LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Prinzipien, Modelle und Algorithmen der Nebenläufigen Programmierung
Wintersemester 2019/20

Zugriff auf mehrere Ressourcen

Prof. Dr. David Sabel

LFE Theoretische Informatik

Letzte Änderung der Folien: 10. Dezember 2019

Übersicht

Deadlocks bei mehreren Ressourcen

Deadlock beim Mutual-Exclusion Problem

- Deadlock: Kein Prozess kommt in den kritischen Abschnitt, obwohl mindestens ein Prozess in den kritischen Abschnitt möchte
- Kommt dem Belegen **einer** Ressource gleich
- Wir betrachten nun Prozesse, die mehrere solcher Ressourcen belegen möchten, wobei die Ressourcen durch Sperren geschützt sind.
- Die genaue Implementierung der Sperren lassen wir dabei außer Acht (diese können z.B. durch Semaphore oder Monitore erfolgen).

Deadlocks bei mehreren Ressourcen (2)

Deadlocks allgemeiner (bei mehreren Ressourcen)

Definition

Eine Menge von Prozessen ist deadlocked (verklemmt), wenn jeder (nicht beendete) Prozess aus der Menge auf ein Ereignis wartet, das nur ein anderer Prozess aus der Menge herbeiführen kann.

Das Ereignis entspricht meist dem Freigeben einer Ressource.

Beispiele

- Mutual-Exclusion Problem: Deadlock, wenn es nicht mehr möglich ist, dass irgendein Prozess den kritischen Abschnitt betritt, obwohl alle Prozesse in den kritischen Abschnitt möchten

Beispiele (2)

- Zwei Prozesse machen Umbuchungen zwischen Konto A und Konto B.
- Vorgehensweise:
  - Sperren des ersten Kontos
  - Sperren des zweiten Kontos
  - Überweisung von erstem Konto auf zweites Konto durchführen
  - Entsperren der Konten.
Prozess P	Prozess Q
wait(KontoA);	wait(KontoB);
wait(KontoB);	wait(KontoA);
buche von A nach B	buche von B nach A
signal(KontoA);	signal(KontoB);
signal(KontoB);	signal(KontoA);
- Deadlock!

Beispiele (3)

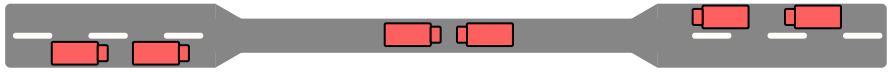
- Funktionierende Lösung (Deadlock-frei)
Prozess P	Prozess Q
wait(KontoA);	wait(KontoA);
wait(KontoB);	wait(KontoB);
buche von A nach B	buche von B nach A
signal(KontoA);	signal(KontoA);
signal(KontoB);	signal(KontoB);

Beispiele (4)

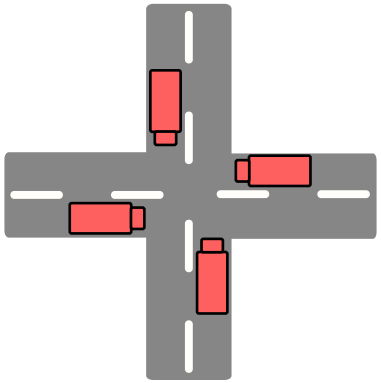
- Ähnliches Problem bei: Speisende Philosophen
- Deadlock möglich, wenn alle Philosophen unsychronisiert
- Alle haben die linke Gabel keiner die rechte.

Beispiele (5)

- Engstelle
- Nur ein Fahrzeug kann passieren

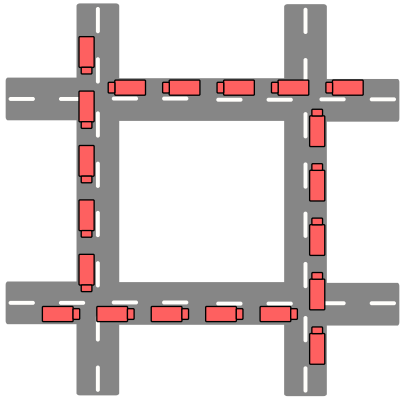


Beispiele (6)



- Alle warten, dass "rechts frei" ist

Beispiele (7)



- Gleiches Problem, aber etwas komplizierter

Deadlock-Behandlung

Vier Ansätze

1. Ignorieren: Keine Vorkehrungen, Hoffnung, dass Deadlocks nur selten auftreten.
2. Deadlock-Erkennung und -Beseitigung: Laufzeitsystem erkennt Deadlocks und beseitigt sie. Problem: Finde Algorithmus der Deadlocks erkennt.
3. Deadlock-Vermeidung: Algorithmus verwaltet Ressourcen und lässt Situation nicht zu, die zu einem Deadlock führen können.
4. Deadlock-Verhinderung: Der Programmierer entwirft die Programme so, dass Deadlocks nicht auftreten können.

Deadlock-Behandlung (2)

- Offensichtlich: Beste Methode: Deadlock-Verhinderung
- Dafür muss man wissen:
  Unter welchen Umständen kann ein Deadlock auftreten?
- Im folgenden: Bedingungen für Deadlock
- und Deadlock-Verhinderung

Wann tritt Deadlock auf?

Vier notwendige Bedingungen (alle gleichzeitig erfüllt):

1. Wechselseitiger Ausschluss (Mutual-Exclusion): Nur ein Prozess kann gleichzeitig auf eine Ressource zugreifen,
2. Halten und Warten (Hold and Wait): Ein Prozess kann eine Ressource anfordern (auf eine Ressource warten), während er eine andere Ressource bereits belegt hat.
3. Keine Bevorzugung (No Preemption): Jede Ressource kann nur durch den Prozess freigegeben (entsperrt) werden, der sie belegt hat.
4. Zirkuläres Warten: Es gibt zyklische Abhängigkeit zwischen wartenden Prozessen: Jeder wartende Prozess möchte Zugriff auf die Ressource, die der nächste Prozesse im Zyklus belegt hat.

