3      P(&to->count);        /* acquire target account */
4  }
```

  - jedes dieser Objekte ist nur in einfacher Ausfertigung verfügbar
  - zu einem Zeitpunkt wird nur ein Prozess ein Exemplar belegen können

- **Bereitstelllung** der beiden Objekte zur Wiederverwendung:

```
5  void clear(account_t *from, account_t *to) {
6      V(&to->count);        /* release target account */
7      V(&from->count);      /* release source account */
8  }
```

  - Programmierkonvention sei paarweise Verwendung von `claim` und `clear`
  - d.h., es werden nie mehr Objekte bereitgestellt als angefordert wurden

- in dem Szenario verbirgt sch eine **wettlaufkritische Aktionsfolge**
  - die zur Verklemmung von „überweisungswilligen“ Prozessen führen kann

Wettlaufkritische Aktionsfolge

- Ausgangssituation:
  - sei P_i ein Überweisungsprozess, wobei $1 \leq i \leq 3$ verschiedene von diesen Prozessen gleichzeitig geschehen
  - sei K_j, $j > 1$, ein Konto, jeweils repräsentiert durch ein Datensatzobjekt
- Szenario (mit α, β und γ beliebige Geldbeträge):

$P_1 : \mathit{transfer}(K_1, K_2, \alpha)$	durchläuft `claim`, belegt K_1, wird verdrängt
$P_2 : \mathit{transfer}(K_2, K_3, \beta)$	durchläuft `claim`, belegt K_2, wird verdrängt
$P_3 : \mathit{transfer}(K_3, K_1, \gamma)$	durchläuft `claim`, belegt K_3, wird verdrängt
$P_1 : \mathit{transfer}(K_1, K_2, \alpha)$	fährt fort, fordert $K_2 \mapsto P_2$ an, blockiert
$P_2 : \mathit{transfer}(K_2, K_3, \beta)$	fährt fort, fordert $K_3 \mapsto P_3$ an, blockiert
$P_3 : \mathit{transfer}(K_3, K_1, \gamma)$	fährt fort, fordert $K_1 \mapsto P_1$ an, blockiert

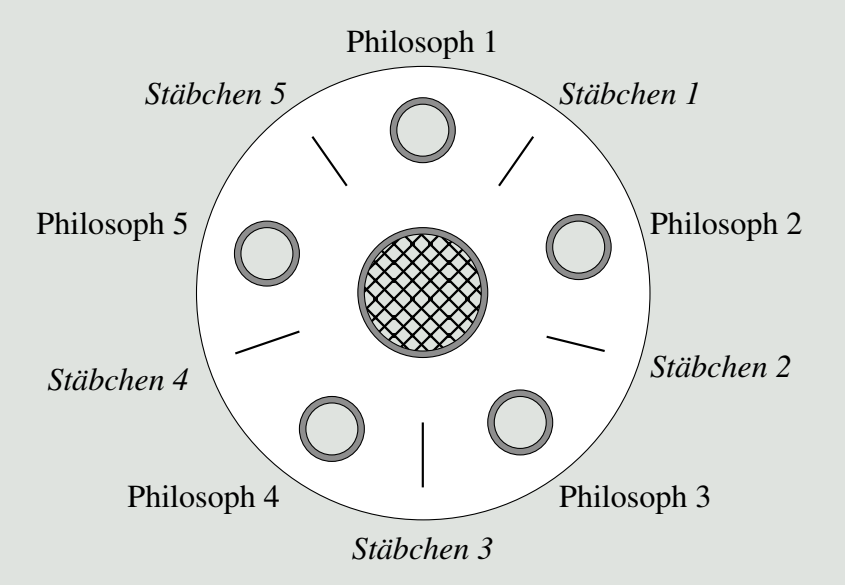
- diese Aktionsfolge bedeutet also:
  - P_1 wartet auf die Zuteilung von K_2, das von P_2 belegt wird
  - P_2 wartet auf die Zuteilung von K_3, das von P_3 belegt wird
  - P_3 wartet auf die Zuteilung von K_1, das von P_1 belegt wird
- d.h., P_1, P_2 und P_3 sind untereinander verklemmt ☹
  - keiner der Prozesse kann sich von selbst aus dieser Situation befreien...

