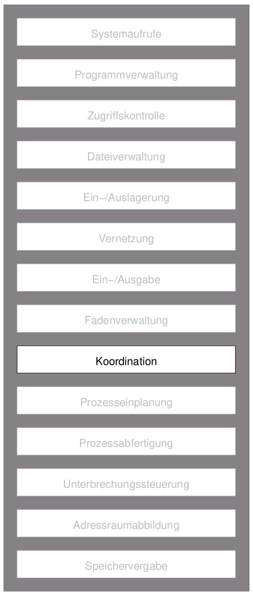
Verklemmung

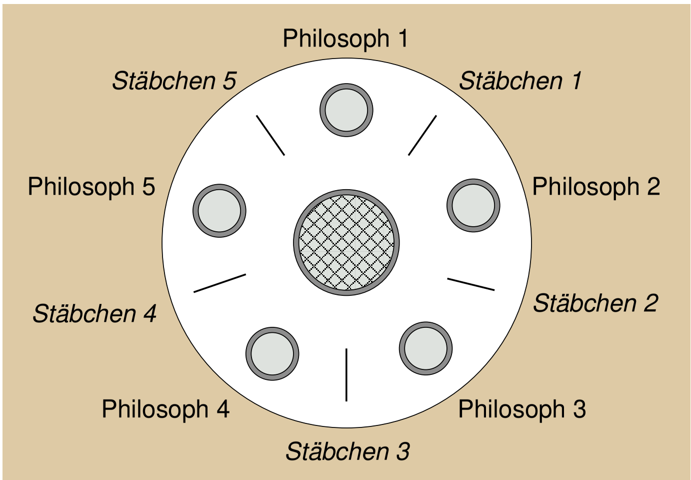


dead·lock 1 a standstill resulting from the action of equal and opposed forces; stalemate **2** a tie between opponents in the course of a contest **3** DEADBOLT — to bring or come to a deadlock

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.“ [2]

Speisende Philosophen (1) — 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.

Speisende Philosophen (2) — Synchronisationsaspekte

einseitige Synchronisation

- ein Philosoph muss warten, bis ihm ein Stäbchen zur Verfügung steht

mehrseitige Synchronisation

- keine zwei Philosophen dürfen gleichzeitig dasselbe Stäbchen aufnehmen

Verhalten

- keiner der Philosophen darf das eine (z. B. rechte) Stäbchen aufnehmen und dann auf das seines (in dem Fall dann linken) Nachbarns warten
- das wäre dann der Fall, wenn alle Philosophen „gleichzeitig“ das eine (z. B. rechte) Stäbchen aufnehmen würden

Speisende Philosophen (3) — Nebenläufiges Programm

```
void philosopher (unsigned who, Semaphore rod[], unsigned all) {
    for (;;) {
        think();
        eat(&rod[who], &rod[(who + 1) % all]);
    }
}
```

```
void think () {
    ⋮
}
```

```
void eat (Semaphore *rod1, Semaphore *rod2) {
    P(rod1);
    P(rod2);
    ⋮
    V(rod2);
    V(rod1);
}
```

Jeder speisende Philosoph wird durch einen Prozess (Faden) repräsentiert, der `philosopher()` ausführt und dazu eine Nummer (`who`, 1–5) „gezogen“ hat. Die Anzahl der Philosophen bzw. Stäbchen liefert `all` (5). Jedem Stäbchen ist ein Semaphor (`rod`) zugeordnet.