Veranschaulichung

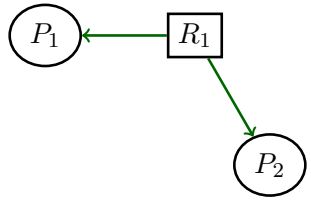
Prozess P_1 will Ressource R_1 belegen (hat sie aber nicht):

Prozess P_1 hat Ressource R_1 belegt:

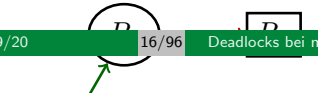
Veranschaulichung der 4 Bedingungen

1. Wechselseitiger Ausschluss (Mutual-Exclusion): Nur ein ausgehender (grüner) Pfeil pro Ressource
2. Halten und Warten (Hold and Wait): Rote und grüne Pfeile für einen Prozess möglich
3. Keine Bevorzugung (No Preemption): Grüne Pfeile können nicht verändert werden (außer vom Prozess, der den Pfeil hat)
4. Zirkuläres Warten: Zyklus im Graphen

Verboten:

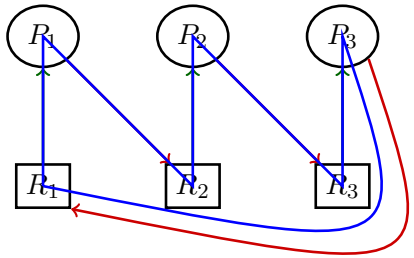


Erlaubt:



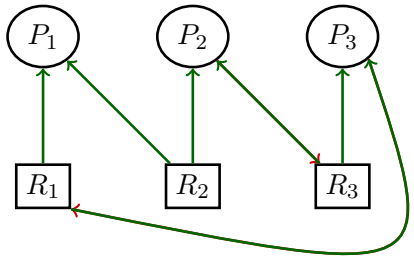
Beispiel

Prozess 1:	Prozess 2:	Prozess 3:
Request R_1	Request R_2	Request R_3
Request R_2	Request R_3	Request R_1
Release R_1	Release R_2	Release R_3
Release R_2	Release R_3	Release R_1



Beispiel: Anderes Scheduling

Prozess 1:	Prozess 2:	Prozess 3:
Request R_1	Request R_2	Request R_3
Request R_2	Request R_3	Request R_1
Release R_1	Release R_2	Release R_3
Release R_2	Release R_3	Release R_1



Deadlock-Verhinderung: Programm so erstellt, dass unabhängig vom Scheduling kein Deadlock auftritt

Deadlock-Vermeidung: Scheduler erkennt Deadlock-Gefahr und schließt diese Schedulings aus

Deadlock-Verhinderung

Ansatz: Greife eine der vier Bedingungen an, so dass sie nie wahr wird.

- Wechselseitiger Ausschluss: Im allgemeinen schwer möglich, manchmal aber schon:
  Z.B. Druckerzugriff wird durch Spooler geregelt.
- No Preemption: Schwierig, man kann z.B. nicht den Zugriff auf den Drucker entziehen, wenn Prozess noch nicht fertig gedruckt hat usw.

Verhindern von Hold and Wait

- Möglichkeit: Prozess fordert zu Beginn alle Ressourcen an, die er benötigt.
- Philosophen: Exklusiver Zugriff auf alle Gabeln
- 1. Problem: Evtl. zu sequentiell
- 2. Problem: Oft nicht klar, welche Ressourcen jeder Prozess braucht
- Variation dieser Lösung: 2-Phasen Sperrprotokoll

2-Phasen Sperrprotokoll

Die Prozesse arbeiten in zwei Phasen

Jeder Prozess führt dabei aus:

- 1. Phase: Der Prozess versucht alle benötigten Ressourcen zu belegen.
  Ist **eine** benötigte Ressource **nicht frei**, so gibt der Prozess **alle** belegten Ressourcen zurück und der Prozess startet von neuem mit Phase 1.
- 2. Phase: Der Prozess hat alle benötigten Ressourcen
  Nachdem er fertig mit seiner Berechnung ist, gibt er alle Ressourcen wieder frei.

Beispiel: Speisende Philosophen

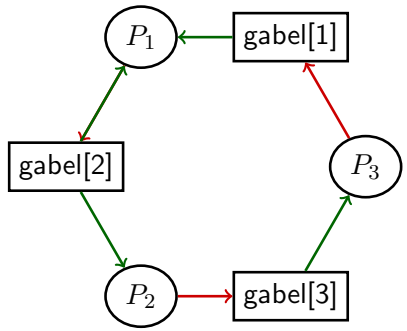
- Initial alle Gabeln (Semaphore) mit 1 initialisiert
- `tryWait`: Nicht-blockierende Implementierung von `wait`:

$$\texttt{tryWait}(S) = \begin{cases} \texttt{True}, & \text{wenn Semaphor belegt werden konnte} \\ \texttt{False}, & \text{sonst} \end{cases}$$

- Phase 1, Phase 2

Philosoph i

```
loop forever
(1)  l := tryWait(gabel[i]);
(2)  if l then
(3)     r:=tryWait(gabel[i+1]);
(4)     if r then
(5)        Philosoph isst
(6)        signal(gabel[i + 1]);
(7)        signal(gabel[i]);
(8)     else signal(gabel[i]);
(9)  Philosoph denkt;
end loop
```



- Deadlock nicht möglich!
- Aber Livelock!

Timestamping-Ordering

- Prozesse erhalten eindeutigen Zeitstempel, wenn sie beginnen.

Zwei-Phasen Sperrprotokoll mit Timestamping

Jeder Prozess geht dabei so vor:

- 1. Phase: Der Prozess versucht alle benötigten Ressourcen auf einmal zu sperren.
  Ist eine benötigte Ressource belegt mit **kleinerem Zeitstempel**, dann gibt Prozess **alle** Ressourcen frei und startet von neuem.
  Ist **eigener** Zeitstempel **kleiner**, dann wartet der Prozess auf die restlichen Ressourcen.
- 2. Phase: Wenn der Prozess erfolgreich in diese Phase gekommen ist, hat er alle benötigten Ressourcen. Er benutzt sie und gibt sie anschließend wieder frei.

Beachte: Neue Zeitstempel werden nur vergeben, nach erfolgreichem Durchlauf durch beide Phasen.

Timestamping-Ordering (2)

- Deadlock-frei
- Livelock nicht möglich
- Starvation-frei.

Definition

Starvation ist eine Situation, in der ein Prozess niemals (nach beliebig vielen Berechnungsschritten) in der Lage ist, alle benötigten Ressourcen zu belegen.