Speisende Philosophen I

Hinweis

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.

- Philosoph
  - Prozess
  - ↪ Überweisung
- Stäbchen
  - Betriebsmittel
  - ↪ Konto



- **Stillstand**:
  - alle Philosophen nehmen zugleich das eine (linke) Stäbchen auf
  - anschließend greifen sie auf das andere (rechte) zu

↪ die Überweisungsprozesse fordern zugleich das jeweilige Quellkonto an, belegen es und fordern anschließend das Zielkonto an

Speisende Philosophen II

- Philosophendasein:

```
1  void phil(int who) {
2    int any = check(who);
3    while (any > 0) {
4      think();
5      claim(any);
6      munch();
7      clear(any);
8    }
9  }

10 void think() { ... }
11 void munch() { ... }
```

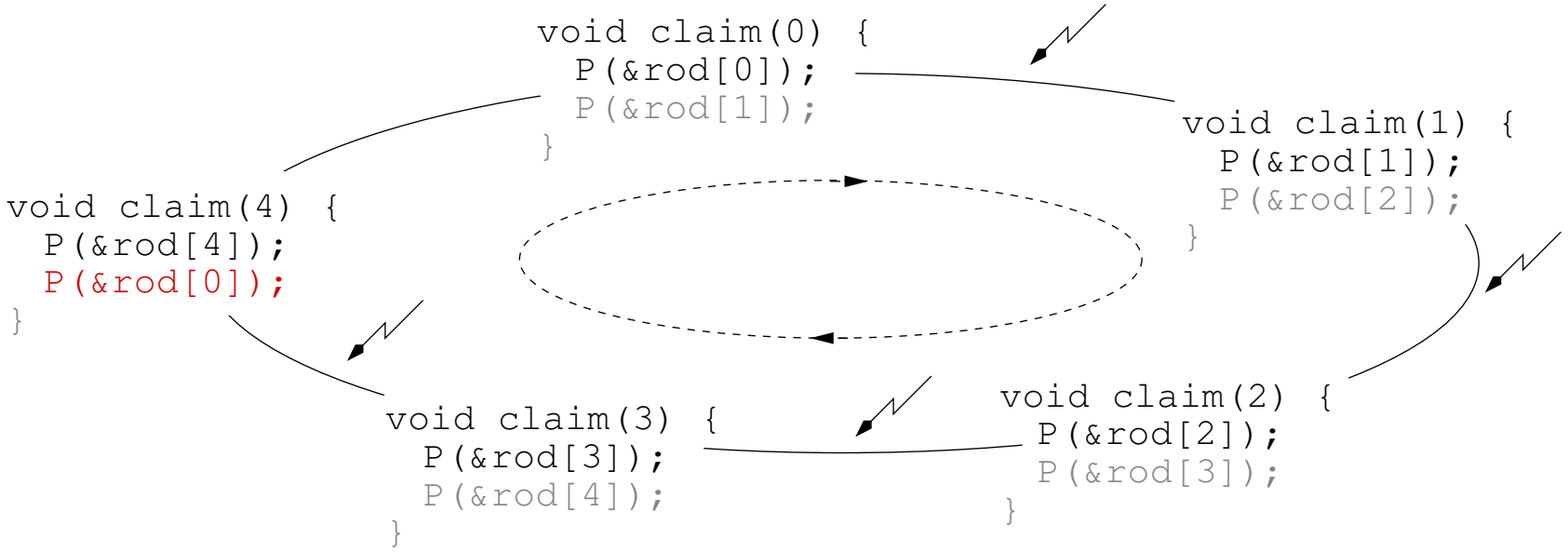
- altbekanntes Muster der Betriebsmittelzuteilung
  - vgl. S. 17
  - *P* vergibt zu einem Zeitpunkt nur ein Stäbchen (*rod*), umgekehrt gibt *V* entsprechend nur eins frei

- Stäbchenverwaltung:

```
12 semaphore_t rod[NPHIL] = {
13     { 1 }, { 1 }, { 1 }, { 1 }, { 1 }
14 };
15
16 int check(int who) {
17     ...          /* validate who */
18     return who - 1;
19 }
20 void claim(unsigned slot) {
21     P(&rod[slot]);
22     P(&rod[(slot + 1) % NPHIL]);
23 }
24
25 void clear(unsigned slot) {
26     V(&rod[(slot + 1) % NPHIL]);
27     V(&rod[slot]);
28 }