Speisende Philosophen (4) — *Race Hazard*

```
void eat (Semaphore *rod1, Semaphore *rod2) {
    P(rod1);
    P(rod2);
    ⋮
    V(rod2);
    V(rod1);
}
```

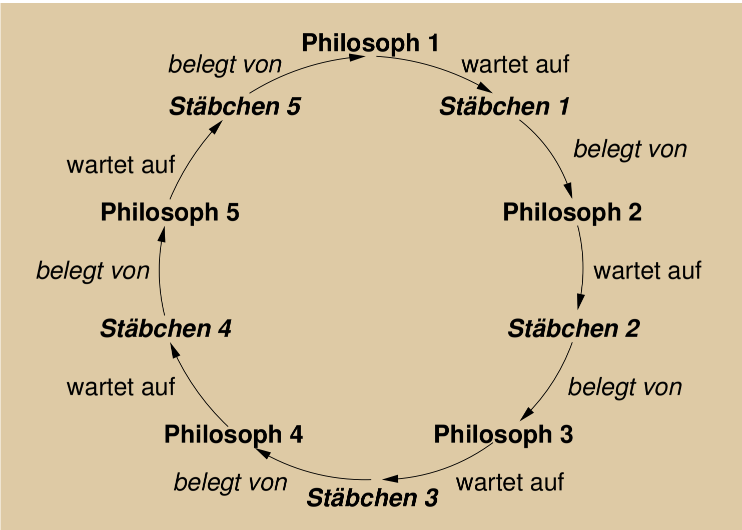
- ☞ Stab ≡ Betriebsmittel
- ☞ Philosoph ≡ Prozess
- ☞ zwei Stäbe ⇝ 2 × P()

Annahme: Zwischen dem ersten und zweiten P() wird der laufende Prozess verdrängt. Ein anderer Prozess ruft eat() auf, dem das gleiche Schicksal ereilt. Das Schema wiederholt sich — bis alle fünf Philosophen an der Reihe waren:

✓ jeder Philosoph hat das erste Stäbchen und wartet auf das zweite
✓ keiner gibt sein Stäbchen zurück, alle werden verhungern . . . . *Deadlock*

[Welche Art von Scheduling führt zu dem Problem, welche nicht?]

Speisende Philosophen (5) — Zirkulares Warten



Betriebsmittelgraph zeigt für jedes Betriebsmittel, welcher Prozess es belegt

Wartegraph verbucht für jeden einzelnen Prozess, auf welches Betriebsmittel er wartet

Ein geschlossener Kreis erfasst all die Prozesse, die sich zusammen im *Deadlock* befinden.

Speisende Philosophen (6) — Kritischer Abschnitt

```
void eat (Semaphore *rod1, Semaphore *rod2) {
    static Semaphore mutex = 1;

    P(&mutex);
    P(rod1);
    P(rod2);          ?
    V(&mutex);
    ⋮
    V(rod2);
    V(rod1);
}
```

Ein Prozess (Philosoph) muss die von ihm gleichzeitig benötigten Betriebsmittel (Stäbchen) atomar anfordern, d. h., die beiden `P()`-Aufrufe bilden einen kritischen Abschnitt:

☞ *binärer Semaphor*

[Sind die `V()`-Befehle nicht auch zu klammern?]

Speisende Philosophen in Realität

Überall dort, wo eine von mehreren Prozessen benutzte Routine (mindestens) zwei *wiederverwendbare Betriebsmittel* zugleich benötigt, diese aber nur nacheinander anfordern kann:

- ☞ 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
- ☞ Blockinhalte von dem einen *Backup Medium* auf ein anderes transferieren

⋮

[Welche Betriebsmittel sind hier die kritischen?]