Zirkuläres Warten verhindern

- Versuche das zirkuläre Warten zu verhindern.
- Philosophen-Problem: N. Prozess hebt zuerst rechte Gabel
- Dann kann kein zirkuläres Warten enstehen
- Denn die Gabeln wurden **total geordnet**: $1 < 2 \ldots N$.
- Und die Philosophen haben die Ressourcen entsprechend dieser Ordnung belegt.

Total-Order Theorem

Allgemein gilt:

Total-Order Theorem

Sind alle gemeinsamen Ressourcen durch eine totale Ordnung geordnet und jeder Prozess belegt seine benötigten Ressourcen in aufsteigender Reihenfolge bezüglich der totalen Ordnung, dann ist ein Deadlock unmöglich.

Beweis: durch Widerspruch. Annahme: Es gibt einen Deadlock.
D.h. es gibt Ressourcen $R_1, \ldots, R_n$ und Prozesse $P_1, \ldots, P_n$ mit

- Prozess P_i hat Ressource R_{i-1} belegt
- Prozess P_i wartet auf Ressource R_i

Sei R_j die kleinste Ressource aus $\{R_1, \ldots, R_n\}$ bezüglich der totalen Ordnung. Dann wartet Prozess P_j auf Ressource R_j, wobei er Ressource R_{j-1} schon belegt hat. Widerspruch.

Total Order Theorem (2)

Man kann nachweisen:

Lemma

Ein Deadlock-freies System, indem die Belegung von (allen) *einzelnen* Ressourcen Starvation-frei ist, ist auch insgesamt Starvation-frei

Mit dem Total Order Theorem lässt sich folgern:

Wenn es eine totale Ordnung der Ressourcen gibt, die Ressourcen entsprechend dieser Ordnung belegt werden und die Belegung einzelner Ressourcen Starvation-frei ist, dann ist das Gesamtsystem Starvation-frei.

Deadlock-Vermeidung

Erinnerung:

- Deadlock-Vermeidung: Algorithmus verwaltet Ressourcen und lässt Situation nicht zu, die zu einem Deadlock führen können.

Im folgenden:

- Beispiel von Dijkstra zur Deadlock-Vermeidung:
- Problem des Deadly Embrace
- Lösung: Bankier-Algorithmus

Deadly Embrace

Annahme:

- Es gibt m gleiche Ressourcen
- Jeder Prozesse P_i benötigt eine gewisse Zahl $m_i \leq m$ dieser Ressource
- Prozesse fordern Ressourcen nach und nach an
- Die maximal benötigte Anzahl m_i ist beim Start bekannt
- Hat ein Prozess seine maximale Anzahl, terminiert er und gibt die Ressourcen zurück.
- Problem: Implementiere Ressourcenverwalter

Dijkstra's Veranschaulichung

- Ressource: Geld
- Prozesse sind Bankkunden
- Ressourcenverwalter: Bankier

Deadlock-Vermeidung: Beispiel

- Vorhandene Ressourcen am Anfang: 98 EUR
- 2 Kunden, beide benötigen 50 EUR
- Beide Kunden haben bereits 48 EUR erhalten
- Beide Kunden fordern 1 EUR an.
- Soll der Bankier beide Anforderungen zulassen?
- Nein: Dann haben beide Kunden 49 EUR, die Bank 0 EUR
- Ein Deadlock ist eingetreten.
- Ein Kunde fordert 1 EUR an
- Soll er das Geld bekommen?
- Ja, denn danach ist der Zustand immer noch **sicher**
- **sicher** = Deadlock noch vermeidbar (muss nicht eintreten)

Lösungsversuch

Naive Lösung:

- Kunden erhalten Reihenfolge
- Bankier bedient immer einen Kunden, bis er seinen maximalen Betrag erhalten hat
- Alle andere Kunden müssen warten
- Schlecht, da sequentieller Algorithmus

Deshalb:

- Zusätzliche Anforderung: Erlaube soviel Nebenläufigkeit wie möglich
- Lehne nur dann Anfrage ab, wenn der Zustand dann unsicher würde,
  d.h. ein Deadlock eintreten muss

Bankier-Algorithmus: Datenstrukturen

- Annahme: Verschiedene Ressourcen, z.B. mehrere Währungen
- Vektor $\overrightarrow{A}$: Aktueller Vorrat in der Bank. Jede Komponente von $\overrightarrow{A}$ ist nicht-negative Ganzzahl und stellt Vorrat einer Währung dar

$\mathcal{P}$ Menge der Kunden (Prozesse). Für $P \in \mathcal{P}$ sei:

- $\overrightarrow{M_P}$ der Vektor der maximal durch Prozess P anzufordernden Ressourcen
- $\overrightarrow{C_P}$ der Vektor der bereits an Prozess P vergebenen Ressourcen

Bankier-Algorithmus: Sicherer Zustand

sicher = Deadlock in der Zukunft vermeidbar

Ein Zustand (mit all seinen Vektoren) ist sicher,
wenn es eine Permutation π der Prozesse $P_1, \ldots, P_n \in \mathcal{P}$ gibt,
sodass es für jeden Prozesse P_i entsprechend der Reihenfolge der Permutation genügend Ressourcen gibt, wenn er dran ist.

Genügend Ressourcen bedeutet hierbei:

$$\overrightarrow{A} + \left(\sum_{\pi(j)<\pi(i)} \overrightarrow{C_{P_{\pi(j)}}} \right) - \overrightarrow{M_{P_i}} + \overrightarrow{C_{P_i}} \geq \overrightarrow{0}$$

- Zu den aktuell verfügbaren Ressourcen $\overrightarrow{A}$ dürfen momentan vergebenen Ressourcen hinzuaddiert werden, deren zugehörige Prozesse entsprechend der Permutation vor P_i vollständig bedient werden.

