Betriebssysteme

Kap. 3: Interprozesskommunikation (IPC)
Kap. 3.4: Deadlocks

Stand: WS 08/09 (7.1.09)

Prof. Dr. Wolfgang Küchlin

Dipl.-Inform., Dr. sc. techn. (ETH)

Arbeitsbereich Symbolisches Rechnen
Wilhelm-Schickard-Institut für Informatik
Fakultät für Informations- und Kognitionswissenschaften

Universität Tübingen

Steinbeis Transferzentrum
Objekt- und Internet-Technologien (OIT)

Wolfgang.Kuechlin@uni-tuebingen.de
http://www-sr.informatik.uni-tuebingen.de

SR 

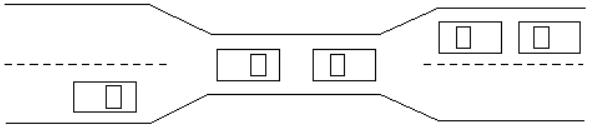
Teil IV: Deadlocks

- Das Deadlock-Problem
  - Modellierung von Verklemmungen
- Deadlock-Vermeidung
  - Der Bankiers Algorithmus von Dijkstra
- Dining Philosophers Problem

Deadlock

- Prozesse blockieren sich gegenseitig beim Zugriff auf kritische Ressourcen
  - Beispiel
    - P_1 reserviert Ressource A,
    - P_2 reserviert B.
    - Anschließend will P_1 auch Ressource B (wartet/blockiert) und P_2 will A (wartet/blockiert)
- Ein Deadlock liegt vor, wenn in einer Menge von Prozessen jeder auf ein Ereignis wartet, das nur andere (ein oder mehrere) dieser Menge auslösen können
  - Lösung im Beispiel
    - Einer der Prozesse muss seine erste Ressource freigeben, wenn er die zweite nicht bekommt.

Bridge Crossing Example



- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.

Silberschatz , Abb.7.3

Deadlock – Bedingungen

- Ein Deadlock kann nur auftreten, falls **alle** der folgenden Bedingungen erfüllt sind.
  - **Wechselseitiger Ausschluss / Unteilbarkeit**
    - Auf mind. zwei Ressourcen kann nur beschränkt und blockierend zugegriffen werden (die Ressourcen sind exklusiv und unteilbar)
  - **Halte und Warte**
    - Mind. zwei Prozesse haben krit. Bereiche, in denen sie auf weitere Ressourcen blockierend zugreifen
  - **No Preemption**
    - Das BS kann die beteiligten Ressourcen den Prozessen nicht entziehen, d.h., diese Ressourcen werden von den Prozessen nur freiwillig wieder hergegeben
  - **Zirkuläres Warten**
    - Prozesse P_1, ..., P_n, die zyklisch auf Ressourcen warten: P_1 wartet auf Ressource, die von P_2 gehalten wird, P_2 auf P_3, ... , P_{n-1} auf P_n und P_n auf P_1

Umgang mit Deadlocks

- Zwei prinzipielle Möglichkeiten
  - Sicherstellen, dass Deadlocks ausgeschlossen werden
    - Problem: Sequentialisierung des Systems
  - Eingetretene Deadlocks vom System auflösen lassen
    - Problem:
      - Aufwendige Erkennung
      - Verlust von geleisteter Arbeit

Methoden zur Vermeidung von Deadlocks

- Statische Prävention
  - Bau des Systems schließt Deadlocks aus
- Dynamische Vermeidung
  - Dynamische Kontrolle der Vergabe von Ressourcen so, dass Deadlocks ausgeschlossen sind
- Deadlocks erkennen
  - Automatische Behebung durch Preemption (falls vorgesehen), oder Termination und Restart des Prozesses
- Verringerung der Häufigkeit und Ignorieren
  - Erhöhung der Anzahl der kritischen Ressourcen
  - keine weitere automatische Behandlung

Deadlock Prävention

Prävention durch Verhinderung einer der Bedingungen:

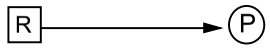
1. Gegenseitiger Ausschluss
   - schwierig / unmöglich
2. Blockierende Zugriffe in kritischen Bereichen: Gemeinsame Locks für mehrere Ressourcen
   - ineffizient
3. No Preemption
   - Freigeben von Ressourcen, wenn Prozess im Begriff ist, in der Anfrage auf andere Ressourcen zu blockieren
4. Circular Wait
   - Durch lineare Ordnung auf den Ressourcen Zugriff auf mehrere Ressourcen immer in Reihenfolge

Für statische Vermeidung eignen sich 2. und 4.
Für dynamische Kontrolle eignen sich 1. und 3.