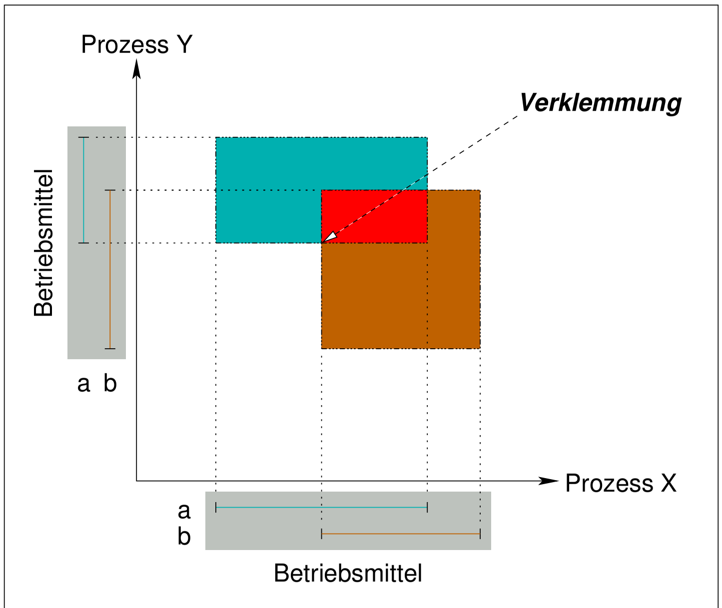
„Nicht-blockierende Verklemmung“

life·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 beteiligten Prozesse auf die Bereitstellung von Betriebsmitteln ***wechselseitig aktiv warten***
- das „größere Übel“ einer Prozessverklemmung, da dieser Zustand nicht eindeutig erkennbar ist und damit die Basis zur „Erholung“ fehlt

[Warum ist ein *Lifelock* nicht eindeutig erkennbar?]

Enstehung einer Verklemmung



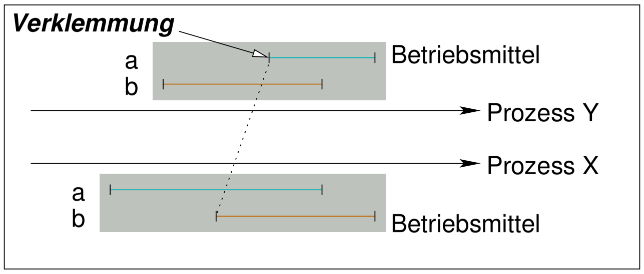
Alles hängt davon ab, (1) ob sich die Prozesse überhaupt einander überlappen und (2) wie sich die Überlappung dann in Bezug auf die gemeinsamen Betriebsmittel zeigt. Die Verklemmung tritt nicht auf, wenn

- P_X B_a erst nach P_Y

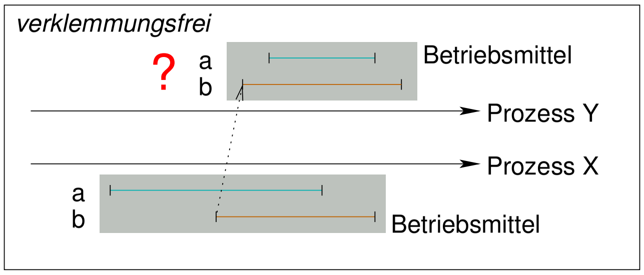
  bzw.
- P_Y B_b erst nach P_X

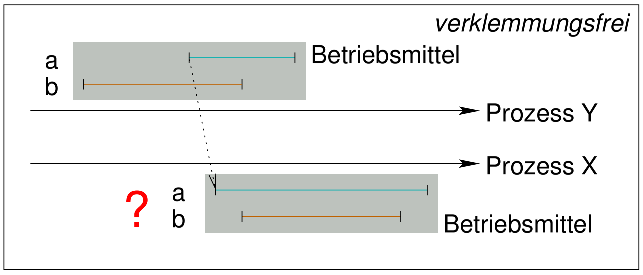
anfordert.

Nicht-deterministische Prozessverläufe



Dass P_X und P_Y ihre Betriebsmittel zeitlich wohlgeordnet anfordern heisst, beide kennen einander und können auch miteinander kooperieren. Für beliebige Prozesse ist dies jedoch nie der Fall.





