— BS //—

Verklemmungen

Betriebssysteme, ©Wolfgang Schröder-Preikschat

Überblick

Verklemmung — *Deadlock*

deadlock **1** das Entriegelschloss **2** (*fig.*) der Stillstand, vollige Stockung; ***break the –,*** den toten Punkt überwinden; ***come to a –,*** steckenbleiben, sich festfahren, auf ein totes Gleis kommen, auf einen toten Punkt gelangen [1]

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 [5]

{A,}synchrone Kommunikation

```
void foo (...) {
    unsigned int size;
    char data[...];
    ⋮
    while ((size = receive(foobox, data))) {
        ⋮
        send(barbox, data, size);
    }
}
```

```
void bar (...) {
    unsigned int size;
    char data[...];
    ⋮
    while ((size = receive(barbox, data))) {
        ⋮
        send(foobox, data, size);
    }
}
```

- Prozesse „`foo`“ und „`bar`“ laufen Gefahr, sich explizit gegenseitig zu blockieren
  - beide Prozesse { warten auf eine Nachrichten vom jeweils anderen Prozess / können sich nicht von selbst aus dem *Deadlock* befreien
- ein dritter Prozess könnte mit einem „`send`“ die Blockade auflösen

Gepufferte (asynchrone) Kommunikation

```
void foo (...) {
    unsigned int size;
    char data[...];
    ⋮
    do {
        send(barbox, data, size);
        ⋮
    } while ((size = receive(foobox, data)));
}
```

```
void bar (...) {
    unsigned int size;
    char data[...];
    ⋮
    do {
        send(foobox, data, size);
        ⋮
    } while ((size = receive(barbox, data)));
}
```

- Prozesse „`foo`“ und „`bar`“ laufen Gefahr, sich implizit gegenseitig zu blockieren
  - beide Prozesse { versenden Daten in einen ggf. bereits gefüllten Puffer; können sich nicht von selbst aus dem *Deadlock* befreien
- wenn konsumierbare Betriebsmittel durch wiederverwendbare verwaltet werden

Asynchron $\neq$ Nichtblockierend

- zuverlässige nichtblockierende asynchrone Kommunikation kann es nicht geben
  - denn dazu müsste unbegrenzter Pufferspeicher zur Verfügung stehen
  - nicht nur, dass beliebig Nachrichten(inhalte) gepuffert werden müssten
    - im Betriebssystem, im Laufzeitsystem oder in der Anwendung
  - es müssten ebenso viele Nachrichten(empfangs)deskriptoren existieren
  - Nachrichten sind konsumierbare Betriebsmittel, von „unendlicher Anzahl“ **!**
- zuverlässige Kommunikation wirkt sich (logisch) blockierend auf Prozesse aus
  - Puffer sind eben wiederverwendbare Betriebsmittel begrenzter Anzahl **!**
  - viele Puffer senken das Blockierungsrisiko, sie heben es aber nie auf
- nichtblockierende asynchrone Kommunikation impliziert Nachrichtenverlust

Verklemmung — „*Lifelock*“

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.

Die beteiligten Prozesse warten aktiv auf die Freigabe von Betriebsmitteln, die von anderen Prozessen belegt werden, die ihrerseits aktiv auf die Freigabe von Betriebsmitteln warten, die von ersteren Prozessen bereits belegt werden.

Interrupt polling, z.B., kann ein *lifelock* auf ein konsumierbares Betriebsmittel sein, wenn sich das betreffende Gerät nicht in dem Zustand befindet, Interrupts liefern zu können.

Asynchrone Kommunikation

```
void foo (...) {
    unsigned int size;
    char data[...];
    ⋮
    do {
        while (!send(barbox, data, size));
        ⋮
    } while ((size = receive(foobox, data)));
}
```

```
void bar (...) {
    unsigned int size;
    char data[...];
    ⋮
    do {
        while (!send(foobox, data, size));
        ⋮
    } while ((size = receive(barbox, data)));
}
```

- Prozesse „`foo`“ und „`bar`“ laufen Gefahr, anhaltend auf sich aktiv zu warten
  - beide Prozesse { scheitern, Nachrichten im (gefüllten) Puffer abzulegen / können sich nicht von selbst aus dem *Lifelock* befreien
- im Gegensatz zum *Deadlock* ist der *Lifelock* vom System nicht/kaum erkennbar