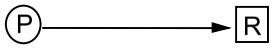
Modellieren von Verklemmungen

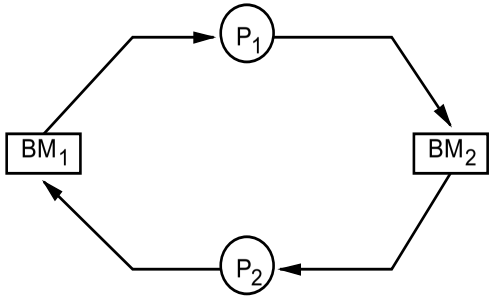
BM: □ Prozesse: ○

BM R wurde Prozess P zugeteilt



Prozess P wartet auf Zuteilung von R



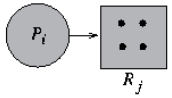


Resource-Allocation Graph

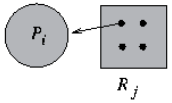
- Process

- Resource type with 4 instances

- P_i requests instance of R_j

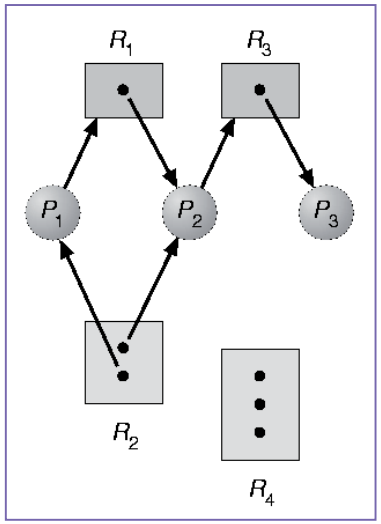


- P_i is holding an instance of R_j



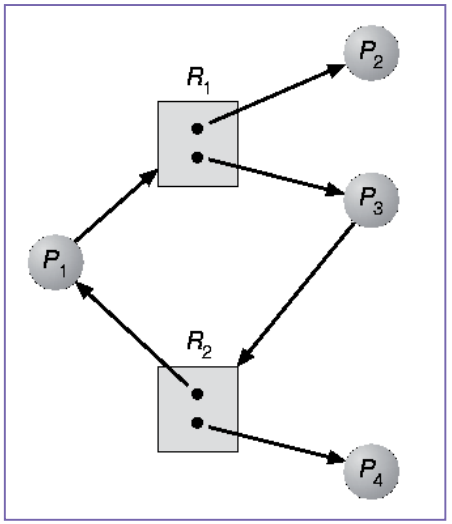
Silberschatz, Abb.7.7

Example of Graph with no cycle



Silberschatz, Abb.7.8

Example of Graph with a cycle



Silberschatz, Abb.7.9

Modellieren von Verklemmungen

- Falls jede Ressource nur einmal vorhanden ist, kann der Ressourcengraph auf Zyklen untersucht werden.
- Keine Zyklen: Kein Deadlock
- Verfahren ist erweiterbar auf mehrere Ressourcen.
- Es ist auch für dynamische Verfahren geeignet

Vermeidung von Verklemmungen

- **Bankiers-Algorithmus**
  - entdeckt Situationen, die zu Verklemmungen führen in Systemen, wo Betriebsmittel mehrfach vorhanden sind
- Voraussetzung:
  - In jedem *Systemzustand* muss bekannt sein, wie viele BM bis zur Beendigung jedes Prozesses maximal benötigt werden.
- Ein Systemzustand besteht aus
  - $E = (E_1, \ldots, E_n)$ Vektor der existierenden BM
  - $A = (A_1, \ldots, A_n)$ Vektor der vorhandenen (noch freien) BM
- Für jeden Prozess *i*:
  - $C_i = (C_{i1}, \ldots, C_{in})$ Vektor der bereits zugeteilten BM
  - $R_i = (R_{i1}, \ldots, R_{in})$ Vektor der noch benötigten BM