Voraussetzungen für eine Verklemmung

notwendige Bedingungen

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

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

Verfahren zur „Verklemmungsbekämpfung“

- zirkulares Warten ist mögliche Konsequenz der ersten drei Bedingungen
  - nicht-deterministische Ereignisfolgen lassen Prozessabläufe variieren
  - trotz Gültigkeit der Bedingungen 1–3 ist ein Zyklus nicht zwingend
- zur *Verklemmungsfreiheit* ist „lediglich“ eine der Bedingungen zu entkräften
  - Maßnahmen zur $\left\{ \begin{array}{c} \text{Vorbeugung} \\ \text{Vermeidung} \end{array} \right\}$ von Verklemmungen sind anzuwenden
- ansonsten bleibt nur übrig, Verklemmungen zu erkennen und aufzulösen

Verklemmungsvorbeugung — *Deadlock Prevention*

- Anwendung von *Regeln*, die das Eintreten von Verklemmungen verhindern:

  indirekte Methoden entkräften eine der Bedingungen 1–3
  1. nicht-blockierende Verfahren verwenden
  2. Betriebsmittelanforderungen unteilbar (atomar) auslegen (✗ 9-7)
  3. Betriebsmittelentzug durch *Virtualisierung* praktizieren

  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. $i < j$).
- der *Softwareentwurf* stellt die Durchgängigkeit der Verklemmungsfreiheit sicher

Speisende Philospohen (7) — Nicht-blockierendes Verfahren

```
void eat (int *rod1, int *rod2) {
    while (!cas2(rod1, 1, 0, rod2, 1, 0));
    :
    *rod1 = *rod2 = 1;
}
```

Jedes Stäbchen (Betriebsmittel) wird durch ein `int` kontrolliert: 1 = frei, 0 = belegt. Die Funktion `philosopher()` ist entsprechend anzupassen (✗ 9-4).

- mit `cas2()` werden unteilbar folgende Aktionen durchgeführt:
  1. `rod1` und `rod2` werden auf 1 (frei) überprüft
  2. nur wenn beide 1 sind, werden sie auf 0 (belegt) gesetzt
- `rod1` und `rod2` auf 1 setzen entspricht der Freigabe beider Betriebsmittel

[Wie sehen passiv wartende Lösungen in Kombination mit Bedingungssynchronisation aus?]
[Wie sähe eine Lösung mit cas() aus?]

Virtueller Speicher — Betriebsmittelentzug

```
void hanoi (int n, char from, char to, char via) {
    if (n > 0) {
        hanoi(n - 1, from, via, to);
        printf("schleppe Scheibe %d von %c nach %c\n", n, from, to);
        hanoi(n - 1, via, from, to);
    }
}

int main (int argc, char *argv[]) {
    hanoi(argc == 2 ? strtol(argv[1], 0, 0) : 0, 'A', 'B', 'C');
}
```

Dass das Programm auch vielfach instanziiert bei sehr großen Eingabewerten zwar ausnahmebedingt aber überhaupt terminiert ist der Enträftung der 3. Bedingung (hier: MacOS X, Linux, Solaris) zu verdanken.

[Warum terminiert es ggf. ausnahmebedingt?]

Verklemmungsvermeidung — *Deadlock Avoidance*

- *strategische Maßnahmen* verhindern das mögliche Entstehen des Zyklus
  1.–3. werden nicht zu entkräften versucht, sie bleiben unberücksichtigt
  4. wird auf Basis einer anhaltenden **Bedarfsanalyse** vermieden
- die Verfahren arbeiten dynamisch, mit Wissen über zukünftige Anforderungen
  - das System wird (laufend) auf „unsichere Zustände" hin überprüft
  - Zuteilungsablehnung in Situationen nicht abgedeckten Betriebsmittelbedarfs
  - anfordernde Prozesse werden nicht bedient und frühzeitig suspendiert
  - Einschränkung der Betriebsmittelnutzung verhindert unsichere Zustände
- der maximale Betriebsmittelbedarf der Prozesse muss im Voraus bekannt sein

Speisende Philosophen (8) Sicherer/Unsicherer Zustand

Ausgangspunkt: fünf Stäbchen (Betriebsmittel) sind insgesamt vorhanden

- jeder der fünf Philosophen (Prozesse) 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 zurückgestellt

Leitungsvermittlung — Sicherer/Unsicherer Zustand

Ausgangspunkt: ein Vermittlungsrechner mit 12 Kommunikationskanälen

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

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

- P_3 fordert einen weiteren 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 weitere 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: $P_2 \rightarrow P_1 \rightarrow P_3$ [Warum?]