Bankier-Algorithmus: Grundgerüst

- Bankier erhält eine Ressourcenanfrage $\overrightarrow{L_P}$ eines Prozesses $P \in \mathcal{P}$.
- Konsistenzbedingung: $\overrightarrow{L_P} + \overrightarrow{C_P} \leq \overrightarrow{M_P}$
- Bankier berechnet den Folgezustand, alsob P die Ressourcen erhält, d.h.

$$\begin{array}{rcl} \overrightarrow{A} & := & \overrightarrow{A} - \overrightarrow{L_P} \\ \overrightarrow{C_P} & := & \overrightarrow{C_P} + \overrightarrow{L_P} \end{array}$$

- Anschließend: Teste ob Zustand noch sicher
- Wenn unsicher, dann stelle Ursprungszustand her und lasse P warten

Wenn Kunde maximale Ressourcen erhält wird $\overrightarrow{A}$ automatisch angepasst,

Bankier-Algorithmus: Test auf Sicherheit

```
function testeZustand(P, A):
  if P = ∅ then
      return "sicher"
  else
      if ∃P ∈ P mit M_P − C_P ≤ A then
          A := A + C_P;
          P := P \ {P};
          testeZustand(P, A)
      else
        return "unsicher"
```

Eigenschaften

- Algorithmus berechnet eine der gesuchten Permutationen
- Laufzeit $O(|\mathcal{P}|^2)$!
- Kriterium ist ausschließlich „Kein Deadlock“ sonst keine „Optimierung“
- kleine Verbesserung:
  Wenn $\overrightarrow{A} \geq \overrightarrow{M_P} - \overrightarrow{C_P}$ (genügend Ressourcen vorhanden um P komplett zu bedienen) dann ist nach Anfrage $\overrightarrow{L_P}$ der Zustand immer sicher (Test muss nicht ausgeführt werden)

Beispiel

4 Ressourcen (EUR, USD, JYN, SFR)
4 Prozesse A, B, C, D
Aktueller Zustand

Maximal-Werte	Erhaltene Werte	Verfügbare Ressourcen
$\overrightarrow{M_A} = (4,7,1,1)$	$\overrightarrow{C_A} = (1,1,0,0)$	$\overrightarrow{A} = (2,2,3,3)$
$\overrightarrow{M_B} = (0,8,1,5)$	$\overrightarrow{C_B} = (0,5,0,3)$	
$\overrightarrow{M_C} = (2,2,4,2)$	$\overrightarrow{C_C} = (0,2,1,0)$	
$\overrightarrow{M_D} = (2,0,0,2)$	$\overrightarrow{C_D} = (1,0,0,1)$	

Ist der Zustand sicher?

Beispiel (2)

Wir betrachten nun die Anfrage $L_A = (2,2,0,0)$, d.h. Prozess A möchte zwei weitere EUR und zwei weitere USD belegen.
Nach Aktualisierung ($\overrightarrow{A} := \overrightarrow{A} - \overrightarrow{L_A}$ und $\overrightarrow{C_A} := \overrightarrow{C_A} + \overrightarrow{L_A}$) erhalten wir den Zustand:

Maximal-Werte	Erhaltene Werte	Verfügbare Ressourcen
$\overrightarrow{M_A} = (4,7,1,1)$	$\overrightarrow{C_A} = (3,3,0,0)$	$\overrightarrow{A} = (0,0,3,3)$
$\overrightarrow{M_B} = (0,8,1,5)$	$\overrightarrow{C_B} = (0,5,0,3)$	
$\overrightarrow{M_C} = (2,2,4,2)$	$\overrightarrow{C_C} = (0,2,1,0)$	
$\overrightarrow{M_D} = (2,0,0,2)$	$\overrightarrow{C_D} = (1,0,0,1)$	

Bleibt der Zustand sicher?

Transactional Memory

Transactional Memory

- Neuer Programmieransatz
- Ziel: Programmierer schreibt mehr oder weniger sequentiellen Code
- System garantiert Deadlockfreiheit und evtl. noch mehr

Motivation: Beispiel

- Überweisungen zwischen Konto A und B
- Lösung: Sperre Konten entsprechend einer totalen Ordnung
- Erweiterung: Buche nur dann ab, wenn Konto gedeckt
- Lösung 1: Abort und Restart: Wenn Konto nicht gedeckt, starte von neuem. Birgt die Gefahr, immer wieder neu zu starten.
- Lösung 2: Warte bis Konto gedeckt (Sperren müssen aufgehoben werden!)
- Fazit: Kleine Erweiterungen erfordern große Änderungen

Lock-basierte Programmierung ist schlecht ...

(Argumente von S. L. Peyton Jones)

- Setzen zu weniger Locks: Programmierer vergisst eine Sperre zu setzen, Folge: Race Condition
- Setzen zu vieler Locks: Programmierer übervorsichtig: Folgen:
  - Programm unnötig sequentiell (Best-Case),
  - Deadlock (Wort-Case)
- Setzen der falschen Locks: Beziehung zwischen Sperren und Daten kennt nur der Programmierer. Compiler oder andere Programmierer kennen die Beziehung evtl. nicht.

Lock-basierte Programmierung ist schlecht ... (2)

- Setzen von Locks in der falschen Reihenfolge: Das Total-Order Theorem kann nur eingehalten werden, wenn jeder Programmierer weiß, wie die Ordnung der Locks aussieht.
- Fehlerbehandlung schwierig, da bei Fehlerbehandlung Locks entsperrt werden müssen.
- Vergessene `signal`-Operationen oder vergessenes erneutes Prüfen von Bedingungen führt zu fehlerhaften Systemen.

Ein schlagendes Argument

- Lock-basierte Programmierung ist nicht modular

Beispiel:

- Buche von Konto A_1 oder A_2 auf Konto B, je nachdem welches Konto gedeckt ist.
- kann nicht aus den bestehenden Programmen zusammengesetzt werden
- sondern erfordert neues Programm

Transactional Memory: Idee

- Datenbanksysteme sind nebenläufig
- Aus Anwendersicht jedoch: Kein Sperren o.ä. nötig
- Anwender schreibt Anfragen, die als Transaktionen auf der Datenbank ausgeführt werden
- Idee: Übertrage dieses Modell für die nebenläufige Programmierung
- Transaktionen auf dem gemeinsamen Speicher

Transactional Memory: Stand der Technik

- Es gibt einige (prototypische) Implementierungen
- Es gibt Software Transactional Memory aber auch Hardware Transactional Memory
- Jede Menge Designunterschiede
- Transaktionsverwaltung ist Zeit- und Platzintensiv

ACID-Eigenschaften

Zunächst betrachten wir Datenbanktransaktionen.
Diese müssen die ACID-Eigenschaften erfüllen:

- Atomicity: Alle Operationen einer Transaktion werden durchgeführt, oder keine Operationen wird durchgeführt.
  Verboten: Operation schlägt fehl, aber Transaktion erfolgreich
  Verboten: fehlgeschlagene Transaktion hinterlässt beobachtbare Unterschiede
  Eine erfolgreiche Transaktion commits,
  eine fehlgeschlagene Transaktion aborts

ACID-Eigenschaften (2)

- Consistency: Eine Transaktion verändert den Zustand der Datenbank. Diese Änderung muss konsistent sein. Konsistenz einer commited Transaktion hängt von der jeweiligen Anwendung ab. (z.B. der Kontostand eines Bankkontos darf nicht beliebig groß negativ sein, oder ein neu hinzugefügtes Konto muss eine eindeutige Kontonummer erhalten, usw.).