```

Speisende Philosophen III

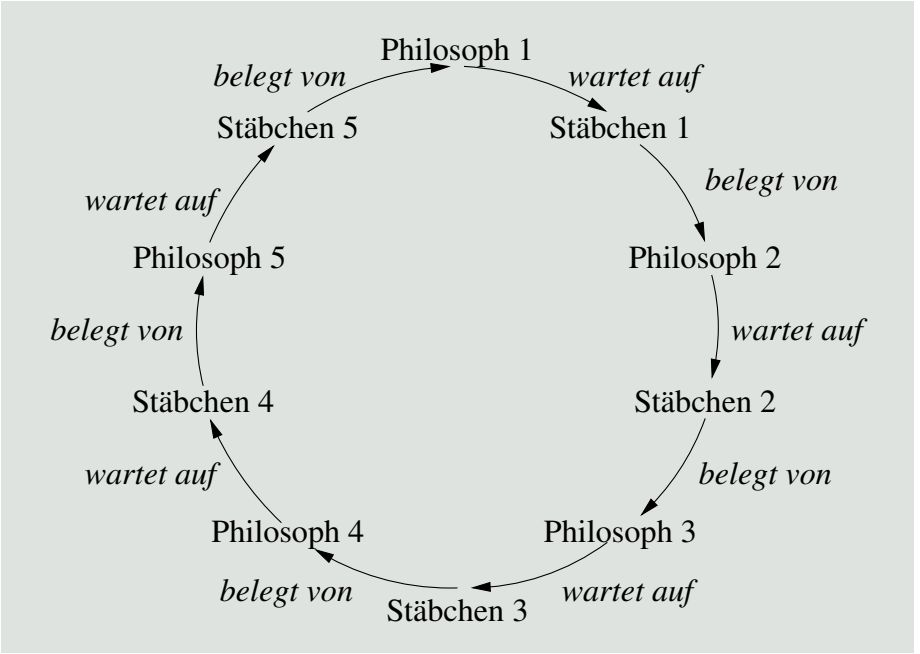
- sei P_i, $1 \leq i \leq 5$, Philosoph i, der Stäbchen S_i und S_{i+1} benötigt
  - mit $r = i - 1$ als Index im Wertebereich $[0, 4]$ für das Stäbchenfeld *rod*



1. P_1 nimmt $S_1 \mapsto r_0$ auf, wird vor Aufnahme von $S_2 \mapsto r_1$ gestört
2. P_2 nimmt $S_2 \mapsto r_1$ auf, wird vor Aufnahme von $S_3 \mapsto r_2$ gestört
3. P_3 nimmt $S_3 \mapsto r_2$ auf, wird vor Aufnahme von $S_4 \mapsto r_3$ gestört
4. P_4 nimmt $S_4 \mapsto r_3$ auf, wird vor Aufnahme von $S_5 \mapsto r_4$ gestört
5. P_5 nimmt $S_5 \mapsto r_4$ auf, fordert $S_1 \mapsto r_0$ an und muss warten
   - in Folge fordern alle anderen P_i, $1 \leq i < 4$, ihr zweites Stäbchen an... ☹

Speisende Philosophen IV

- für jedes belegte Betriebsmittel ist der **Eigentümer** bekannt
- für jeden wartenden Prozess ist der **Blockadegrund** bekannt
- der **Wartegraph** (S. 31) zeigt die Verklemmungssituation
  - ist das Ergebnis der Analyse der Betriebsmittelbelegung
  - wird im Bedarfsfall aufgebaut



↪ ein geschlossener Kreis (im Wartegraphen) erfasst all jene 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

Voraussetzungen für Verklemmungen

nach [1, p. 70]

Hinweis (Lebendsperre)

Nachfolgendes gilt auch für Stillstand im aktiven Wartezustand.