Banker's Algorithm

Resources inexistence
$(E_1, E_2, E_3, \ldots, E_m)$

Resources available
$(A_1, A_2, A_3, \ldots, A_m)$

Current allocation matrix

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm} \end{bmatrix}$$

Row n is current allocation to processn

Request matrix

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{bmatrix}$$

Row 2 is what process 2 needs

Tanenbaum, Abb.3-6

Banker's Algorithm

Tape drives, Plotters, Scanners, CDRoms

E = (4 2 3 1)

Tape drives, Plotters, Scanners, CDRoms

A = (2 1 0 0)

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Tanenbaum, Abb.3-7

Banker‘s Algorithm

- *unsicherer Zustand*:
  - Ein Zustand, bei dem sich eine Verklemmung nicht vermeiden lässt, falls alle Prozesse sofort das Maximum ihrer Ressourcen anfordern.
- *sicherer Zustand*:
  - Zustand, der bei geeignetem scheduling nicht zu Verklemmungen führen wird, auch wenn alle Prozesse sofort das Maximum ihrer Ressourcen anfordern.
- Bankiers-Algorithmus gewährt BM Anforderungen dann, wenn sie in einen sicheren Zustand münden.
  - Man nimmt an, die Anforderung werde gewährt und setzt die zugehörigen Vektoren C_i, R_i auf.
- Dann prüft man gemäß dem folgenden Verfahren, ob der Zustand sicher ist.

Banker‘s Algorithm

Bestimmung von **sicheren Zuständen**

- Für alle lauffähigen Prozesse simuliert man die Auswirkungen ihres Ablaufs auf den BM Vektor A. Bleiben zum Schluss nicht lauffähige Prozesse übrig, so sind diese verklemmt.
  1. Suche einen Prozess P_i mit $R_i \le A$ (komponentenweise $\le$).
  2. Falls es keinen solchen Prozess gibt, terminiere.
     Die übrig bleibenden Prozesse sind verklemmt;
     sind keine übrig, ist der Zustand sicher.
  3. Lasse den gefundenen Prozess logisch ablaufen:
     $A := A+C_i$; entferne C_i und R_i, entferne den Prozess.
  4. Go to 1 .
- Die Reihenfolge in 1. ist egal, da nach jedem Schritt A anwächst.

Banker‘s Algorithm, einfaches Beispiel

- Der Bankier hat 100$
- Kreditlinien:
  - Friseur 60$
  - Krämer 80$
- Anforderung 1 (→ sicherer Zustand)
  - Friseur 30$, Krämer 40$
  - Bankier behält 30$. Diese reserviert er für den Friseur, der damit sein Vorhaben abschließen kann. Danach (scheduling) kann auch der Krämer sein Vorhaben durchführen.
- Anforderung 2 (→ unsicherer Zustand)
  - Friseur 40$, Krämer 60$
  - Bankier behält 0$. Das System ist verklemmt, keiner kann weiter arbeiten. Der Bankier darf diese Anforderung nicht gewähren.

Banker‘s Algorithm, Kritik

- Der Bankiers-Algorithmus berücksichtigt nicht, dass
  - Ressourcen auch zurückgegeben werden können, bevor ein Prozess terminiert. (Nicht jeder unsichere Zustand muss daher zu einer Verklemmung führen.)
  - Die maximal gebrauchten BM nicht immer von vornherein bekannt sind.
  - Neue Prozesse entstehen können durch fork oder durch Benutzer.

Banker‘s Algorithm, Beispiel (1 Ressource)

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3 (a)

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1 (b)

	Has	Max
A	3	9
B	0	–
C	2	7

Free: 5 (c)

	Has	Max
A	3	9
B	0	–
C	7	7

Free: 0 (d)

	Has	Max
A	3	9
B	0	–
C	0	–

Free: 7 (e)

- **Beispiel für sicheren Zustand** (bei nur einer Ressource R)
  - Insgesamt 10 Exemplare von R vorhanden
  - Zustand (a) sicher, da es eine Folge von Allokationen gibt, so dass alle Prozesse terminieren können.
  - (a)→(b): B bekommt die maximale Anzahl
  - (b)→(c): B terminiert
  - (c)→(d): C bekommt die maximale Anzahl
  - (d)→(e): C terminiert