ACID-Eigenschaften (3)

- Isolation: Eine Transaktion liefert ein korrektes Resultat unabhängig davon, wieviele weitere nebenläufige Transaktionen durchgeführt werden.
- Durability: Das Ergebnis einer commited Transaktion ist permanent. D.h. es wird auf der Festplatte oder ähnliches permant geschrieben, bevor die Transaktion als commited gekennzeichnet werden darf.

Speichertransaktionen

- A,C,I wünschenswert
- Durability nicht möglich, da Hauptspeicher flüchtig.
- Atomarität nur gewährleistet, wenn alle Operationen als Transaktionen durchgeführt werden.

Weitere Anforderungen an TM:

- Datenbanken können Zugriffszeiten auf Festplatten mit Rechenzeit gegenrechnen, im Hauptspeicher geht das nicht, da Zugriffszeiten viel kürzer
- TM muss in bestehende Programmiersprachen integriert werden.

Basisoperationen von TM

`atomic`-Blöcke:

```
atomic {
    Code der Transaktion
}
```

- Code wird als Transaktion durchgeführt
- Vorteil gegenüber Monitoren: Variablen müssen nicht explizit aufgezählt werden.
- Problem: Was passiert wenn gleiche Variable auch von außerhalb geändert wird?

Atomarität ist nicht garantierbar

Transaktionen müssen abbrechen oder comitten, aber:

```
atomic {
    while True do skip;
}
```

Deswegen andere Semantik (anders als bei Datenbanken): Die Ausführung einer Transaktion (atomaren Blocks) hat drei mögliche Ergebnisse:

- commit, wenn die Transaktion erfolgreich war,
- abort, wenn die Transaktion abgebrochen wurde
- undefiniert, wenn die Transaktion nicht terminiert

Abbrechen von Transaktion

Transaktionen brechen ab, wenn

- Ein Konflikt auftritt, und der Transaktionsmanager auf Abbruch (und Roll-Back) entscheidet
  Verhalten normalerweise: Manager versucht die Transaktion erneut durchzuführen
- ein expliziter `abort`-Befehl aufgerufen wird.
  Verhalten normalerweise: Transaktion wird abgebrochen

```
atomic {
    Guthaben := Guthaben + Zins;
    if Guthaben < 0 then abort;
}
```


Semantik von atomic

- Programm verhält sich, alsob für alle `atomic`-Blöcke ein globaler Lock verwendet wird.
- Zugriff auf Transaktionsvariablen außerhalb von Transaktionen kann beliebigen Unsinn anstellen
- Schreibender Zugriff: Variable wird verändert
- Lesender Zugriff: Kann Werte beobachten, die noch nicht comitted sind!

Basisoperationen (2)

Der `retry`-Befehl

- Ermöglicht es Transaktionen zu koordinieren
- `retry`: Transaktion wird abgebrochen (Roll-back) und erneut gestartet

Beispiel: Bounded Buffer:

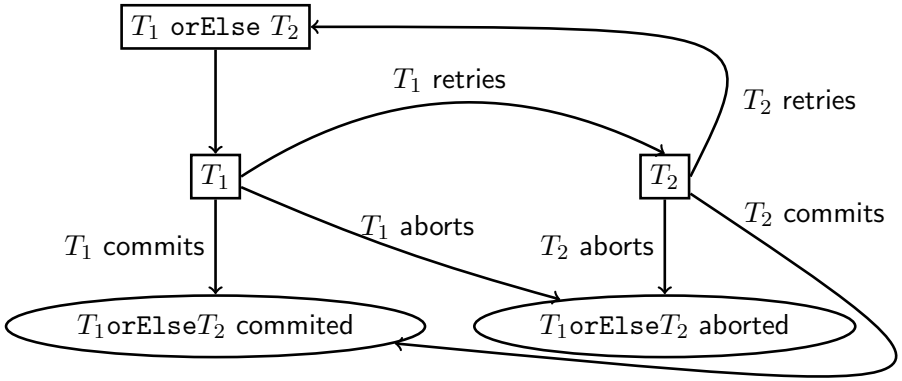
```
atomic{
  if isEmpty(buffer) then retry;
  Lese erstes Element des Puffers usw.
}
```

Basisoperationen (3)

Der `orElse`-Befehl

- Gibt Alternativen vor, wenn Transaktionen abbrechen
- T_1 `orElse` T_2.
  - Zunächst wird T_1 durchführt. Falls T_1 commited oder T_1 explizit via Befehl abgebrochen wird, dann wird T_2 nicht ausgeführt. Die Transaktion T_1`orElse`T_2 commited oder aborts, je nachdem was die Ausführung von T_1 macht.
  - Falls T_1 durch den Transaktionsmanager abgebrochen wird oder `retry` ausgeführt wird, dann startet dieser T_1 nicht neu, sondern versucht T_2 durchzuführen.
  - Falls T_2 explizit aborted oder commited, dann ist die gesamte Transaktion beendet.
  - Falls T_2 ein `retry` durchführt (oder vom System abgebrochen wird), dann wird wieder mit T_1`orElse`T_2 gestartet.

`orElse`-Schaubild



orElse-Beispiel

```
atomic {
  {
    B := B + Betrag;
    A1 := A1 - Betrag;
    if A1 ≤ 0 then retry;
  }
  orElse
  {
    B := B + Betrag;
    A2 := A2 - Betrag;
    if A2 ≤ 0 then retry;
  }
}
```


Merkmale von TM-Systemen

Weak / Strong Isolation

Weak Isolation:

- Speicherplätze können auch außerhalb von Transaktionen manipuliert werden

Strong Isolation:

- Trennung in Transaktionsvariablen und andere Variablen.
- System schützt den Zugriff auf Transaktionsvariablen: Z.B. durch
  - Automatisches Einfügen von `atomic`-Blöcken durch den Compiler
  - Typsystem verbietet Zugriff auf Transaktionsvariablen von außerhalb (z.B. in Haskell)

Merkmale von TM-Systemen (2)

Geschachtelte Transaktionen

```
x := 1;
atomic {
  x := 2;
  atomic {
     x := 3;
     abort;
  }
}
```

Was soll eine solche Schachtelung bedeuten?