- **notwendige Bedingungen** für die gekoppelten Prozesse:
  1. **Ausschließlichkeit** in der Betriebsmittelnutzung (*mutual exclusion*)
  2. **Nachforderung** eines oder mehrerer Betriebsmittel (*hold and wait*)
  3. **Unentziehbarkeit** (*no preemption*) der zugeteilten Betriebsmittel
- **notwendige und hinreichende Bedingung**:
  4. **zirkulares Warten** muss eingetreten sein
     - jeder der Prozesse hält eins oder mehrere Betriebsmittel, die einer oder mehrere andere Prozesse in der Kette angefordert haben

Hinweis (Vorbeugung/Vemeidung)

*Jede dieser Bedingungen muss zu einem Zeitpunkt erfüllt sein, damit Prozesse verklemmen. Die Aufhebung nur einer dieser Bedingungen resultiert in ein **verklemmungsfreies System** von Prozessen.*

Verklemmungsvorbeugung

deadlock prevention

Hinweis (Prävention)

Vorsorgemaßnahmen, damit gekoppelte Prozesse gar nicht erst eine Verklemmung entwickeln können.

- **indirekte Methoden**, eine der notwendigen Bedingungen aufheben
  1. nichtblockierende Synchronisation, Atomarität auf tieferer Ebene nutzen
  2. alle benötigten Betriebsmittel unteilbar anfordern
  3. Betriebsmittel virtualisieren, damit den Entzug der realen Betriebsmittel ermöglichen, nicht jedoch ihrer virtuellen Gegenstücke
- **direkte Methoden**, notwendige & hinreichende Bedingung aufheben
  4. eine lineare Ordnung von Betriebsmittelklassen definieren, die zusichert, dass Betriebsmittel R_i vor R_j zuteilbar ist, nur wenn $i < j$

Hinweis (Prophylaxe)

*Jede dieser Methoden steht für eine **konstruktive Maßnahme**, die Aufbau und Struktur nichtsequentieller Programme beeinflusst.*

Betriebsmittel unteilbar anfordern

- Beispiel zum Bankwesen (vgl. S. 17):

```
1  void claim(account_t *from, account_t *to) {
2      static semaphore_t mutex = {1};
3      P(&mutex);                    /* enter critical section */
4      P(&from->count);              /* demand */
5      P(&to->count);                /* additional demand */
6      V(&mutex);                    /* leave critical section */
7  }
```

- Beispiel zum Philosophenproblem (vgl. S. 20):

```
1  void claim(unsigned slot) {
2      static semaphore_t mutex = {1};
3      P(&mutex);                    /* enter critical section */
4      P(&rod[slot]);                /* demand */
5      P(&rod[(slot + 1) % NPHIL]);/* additional demand */
6      V(&mutex);                    /* leave critical section */
7  }
```

- die **Nachforderungsbedingung** wurde insofern aufgehoben, als dass An- und Nachforderung (Z. 4–5) nunmehr unteilbar geschehen

Atomarität auf tieferer Ebene nutzen

- Beispiel zum Bankwesen (vgl. S. 16):
  - Umstrukturierung, **prozedurale Abstraktion** der kritischen Aktionsfolge:

```
1 void transfer(account_t *from, account_t *to, double amount) {