Tanenbaum, Abb. 3-9

Banker‘s Algorithm, Beispiel

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3 (a)

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2 (b)

	Has	Max
A	4	9
B	4	4
C	2	7

Free: 0 (c)

	Has	Max
A	4	9
B	--	--
C	2	7

Free: 4 (d)

- **Beispiel für unsicheren Zustand**
- Situation wie vorher mit Unterschied:
  - (a)→(b): A bekommt zunächst ein zusätzliches Exemplar von R
- Ablauf
  - (b)→(c): B bekommt max. Anzahl von R
  - (c)→(d): B terminiert
  - Deadlock: Weder A noch C können zu Ende laufen, wenn sie die maximale Anzahl von R brauchen.
  - (b) ist daher ein *unsicherer Zustand*.

Tanenbaum, Abb. 3-10

Banker‘s Algorithm, Beispiel

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10 (a)

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2 (b)

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1 (c)

- Beispiel für Bankiers Algorithmus mit zehn Exemplaren einer Ressource
  - Zustand (b) ist sicher, da C zu Ende laufen kann (und danach B oder D schließlich gefolgt von A)
  - Zustand (c) ist unsicher (B hat eine zusätzliche Ressource bekommen), da kein Prozess zu Ende laufen kann.
- Algorithmus erteilt Ressourcen nur, wenn danach sicherer Zustand erreicht wird.

Tanenbaum, Abb. 3-11

Banker‘s Algorithm, Beispiel (4 Ressourcen)

	Process	Tape drives	Plotters	Scanners	CD ROMs
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Resources assigned

	Process	Tape drives	Plotters	Scanners	CD ROMs
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Resources still needed

E = (6342)
P = (5322)
A = (1020)

Tanenbaum, Abb. 3-12

Banker's Algorithm, Beispiel

- Erklärung zum Beispiel
  - Matrix „Resources assigned" entspricht C
  - Matrix „Resources still needed" entspricht R
  - E: Vektor der existierenden BM
  - P: Vektor der bereits vergebenen BM
  - A: Vektor der noch freien BM
- Ist Zustand sicher?
  - Prozess D ≤ A → Prozess D kann ablaufen und seine BM zurückgeben → A = (2,1,2,1)
  - Prozess A ≤ A → Nachdem Prozess A terminiert: A = (5,1,3,2)
  - ...
  - Zustand ist sicher!

Behandlung von Deadlocks (Zusammenfassung)

Prävention
Statisch, evtl. ineffizient

Vermeidung
Dynamisch, erzeugt Overhead

Erkennung
Verlust von Arbeit durch Restart

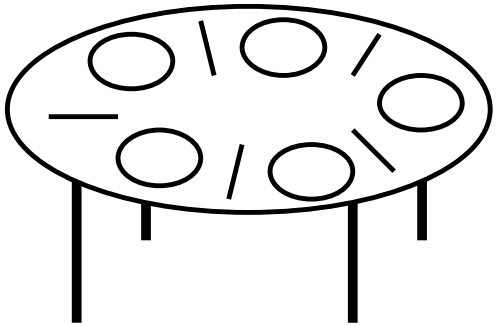
Ignorieren
unbefriedigend

Banker's Algorithm

- Annahme
  - Bedarf an Ressourcen bekannt, Prozesse geben Ressourcen erst am Schluss frei (kein reallocate)
- Ziel
  - Prozessbearbeitung in **sicheren** Zuständen.

System mit Prozessen P_1, ... ,P_n ist in sicherem Zustand, falls es eine Reihenfolge P_{i1}, ... ,P_{in} der Prozesse gibt, so dass sie in dieser Reihenfolge (**sequentiell!**) fertig bearbeitet werden können.