Drei Varianten: geglättete, geschlossene, offene Transaktionen

Merkmale von TM-Systemen (3)

Geglättete Transaktionen:

Verhalten, alsob das innere atomic Statement fehlt.

```
x := 1;
atomic {
  x := 2;
  atomic {
     x := 3;
     abort;
  }
}
```

verhält sich wie:

```
x := 1;
atomic {
  x := 2;
  x := 3;
  abort;
}
```

Das Beispiel ergibt abort (x=1).

Merkmale von TM-Systemen (4)

Geschlossene Transaktionen:

- Verhalten wie bei geglätteten Transaktionen, aber:
- Innerer Abbruch führt nicht zum Abbruch der äußeren Transaktion.

```
x := 1;
atomic {
  x := 2;
  atomic {
     x := 3;
     abort;
  }
}
```

verhält sich wie

```
x := 1;
atomic {
  x := 2;
}
```

(da innere Transaktion abbricht)

Beispiel ergibt x=2.

Merkmale von TM-Systemen (5)

Offene Transaktionen:

- Innere Transaktionen sind eigenständig.
- Sobald innere Transaktion committed, ist deren Effekt für alle sichtbar und bleibt sichtbar, selbst dann, wenn die äußere Transaktion abbricht

```
x := 1
atomic {
  x:=2;
  atomic {
     x:= 3;
  }
  abort;
}
```

Ergebnis: x = 3!

Merkmale von TM-Systemen (6)

Granularität

- Welche Einheiten werden auf Konflikte überwacht?
- Wort-/ Blockgranularität: Wenn paralleler Zugriff auf Speicherworte, dann Konflikt
- Objektgranularität: Paralleler Zugriff auf ein Objekt führt zum Konflikt

Beachte bei Objektgranularität: Konflikt auch dann, wenn unterschiedliche Objektattribute geändert werden

Merkmale von TM-Systemen (7)

Direktes / verzögertes Update

Direktes Update

- Ausführende Transaktionen modifizieren Objekte sofort.
- Erfordert Schutz vor gleichzeitigem Zugriff (Concurrency Control)
- Alte Werte werden in einem Log gespeichert
- Abbruch der Transaktion: Roll-Back und Recovery: Wiederherstellen der Daten aus dem Log

Merkmale von TM-Systemen (8)

Verzögertes Update

- Transaktionen erhalten lokale Kopien der Objekte
- Modifikation zunächst an den Kopien
- Beim Commit: Originale werden durch lokale Kopien ersetzt
- Logging: Wer hat welche Kopien, wer darf committen?

Merkmale von TM-Systemen (9)

Direktes / verzögertes Update

- Direktes Update ist **optimistisch**, verzögertes Update **pessimistisch**
- Optimistisch: Annahme, dass Konflikte selten auftreten
- Pessimistisch: Annahme, dass Konflikte häufig auftreten

Merkmale von TM-Systemen (10)

Zeitpunkt der Konflikterkennung

- **Early:** Beim ersten Zugriff, der zum Konflikt führt:
  (Notwendig dafür: Logging, wer wann zugegriffen hat)
- **Late:** Erst beim committen wird geprüft, ob ein Konflikt vorliegt
- **Irgenwann:** Iteratives Prüfen (in Zeitabständen)

Merkmale von TM-Systemen (11)

Konfliktmanagement

- Beim Konflikt: Welche Transaktion wird abgebrochen?
- Viele verschiedene Strategien
- Ziel: Fortschritt des Gesamtsystems
- Dadurch verboten: Breche alle Transaktionen ab

Korrektheitskriterien für STM-Systeme

- Was muss gelten damit ein TM-System als „korrekt“ gilt
- Eher umstritten, viele Forschungsarbeiten
- Z.B. Dziuma,Fatourou,Kanellou 2015: Überblick mit 16 verschiedenen Korrektheitskriterien und Beziehungen zwischen diesen:
  - strict-serializability
  - serializability
  - opacity
  - causal-consistency
  - causal-serializability
  - virtual-world-consistency
  - strong-virtual-world-consistency
  - snapshot-isolation

  jeweils in einer eager und einer deferred-update Variante

Korrektheitskriterien für STM-Systeme (2)

Man kann das ganze sehr formal machen:

- Transaktionsausführung:
  Schrittweise Abarbeitung, wobei 1 Schritt einem Aufruf auf einem nebenläufigen Objekt (z.B. atomares Register, CAS-Objekt, ...) und lokalen Berechnungen entspricht
- Abstrakter: Historie der Transaktionsausführung als Folge von Ereignissen.

  Ereignis:

  - Aufruf/Rückgabe von
    READ(x), WRITE(x,v), COMMIT, ABORT
  - Rückgabe kann immer auch Spezialwert A_T sein:
    Transaktion T ist abgebrochen.

Beispiel

Z.B. $x = y = z = 1$ am Anfang

Transaktion 1: $x := y + 1$

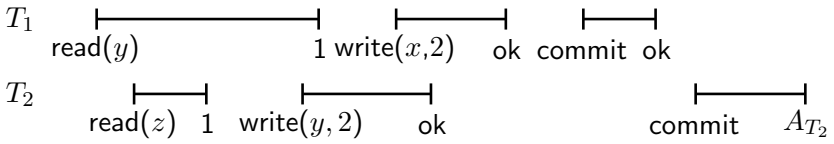
Transaktion 2: $y := z + 1$

Eine mögliche Historie ist:

T_1.read(y)
T_2.read(z)
T_2.1
T_2.write(y,2)
T_1.1
T_1.write(x,2)
T_1.ok
T_1.commit
T_1.ok
T_2.commit
$T_2.A_{T_2}$

Beispiel (2)

Historie anders (genauer?) dargestellt



Unser Vorgehen

- Wir formulieren die Korrektheitskriterien direkt, möglichst ohne das formale Modell der Historien zu verwenden.
- Wir stellen nur einige wenige Kriterien vor (aus Guerraoui & Kapalka, PPoPP 2008)

Wesentliche Anforderungen einer korrekten STM-Implementierung

- Transaktionen, die erfolgreich sind, erscheinen, alsob sie unteilbar in einem Schritt durchgeführt wurden
- abgebrochene Transaktionen erscheinen, alsob sie gar nicht durchgeführt wurden.