Vermeidung 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 durch:

- **Betriebsmittelbelegungsgraph** (*resource allocation graph*, [40])
  - eine Vorhersage über das Eintreten von Zyklen wird getroffen $O(n^2)$
- **Bankiersalgorithmus** (*banker's algorithm*, [31])
  1. Prozesse beenden ihre Operationen in endlicher Zeit
  2. der Betriebsmittelbedarf aller Prozesse übersteigt nicht den Gesamtvorrat
  3. Prozesse definieren einen verbindlichen *Kreditrahmen*
  4. Betriebsmittelzuteilung erfolgt variabel innerhalb dieses Rahmens
- beide Verfahrensweisen führen Prozesse dem *long-term scheduling* zu

Verklemmungserkennung — *Deadlock Detection*

- nichts ist im System vorgesehen, um das Auftreten von Zyklen zu verhindern

  1.-4. werden nicht entkräftet, jedoch wird auf 4. nachträglich geprüft

- ein **Wartegraph** wird erstellt und auf Zyklen hin abgesucht $O(n^2)$
  - zu häufige Überprüfung verschwendet Betriebsmittel und Rechenleistung
  - zu seltene Überprüfung lässt Betriebsmittel unausgenutzt bzw. brach liegen

- Kriterien zum Starten der Zyklensuche (in zumeist großen Zeitabständen):
  - Betriebsmittelanforderungen dauern zu lange an
  - die Auslastung der CPU sinkt trotz Prozesszunahme
  - die CPU ist bereits über einen sehr langen Zeitraum untätig

Verklemmungsauflösung

- zwei grundsätzliche Herangehensweisen (auch in Kombination) werden verfolgt:

  Prozesse abbrechen und dadurch Betriebsmittel frei bekommen
  - alle verklemmten Prozesse terminieren (gr. Schaden)
  - verklemmte Prozesse schrittweise abbrechen (gr. Aufwand)

  Betriebsmittel entziehen und mit dem „effektivsten Opfer“ beginnen
  - der betreffende Prozess ist zurückzufahren bzw. wieder aufzusetzen
    ☞ Transaktionen, *checkpointing*/*recovery* (gr. Aufwand)
  - ein Aushungern der zurückgefahrenen Prozesse ist zu vermeiden

- die Kunst besteht in der Gratwanderung zwischen Schaden und Aufwand
  - Schäden sind unvermeidbar und die Frage ist, wie sie sich auswirken

Nachlese . . .

- Verfahren zum Vermeiden/Erkennen von Verklemmungen sind praxisirrelevant
  - sie sind kaum umzusetzen, zu aufwendig und damit nicht einsetzbar
  - die Vorherrschaft sequentieller Programmierung macht sie wenig notwendig
- die Verklemmungsgefahr ist lösbar durch **Virtualisierung** von Betriebsmitteln
  - Prozessen werden in krit. Momenten *physische Betriebsmittel* entzogen[49]
  - dadurch wird eine der Bedingungen (genauer: die 3.) außer Kraft gesetzt
  - Prozesse beanspruchen/belegen ausschließlich *logische Betriebsmittel*
  - der Entzug physischer Betriebsmittel erfolgt somit ohne Wissen der Prozesse
- Verfahren zum Vorbeugen von Verklemmungen sind praxisrelevant/verbreitet

[49] Etwa der Entzug von Hauptspeicher d. h. die Auslagerung von Prozessen (☞ *medium-term scheduling*).

Zusammenfassung

- Verklemmung bedeutet „*deadlock*“ oder „*lifelock*“:

  „[. . .] 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.“ [2]
- für eine Verklemmung müssen vier Bedingungen gleichzeitig gelten
  - exklusive Belegung, Nachforderung und kein Entzug von Betriebsmitteln
  - zirkulares Warten der die Betriebsmittel beanspruchenden Prozesse
- Verfahren zur Verklemmungsbekämpfung: Vorbeugen, Vermeiden, Erkennen