Adressraumausdehnung

```
char* stretch (int size, int more, char* page[], int slot) {
    if (!more) return 0;

    char* core;

    if ((core = sbrk(size)) != (char*)-1) {
        if (stretch(size, more - 1, page, slot + 1) == (char*)-1) {
            sbrk(-size);
            core = (char*)-1;
        }
    }
    return page[slot] = core;
}
```

Deadlock ist, wenn die ihre Adressräume ausdehnenden Prozesse in `sbrk(2)` auf die Zuteilung von Seiten(rahmen) blockierend warten müssten und kein Prozess terminiert bzw. die von ihm bereits belegten Seiten(rahmen) freiwillig abgibt. Daher blockiert `sbrk(2)` auch nicht.

Lifelock ist, wenn die ihre Adressräume ausdehnenden Prozesse im Fehlerfall `stretch()` wiederholt aufrufen müssen, um ihrer eigentlichen Aufgabe nachkommen zu können. Liefert `errno(3C)` `EAGAIN`, konnte mit `sbrk(2)` der Adressraum mangels Seiten(rahmen) nicht mehr ausgedehnt werden. Es liegt an den Anwendungen, darauf reagieren zu wollen bzw. zu können.

Deadlock vs. *Lifelock*

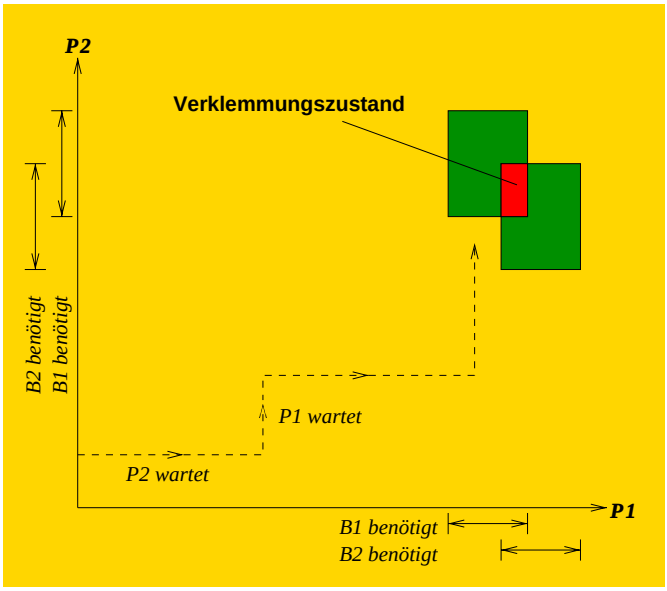
- in beiden Fällen können Prozesse keine (sinnvolle) Arbeit mehr verrichten

  Deadlock die Prozesse laufen nicht, sie sind gegenseitig blockiert
  - der Zustand kann vom Betriebssystem erkannt und aufgelöst werden
  - z.B. durch Zyklensuche in einem Betriebsmittelbelegungsgraphen (→ p. 11)

  Lifelock die Prozesse laufen zwar, sie sind jedoch logisch gegenseitig blockiert
  - der Zustand kann vom Betriebssystem höchstens vermutet/erraten werden
    - welche Kriterien sollen dazu jedoch zu Grunde gelegt werden **?**
    - ist minutenlanges Verharren an derselben Stelle Kriterium genug **?**
    - was ist, wenn die Prozesse ein schlechtes Lokalitätsverhalten zeigen **?**
    - wie sind vollständige und eindeutige (PC) Referenzfolgen herleitbar **?**
  - im Regelfall scheidet eine Erkennung/Auflösung im Betriebssystem aus

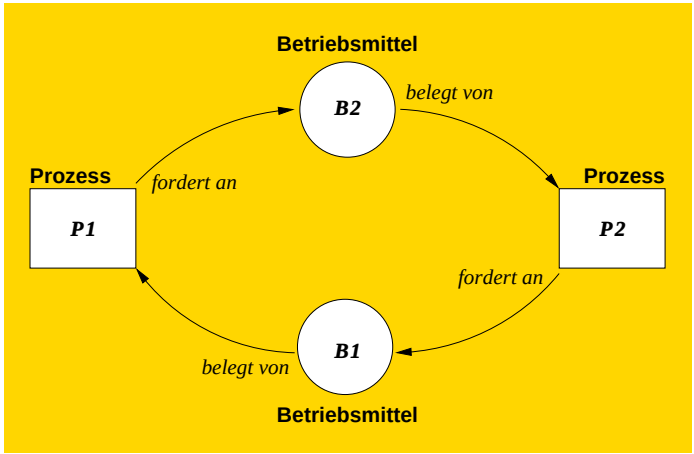
- der *Deadlock* ist das „geringere Übel“ einer Verklemmung unter Prozessen