2     issue(from, -amount);    /* withdraw from one account */
3     issue(to, amount);       /* credit the other account */
4 }
```

  - Abbildung auf eine problemspezifische **Elementaroperation**:

```
5 inline void issue(account_t *this, double amount) {
6     FAA(&this->balance, amount);
7 }
```

  - Reduktion auf eine atomare, in GCC eingebaute intrinsische Funktion:

```
8 #define FAA __sync_fetch_and_add
```

- die **Nachforderungsbedingung** wurde abgebildet auf nur noch eine Anforderung des Betriebsmittels „kritischer Abschnitt“: **FAA**
  - die Verwendung eines binären Semaphors anstelle von FAA ginge ebenso, jedoch wäre dies bei weitem nicht so effizient, wie mit FAA

↪ vollständig semantisch äquivalent im Vergleich zur Vorlage (S. 16) ist die Lösung jedoch nicht, da Datensatzobjekte ungeschützt sind

Verklemmungsvermeidung

deadlock avoidance

Hinweis

***Vorabwissen** zu Prozessen und ihren Betriebsmittelanforderungen.*

- Vereitelung „tödlicher Umarmung“ durch **strategische Methoden**
  - kein Versuch wird unternommen, eine notwendige Bedingung aufzuheben
  - vielmehr verhindert **laufende Anforderungsanalyse** zirkulares Warten
- Prozesse und ihre Betriebsmittelanforderungen werden überwacht
  - jede Anforderung prüft auf einen möglichen **unsicheren Zustand**
  - sollte ein solcher möglich sein, wird die **Zuteilung abgelehnt**
  - anfordernden Prozess suspendieren $\rightsquigarrow$ langfristige Planung [6, S. 19]
- Betriebsmittelzuteilung erfolgt nur im **sicheren Zustand**
  - im Falle einer **Prozessfolge**, die alle zukünftigen Anforderungen erfüllt
  - in Anbetracht aller aktuellen Belegungen und anstehenden Freigaben

Hinweis (Vermeidung)

*Jede Methode, die Verklemmungen „vermeidet“, ist eine **analytische Maßnahme** zur Laufzeit nichtsequentieller Programme.*

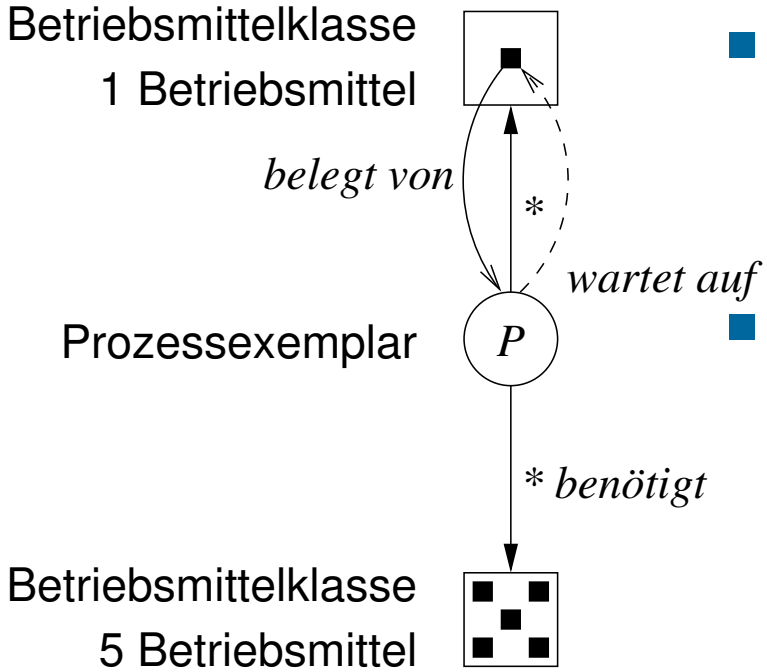
Bestimmung des unsicheren Zustands

- Ansatz **Betriebsmittelgraph** (S. 29)
  - definiert einen **Mengenkontrakt** für Prozessexemplare bezüglich Bedarf und aktueller Vergabe von Betriebsmitteln bestimmter Klassen
  - mit Vorabwissen angelegt und initialisiert bei der Prozesserzeugung und fortgeschrieben mit jeder Betriebsmittelanforderung
  - anhaltende Analyse hinsichtlich möglicher **Zyklenbildung** im Graphen
- Ansatz **Bankiersalgorithmus** (*banker's algorithm* [2])
  - Annahme ist, dass das System die Menge von Betriebsmitteln kennt, die:
    - i jeder Prozess möglicherweise anfordert (*maximum claim*: Bedarf),
    - ii jeder Prozess gegenwärtig hält (*allocated*: Belegung) und
    - iii noch nicht allen Prozessen zugewiesen wurde (*available*: Verfügungsrahmen)
  - **sicherer Zustand** ist, wenn eine Anforderung folgendes nicht übersteigt:
    - (a) den Bedarf des anfragenden Prozesses und