Dining Philosophers



Dining Philosophers

- 5 Philosophen essen, denken oder sind hungrig.
- Kritische Betriebsmittel sind Ess-Utensilien
  - Es gibt 5 Teller mit Reis und
  - zu beiden Seiten jeden Tellers je ein Essstäbchen (*chopstick*)
  - Man benötigt 2 Stäbchen, um essen zu können.
- Probleme:
  1. **Synchronisation**: Die Betriebsmittel Teller und Stäbchen müssen zugeteilt werden.
  2. **Effizienz**: Es soll kein Philosoph unnötig warten müssen (es soll nicht immer nur einer am Tisch sitzen).
  3. **Kein Verhungern**: Es soll nicht vorkommen können, dass ein Philosoph verhungert (*starvation*), weil sich die anderen gegen ihn verbünden und ihm immer die Stäbchen blockieren.
  4. **Keine Verklemmung**: Die Lösung soll nicht zu Verklemmungen führen können.

Dining Philosophers

- Standard-Beispiel für Synchronisation
  - Gegenseitiger Ausschluss
  - Kein Deadlock
  - Fair, kein Verhungern (!-)
  - Effizient
- Lösungen
  - Nur 4 gleichzeitig am Tisch
  - globales Lock zur Besteck-Ergreifung
  - ...

Wechselseitiger Ausschluss: Desiderata

n Prozesse mit kritischen Bereichen

- Gegenseitiger Ausschluss garantiert, dass max. 1 Prozess in einem kritischen Bereich ist.

Gesuchtes Verfahren soll erfüllen

1. [**Korrektheit**]
   Nie dürfen 2 Prozesse gleichzeitig in einem gemeinsamen kritischen Abschnitt sein.
2. [**Allgemeinheit, Portabilität**]
   Es dürfen keine Annahmen über Geschwindigkeit oder Anzahl der CPUs in die Lösung eingehen.
3. [**Effizienz**]
   Kein Prozess, der außerhalb eines kritischen Abschnitts läuft, darf einen anderen Prozess aufhalten (blockieren).
4. [**Fairness**]
   Kein Prozess muss je ewig darauf warten, einen kritischen Abschnitt ausführen zu dürfen. (**Starvation**)

Dining Philosophers

- Wir gehen von genau fünf Philosophen aus. Mehr Philosophen können durch ein Semaphor kanalisiert werden.
- Die folgende Lösung mit Semaphoren erfüllt 1, 2 und 3. Problem 4 (starvation) kann dagegen zufällig auftreten.

Dining Philosophers mit Semaphoren

```
/* Dining Philosophers with Semaphores */

int state[5];
semaphore s[5];
semaphore mutex = 1;
#define left(i) ((i-1)%5)
#define right(i) ((i+1)%5)

void philosopher(i)
int i;
{   while(1)
    {   think();
        take_sticks(i);
        eat();
        put_sticks(i);
    }
}
```

Dining Philosophers mit Semaphoren

```
void take_sticks(i)
int i;
{   down(&mutex);
    state[i] = HUNGRY;
    test(i); up(&mutex);
    down(& s[i]);     /* wait for test */
                      /* to succeed */
}

void test(i)
int i;
{   if (state[i] == HUNGRY
    && state[left(i)] != EATING
    && state[right(i)] != EATING)
    {   state[i] = EATING;
        up(&s[i]); /* signal successful test */
    }
}
```

Dining Philosophers mit Semaphoren

```
void put_sticks(i)
int i;
{   down(&mutex);
    state[i] = THINKING;
    test(left(i));      /* left */
    test(right(i));     /* right */
    up(&mutex);
}
```

Dining Philosophers mit C Threads

```
/* Dining Philosophers in C Threads */

int state[5];
condition_t sticks_available[5];
mutex_t table;

void philosopher(i)
int i;
{   while(1)
    {   think();
        take_sticks(i);
        eat();
        put_sticks(i);
    }
}
```

Dining Philosophers mit C Threads

```
void take_sticks(i)
int i;
{   mutex_lock(table);
    state[i] = HUNGRY;
    while(! ready(i)) condition_wait(sticks_available[i],table);
    state[i] = EATING;
    mutex_unlock(table);
}

void put_sticks(i)
int i;
{   mutex_lock(table);
    state[i] = THINKING;
    condition_signal(sticks_available[(i-1)%5]);
    condition_signal(sticks_available[(i+1)%5]);
    mutex_unlock(table);
}
```

Dining Philosophers mit C Threads

```
int ready(i)
int i;
{   if (state[(i-1)%5] != EATING
        && state[(i+1)%5] != EATING)
      return(1);
    else return(0);
}
```