Konfliktfall



Dilemma (nach [2]). Dargestellt sind zwei Prozesse, P1 und P2, die jeweils zwei Betriebsmittel, B1 und B2, zugleich benötigen. Dabei fordert P1 zuerst B1 und dann B2 an, wohingegen P2 zuerst B2 und dann B1 anfordert. Der Verklemmungszustand tritt jedoch nur dann ein, wenn eine zeitliche Überlappung der Anforderungen beider Prozesse zustande kommt. Ob und, wenn ja, wann dies wirklich eintritt, ist ungewiss.

Betriebsmittelbelegungsgraph



Zyklus. Prozess P1 belegt Betriebsmittel B1 und fordert Betriebsmittel B2 an, das von Prozess P2 belegt wird und der seinerseits Betriebsmittel B1 anfordert. Beide Prozesse brauchen zugleich beide Betriebsmittel, um ihre Aufgabe erfüllen zu können. Derartige Zyklen sind vorzubeugen, zu vermeiden oder sie sind zu erkennen und dann geeignet aufzulösen. Auflösung bedeutet, schrittweises terminieren von Prozessen und/oder entziehen von Betriebsmitteln.

Die Auflösung des Zyklus ist nicht immer „sauber“ durchführbar, z.B. wenn belegte Betriebsmittel überhaupt nicht (mehr) entziehbar sind. Andere Maßnahmen sind daher vorzuziehen.

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 gegenseitig blockierter Prozesse (→ p. 11)

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 (→ p. 10)
- zur Verklemmungsfreiheit ist „lediglich“ eine der Bedingungen zu entkräften
  - Maßnahmen zur $\left\{ \begin{array}{l} \text{Vorbeugung} \\ \text{Vermeidung} \end{array} \right\}$ 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. nichtblockierende anstatt blockierende Synchronisation verwenden ☺
  2. Betriebsmittelanforderung unteilbar, als kritischen Bereich gestalten ☺
  3. Betriebsmittelentzug (wenn möglich) 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

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

Ansätze zur Verklemmungsvermeidung

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*, [6])
  - eine Vorhersage über das Eintreten von Zyklen wird getroffen $O(n^2)$
- ***Bankiersalgorithmus*** (*banker's algorithm*, [3])
  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

- der „goldene Mittelweg" ist schwer zu finden; die Zyklensuche startet . . .
  - falls
    - eine Betriebsmittelanforderung bereits sehr lange andauert
    - die CPU bereits über einen langen Zeitraum untätig ist
    - die CPU-Auslastung (trotz Prozesszunahme) sinkt
  - in mehr oder weniger regelmäßigen (zumeist großen) Zeitabständen

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
    - das bedeutet: 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[1]
  - 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

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

Zusammenfassung

- der Begriff der Verklemmung steht für *Deadlock* bzw. *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.“ [4]

- 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

Referenzen

[1] H. T. Betteridge, editor. *Cassell's Wörterbuch*. Macmillan Publishing Company, New York, 1978. ISBN 0-02-522920-6.

[2] E. G. Coffman, M. J. Elphick, and A. Shoshani. System Deadlocks. *ACM Computing Surveys*, 3(2):66–78, 1971.

[3] E. W. 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).

[4] J. Nehmer and P. Sturm. *Systemsoftware: Grundlagen moderner Betriebssysteme*. dpunkt.Verlag GmbH, zweite edition, 2001. ISBN 3-89864-115-5.

[5] V. E. Neufeld, editor. *Webster's New World Dictionary*. Simon & Schuster, Inc., third college edition, 1988. ISBN 0-13-947169-3.

[6] J. L. Peterson and A. Silberschatz. *Operating System Concepts*. Addison-Wesley, second edition, 1985.