    - (b) den Verfügungsrahmen von Betriebsmitteln der angefragten Klasse
  - scheitert (a), wird die Anforderung zurückgewiesen, scheitert (b), wird der anfordernde Prozess suspendiert $\rightsquigarrow$ langfristige Planung [6, S. 19]
- nicht nur die Erfordernis von **Vorabwissen** erweist sich als ein großes Problem, auch die Skalierbarkeit von Methode und Implementierung

Betriebsmittelgraph

resource allocation graph, RAG

- ein **gerichteter Graph**, der Prozessexemplare und Betriebsmittel oder Betriebsmittelklassen zusammenhängend darstellt
  - eine vom Betriebssystem zu verwaltende **dynamische Datenstruktur**



  - optionales (Vorab-) Wissen, um die *benötigt*-Beziehung zu bilden:
    - Betriebsmittelklassen und den jeweiligen Betriebsmittelbedarf
  - obligatorisches Wissen bezüglich aller Prozesse/Betriebsmittel:
    - für jeden Prozess gibt es eine Liste zugeteilter Betriebsmittel (*benötigt*)
    - jedes Betriebsmittel verbucht den Besitzerprozess (*belegt von*)
- schließlich noch (obligatorisches) Wissen, um aus dem RAG einen **Wartegraphen** abzuleiten zu können:
  - für jeden Prozess ist vermerkt, worauf er wartet (*wartet auf*)

Verklemmungsnachweis und Erholung *deadlock detection*

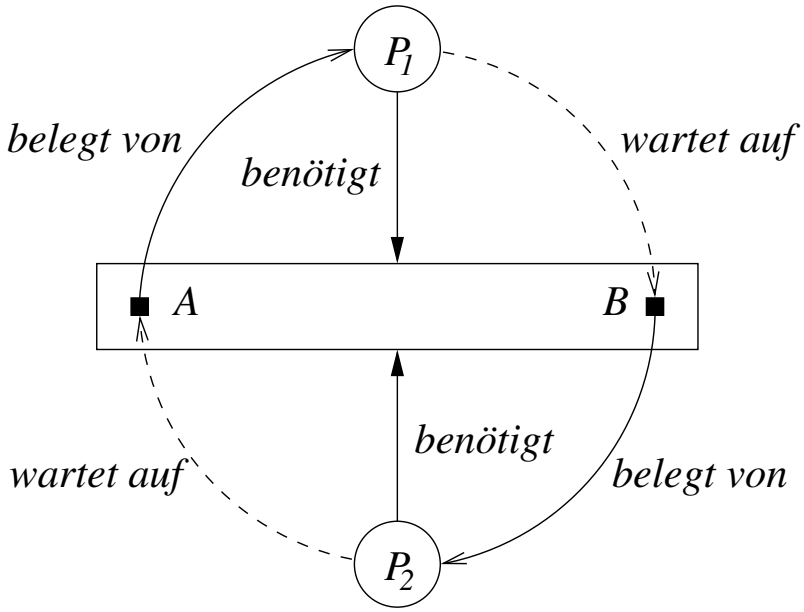
- Prozessverklemmungen werden stillschweigend in Kauf genommen
  - kein Versuch wird unternommen, eine der vier Bedingungen aufzuheben
  - stattdessen läuft die **sporadische Suche** nach blockierten Prozessen
    - ein **Wartegraph** (S. 31) wird aufgebaut und nach Zyklen abgesucht
    - Grundlage dafür bildet der **Betriebsmittelgraph** (S. 29)
- erkannte Zyklen werden im nachgeschalteten Schritt durchbrochen
  - eine Option ist die **Prozesszerstörung** eines einzelnen, ausgewählten Exemplars oder aller Exemplare im Zyklus
  - die andere Option ist **Betriebsmittelentzug**, durch Auswahl eines Opfers und anschließendem **Zurückrollen** des besitzenden Prozesses
- gegebenenfalls wiederholt sich der ganze Vorgang, solange nicht alle Zyklen aufgelöst werden konnten

Gratwanderung zwischen Schaden und Aufwand

Schaden macht klug, aber zu spät. (Sprichwort)

Wartegraph

wait-for graph



Hinweis (Erzeugung)

Wenn das Betriebssystem den Verklemmungsfall für wahrscheinlich hält:

- *Antwortzeitzunahme*
- *Durchsatzabnahme*
- *Leerlaufzeitanstieg*

- seien A und B Betriebsmittel derselben Klasse:
  1. P_1 vollzieht $P(A)$, A ist frei und wird P_1 zugeteilt
  2. P_2 vollzieht $P(B)$, B ist frei und wird P_2 zugeteilt
  3. P_1 vollzieht $P(B)$, B ist belegt $\rightsquigarrow$ P_1 muss auf $V(B)$ durch P_2 warten
  4. P_2 performs $P(A)$, A ist belegt $\rightsquigarrow$ P_2 muss auf $V(A)$ durch P_1 warten
- ein **Zyklus** von P_1 nach P_2 über A und B, hin und zurück
  - P_1 und P_2 befinden sich im ***Deadlock***...