Weitere Forderungen: Erhaltung der Realzeit-Ordnung

- Der Zeitpunkt zudem die Effekte einer Transaktion erscheinen, liegt irgendwo in der Laufzeit der Transaktion.
- D.h. die Transaktion selbst sollte keine veralteten Speicherzustände lesen.
- Effekt kann z.B. dadurch auftreten, dass durch die Verwendung von Caches noch alte Werte der transaktionalen Variablen gespeichert sind.
- Es sollte gelten: Wenn eine Transaktion T_1 ein Objekt x modifiziert und committed, und danach eine Transaktion T_2 startet und x liest, dann liest T_2 den von T_1 geschriebenen Wert und nicht einen älteren Wert.

Weitere Forderungen: Keine inkonsistenten Zustände lesen

- Dürfen laufende Transaktionen, die weder committed noch abgebrochen sind, inkonsistente Speicherzustände lesen, wenn sichergestellt ist, dass sie später sowieso abgebrochen werden?
- Für Datenbanktransaktionen ist sowas erlaubt und unproblematisch
- Aber: STM-Transaktionen laufen nicht in einer völlig abgekapselten Umgebung

Weitere Forderungen: Keine inkonsistenten Zustände lesen

Beispiel:

- Seien x und y zwei (transaktionale) Variablen, welche die Invariante $y = x^2$ und $x \geq 2$ stets erfüllen sollen.
- Der Programmierer erfüllt die Invariante, indem darauf achtet, dass alle Transaktionen sie erfüllen
- Sei $x = 4$ und $y = 16$
- Betrachte nun die Transaktion T_1: $x := 2; y := 4;$ commit.
- Wenn eine andere Transaktion T_2 den alten Wert von x (4) und den neuen Wert von y (4) beobachten kann, könnte diese $1/(y - x)$ berechnen und damit einer Laufzeitfehler verursachen.
- Das Abbrechen von T_2 aufgrund des Lesen inkonsistenter Werte verhindert diesen Fehler nicht.

Korrektheitskriterien (1)

Linearisierbarkeit:

- Jede Transaktion T wirkt wie ein atomarer Funktionsaufruf auf den transaktionalen Variablen, der in einem Schritt ausgeführt wird.
- Die erfolgreichen Transaktionen sind dann eine Sequenz solcher Schritte.

Korrektheitskriterien (2)

Sequentialisierbarkeit

- Historie der einzelnen Read/Write-Zugriffe der **erfolgreichen** Transaktionen
- Historie des nebenläufigen Ablaufs ist **äquivalent** zur Historie eines sequentiellen Ablaufs aller erfolgreichen Transaktionen.

Äquivalenz:
Zwei Historien sind äquivalent, wenn die Ereignissefolge pro Transaktion dieselbe ist (d.h. gleiche Reihenfolge innerhalb einer Transaktion und gleiche Rückgaben).

Sequentieller Ablauf:
Die Ereignisse verschiedener Transaktionen treten nicht verzahnt sondern sequentiell nacheinander auf.

Beispiel

Historie H_1:	Historie H_2:	Historie H_3:	Historie H_4:
T_1.read(x)	T_1.read(x)	T_1.read(x)	T_2.read(x)
T_2.read(x)	T_2.read(x)	$T_1.v$	$T_2.v$
$T_1.v$	$T_2.v$	T_1.write(x,v')	T_2.read(y)
$T_2.v$	$T_1.v$	T_1.ok	$T_2.v''$
T_2.read(y)	T_1.write(x,v')	T_1.commit	T_2.commit
T_1.write(x,v')	T_1.ok	T_1.ok	T_2.ok
T_1.ok	T_1.commit	T_2.read(x)	T_1.read(x)
T_1.commit	T_2.read(y)	$T_2.v'$	$T_1.v$
$T_2.v''$	T_1.ok	T_2.read(y)	T_1.write(x,v')
T_1.ok	$T_2.v''$	$T_2.v''$	T_1.ok
T_2.commit	T_2.commit	T_2.commit	T_1.commit
T_2.ok	T_2.ok	T_2.ok	T_1.ok

- Historien H_3, H_4 sind sequentiell
- H_1 und H_2 sind äquivalent
- H_1 und H_3 sind nicht äquivalent, H_1 und H_4 sind äquivalent

Korrektheitskriterien (3)

Strikte Sequentialisierbarkeit

- Zusätzlich zur Sequentialisierbarkeit: Realzeit-Ordnung wird eingehalten, d.h. der gesuchte sequentielle Ablauf muss der Realzeit-Ordnung entsprechen.

Erweiterung: **Globale Atomarität**: Erlaubt auch andere Operationen als Read und Write und erlaubt Kopien nebenläufiger Objekte.

Alle Begriffe bisher betrachten nur erfolgreiche Transaktionen.

Korrektheitskriterien (4)

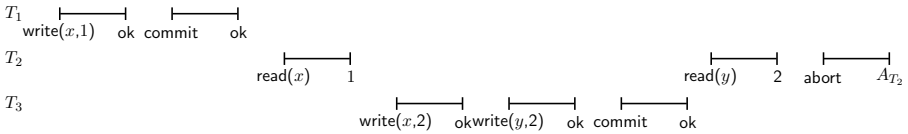
Recoverability

- Wenn eine Transaktion T_i eine transaktionale Variable beschreibt, dann darf keine andere Transaktion T_j die selbe transaktionale Variable lesen, bevor T_i erfolgreich oder abgebrochen ist.

Probleme:

- Erzwingt manchmal zuviel Sequentialisierung der Transaktionsausführung.
- Verbietet nicht, dass abbrechende Transaktionen inkonsistente Zustände sehen.