Gliederung

Resümee

- **Betriebsmittel** zeigen sich als Entitäten von Hardware und Software
  - wiederverwendbar
    - begrenzt verfügbar: teilbar, unteilbar
  - konsumierbar
    - unbegrenzt verfügbar
- Ziele, Aufgaben und Verfahrensweise der **Betriebsmittelverwaltung**
  - Betriebsmittelanforderung frei von Verhungern/Verklemmung
  - Buchführung der Betriebsmittel, Steuerung der Anforderungen
  - statische/dynamische Zuteilung von Betriebsmitteln
- für eine Verklemmung müssen **vier Bedingungen** gleichzeitig gelten
  - exklusive Belegung, Nachforderung, kein Entzug von Betriebsmitteln
  - zirkulares Warten der die Betriebsmittel beanspruchenden Prozesse
  - nicht zu vergessen: Verklemmung bedeutet „*deadlock*“ oder „*livelock*“
- die **Gegenmaßnahmen** sind:
  - Vorbeugen, Vermeiden, Erkennen & Erholen
  - die Verfahren können im Mix zum Einsatz kommen
- Verfahren zum **Vermeiden/Erkennen** sind eher praxisirrelevant
  - sie sind kaum umzusetzen, zu aufwendig und damit schlecht einsetzbar
  - Vorherrschaft sequentieller Programmierung macht sie verzichtbar. . .

Literaturverzeichnis I

[1] Coffman, Jr., E. G. ; Elphick, M. J. ; Shoshani, A. :
System Deadlocks.
In: *Computing Surveys* 3 (1971), Jun., Nr. 2, S. 67–78

[2] Dijkstra, E. W.:
Cooperating Sequential Processes / Technische Universiteit Eindhoven.
Eindhoven, The Netherlands, 1965 (EWD-123). –
Forschungsbericht. –
(Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New York, NY, 1996)

[3] Habermann, A. N.:
Prevention of System Deadlocks.
In: *Communications of the ACM* 12 (1969), Jul., Nr. 7, S. 373–377/385

[4] Holt, R. C.:
On Deadlock in Computer Systems.
Ithaca, NY, USA, Cornell University, Diss., 1971

[5] Holt, R. C.:
Some Deadlock Properties of Computer Systems.
In: *ACM Computing Surveys* 4 (1972), Sept., Nr. 3, S. 179–196

Literaturverzeichnis II

[6] Kleinöder, J. ; Schröder-Preikschat, W. :
Einplanungsgrundlagen.
In: Lehrstuhl Informatik 4 (Hrsg.): *Systemprogrammierung*.
FAU Erlangen-Nürnberg, 2015 (Vorlesungsfolien), Kapitel 9.1

[7] Nehmer, J. ; Sturm, P. :
Systemsoftware: Grundlagen moderner Betriebssysteme.
dpunkt.Verlag GmbH, 2001. –
ISBN 3–898–64115–5

Sicherer Zustand

nach [3]

- sei P_k ein sequentieller Prozess
- sei S eine geordnete Menge solcher Prozesse
- sei b_k der Betriebsmittelanspruch eines Prozesses, P_k
- sei $s(k)$ die Ordnungszahl von $P_k \in S$
- sei $r(t)$ die Anzahl verfügbarer Betriebsmittel zum Zeitpunkt t
- sei $c_k(t)$ die Anzahl der P_k zur Zeit t zugeteilten Betriebsmittel
- dann gilt ein Zustand als sicher, wenn es eine vollständige Folge in S gibt, so dass:

$$\forall_{P_k \in S} b_k \leq r(t) + \sum_{s(l) \leq s(k)} c_l(t) \tag{1}$$

Condition (1) says that the claim by process P_k must not exceed the sum of the free resources and those resources which will become free "in due time," when the processes preceding in S have released theirs. [3, p. 375]