Beispiel



Obwohl Recoverability gilt, liest T_2 inkonsistente Werte

Korrektheitskriterien (5)

Opacity

- Die Operationen jeder erfolgreichen Transaktion werden wie atomar in einem Schritt durchgeführt. D.h. insbesondere: Entfernt man alle Schritte von abgebrochenen oder laufenden Transaktionen, so ist die (nebenläufige) Ausführung der erfolgreichen Trransaktionen äquivalent zu einer sequentiellen Ausführung dieser Transaktionen.
  Zusätzlich muss gelten, dass die sequentielle Ausführung die Realzeit-Ordnung einhält.
- Effekte von abgebrochenen Transaktionen sind niemals sichtbar für andere Transaktionen.
- Jede Transaktion (egal ob erfolgreich oder abgebrochen) sieht nur konsistente Zustände, d.h. solche die von committeten Transaktionen entstanden sind, und einem Zustand in der sequentiellen Ausführung entsprechen.

Anforderungen für Opacity etwas genauer

Für jede vollständige Historie H muss gelten: Es gibt eine sequentielle Historie S sodass

- S ist äquivalent zu H
- S erhält die Realzeit-Ordnung bezüglich H
- Für jede Transaktion T_k in S, ist die längste Teilfolge bestehend aus allen Transaktionen, die vor T_k committed sind, und T_k selbst eine zulässige Historie. Zulässig meint, dass die Rückgaben der Semantik der Operationen entsprechen.

Vervollständigung von unvollständigen Historien (es gibt noch Transaktionen, die weder abgebrochen noch committed sind)

- Transaktion die noch nicht commit aufgerufen haben, werden abgebrochen
- Transaktionem die commit aufgerufen haben, werden committed oder abgebrochen (beide Möglichkeiten testen)

Der TL2-Algorithmus

- Vorgeschlagen von Dice, Shalev und Shavit, 2006
- TL = Transactional Locking
- Implementiert den Transaktionsmanager
- Kein `retry` und `orElse`
- Keine verschachtelten Transaktionen
- Verwendet einen globalen Zähler gc (fetch-and-increment-Objekt)
- Erfüllt Opacity.

TL2: Transaktionale Speicherplätze

Jedem Speicherplatz hat ein CAS-Objekt `lock` zugeordnet, mit Inhalt (l,stamp):

- $l \in \{\texttt{True}, \texttt{False}\}$: Lockinglabel, zeigt an, ob Speicherplatz gesperrt.
- stamp: Zeitstempel markiert letzten Schreibzugriff, oder falls $l = \texttt{True}$ eine Prozess-Identifikation.

TL2: Datenstrukturen der Transaktionen

Jede Transaktion verwendet:

- Ein **Readset** von Speicherplätzen: Enthält alle Adressen der CAS-Objekte `lock` von Speicherplätzen, die von der Transaktion gelesen wurden
- Ein **Writeset** von Paaren (Speicherplatz,neuer Inhalt): Alle Schreiboperationen werden zunächst auf dem Writeset durchgeführt, nicht auf dem gemeinsamen Speicher
- rv, wv: Lokale Variablen, die Zeitstempel enthalten.

TL2: Ablauf der Transaktion (1)

Start der Transaktion:

- Initialisiere Readset und Writeset mit leeren Mengen
- Lese globalen Zähler und setze: rv := gc;

Lesen von Speicherplatz t:

- Wenn (t, w) bereits im Writeset: Gebe w zurück.
- Sonst:
  (l_1,stamp$_1$) := t.`lock`
  result := t.`content`
  (l_2,stamp$_2$) := t.`lock`
- Breche Transaktion ab (und beginne von vorne), wenn
  - $l_1 = \texttt{True}$ oder $l_2 = \texttt{True}$
  - stamp$_1$ > rv (ein anderer hat geschrieben, nachdem die Transaktion begann.
  - stamp$_1 \neq$ stamp$_2$: Änderung während des Lesens.
- Sonst: Füge Adresse von t.`lock` zum Readset hinzu und gebe result zurück.

TL2: Ablauf der Transaktion (2)

Schreib-Operation auf t mit Wert w

- Füge (t, w) dem Writeset hinzu (oder update das Writeset entsprechend)

Commit-Phase:

- Alle Befehle wurden abgearbeitet
- Sperre alle Speicherplätze aus dem Writeset
  (wenn dies nicht gelingt, gebe alle Sperren zurück und starte die Transaktion neu)
- Inkrementiere gc und setze wv auf den neuen Wert von gc
- Validiere Readset:
  - Prüfe für jede `lock`-Adresse, ob rv $\geq$ stamp gilt.
  - Wenn eine Sperre gesetzt oder rv $<$ stamp, dann nicht valide!

  Wenn Readset nicht valide, dann starte Transaktion neu
- Schreibe Writeset-Einträge in den gemeinsamen Speicher
- Entferne Sperren und setze die Zeitstempel auf wv.

TL2: Bemerkungen

- Prüfen des Readset und Sperren des Writeset stellt sicher, dass der Effekt der Transaktion wie eine Funktion wirkt
- TL2 verwendet wenige Sperren: Lock wird nur beim Committen gesetzt.
- Erzeugen neuer Speicherplätze: Einfach durch ein weiteres Set.

TL2: Bemerkungen (2)

Nur lesende Transaktionen

- Können auch auf das Readset verzichten
- Optimierung: Behandele jede Transaktion zunächst wie eine nur lesende, beim ersten Schreiben: Neustart und lese/schreibe Transaktion

TL2: Semantischer Fehler bei bedingter Nichtterminierung

Sei x = True am Anfang

Transaktion 1	Transaktion 2
`if x then`	`x` := False;
`  loop forever`	
`else return`	

Wenn Transaktion 1 zuerst läuft, wird diese in eine Endlosschleife laufen, obwohl nach Ablauf von Transaktion 2, keine Endlosschleife notwendig.

Mögliche Abhilfe: Iteratives Prüfen des Readset auf Konflikt.

Fazit

- Leichtere Programmierung mit TM
- Problem: Transaktionen können nur Operationen sicher verarbeiten, die umkehrbar sind
- Z.B. nicht umkehrbar: Drucke „Hallo“, „Launch Missiles“ etc.
- Weitere Probleme: Code in Transaktionen zerlegen:
  Kleine Transaktionen sind besser als große (wegen Konfliktpotential)
  Man muss herausfinden: Welcher Codeabschnitt muss atomar durchgeführt werden.
  Datenstrukturen: Z.B. ein Baum in ein transaktionalen Variable vs. ein Knoten in einer transaktionalen Variablen
- Wir sehen noch Programmierbeispiele beim Thema Haskell STM