Kapitel 13: Concurrency Control – Synchronisation von Prozessen und Transaktionen

Wenn mehrere Programme gleichzeitig auf eine Datenbank zugreifen und mindestens ein Programm die Daten ändert, kann es zu Inkonsistenzen kommen, auch wenn jedes einzelne Programm konsistenzbewahrend ist. Die potentiellen Anomalien resultieren aus der parallelen bzw. nebenläufigen Ausführung der Programme. *Nebenläufigkeit (engl.: concurrency)* bedeutet dabei, dass die verschiedenen Schritte mehrerer Programme zeitlich verschränkt ablaufen, also z.B. bei Programm 1 mit den Schritten p11, p12 und p13 und Programm 2 mit den Schritten p21 und p22 in der Reihenfolge p11, p21, p12, p13, p22. Bei einer parallelen Ausführung können sogar Schritte verschiedener Programme echt gleichzeitig ablaufen (z.B. auf einem Multiprozessor-Rechner). Die in diesem Kapitel betrachteten Anomalien treten jedoch bereits bei nebenläufiger Ausführung auf, die wir aus diesem Grund auch als *Quasi-Parallelität* bezeichnen.

Zur Lösung, also garantierten Vermeidung von Anomalien, müssen Programme bzw. deren Instantiierungen zur Laufzeit synchronisiert werden; die Instantiierung bezeichnen wir auch als *Prozesse* oder – im Kontext eines Datenbanksystems – als *Transaktionen*. Die Maßnahmen zur *Synchronisation* bezeichnen wir generell auch als *Concurrency Control* oder Nebenläufigkeitssteuerung. Im folgenden werden zunächst allgemeine Mechanismen zur Synchronisation von Prozessen – auf Betriebssystemebene – und anschließend weitergehende Möglichkeiten zur Synchronisation von Transaktionen – auf Datenbanksystemebene – betrachtet. Die vorgestellten Lösungen für nebenläufige Ausführungen lösen auch die Situationen bei echt paralleler Ausführung und – mit Erweiterungen – auch verteilter Ausführung auf mehreren Rechnern.

13.1 Synchronisation nebenläufiger Prozesse

Unter einem *Prozess* verstehen wir den Ausführungskontext eines beliebigen Programms, also den aktuellen Befehlszähler, die Belegung der Registersätze, die Daten im virtuellen Adressraum usw. Ein Prozess läuft unter der Steuerung des Betriebssystems, das – zusammen mit der Hardware – den Ausführungskontext verwaltet. Synchronisation nebenläufiger Prozesse bedeutet, dass bestimmte Prozesse unter bestimmten Bedingungen auf andere Prozesse bzw. Ereignisse, die von anderen Prozessen ausgelöst werden, warten müssen. Die wichtigste Situation, die Synchronisation erfordert, ist der Zugriff mehrerer Prozesse auf eine gemeinsame Datenstruktur. Um die Datenstruktur konsistent zu halten, darf in vielen dieser Situationen nur maximal ein Prozess zu einer Zeit auf die Datenstruktur zugreifen. Diese Notwendigkeit nennen wir *wechselseitigen Ausschluss (engl.: mutual exclusion)*. Eine andere Situation, die wechselseitigen Ausschluss erfordert, ist die Notwendigkeit, dass von mehreren Prozessen, die dasselbe Programm ausführen, maximal einer in einem bestimmten Abschnitt des Codes sein darf (z.B. der Interrupt-Behandlung des Betriebssystemskerns auf einem Multiprozessor-Rechner). Solche Codeabschnitte nennen wir *kritische Abschnitte (engl.: critical sections)*.

Synchronisation mit Semaphoren

Zur Realisierung von wechselseitigem Ausschluss stellen Betriebssystem Synchronisationsprimitive zur Verfügung, die als *Semaphore* bezeichnet werden (griechisches Wort für Zeichenträger, z.B. für Flaggensignale in der Antike, sinngemäß Ampel). Ein Semaphor ist eine ganzzahlige, nichtnegative Variable mit zwei atomaren, also ununterbrechbaren und unteilbaren, Operationen *P bzw. Wait* und *V bzw. Signal* (von den holländischen Wörtern „passeer = betreten, passieren“ und „verlaat = verlassen“ herrührend – das Semaphorkonzept stammt von Edsger W. Dijkstra). In Unix werden (verallgemeinerte) Semaphore durch die System-Calls semctl, semget und semop unterstützt; Windows hat analoge System-Calls.

Die Semantik der Semaphor-Operationen P und V ist die folgende:

```
Boolean P (Semaphore x) {
        if (x == 1) {x = x-1; return True} else return False;}

void V (Semaphore x) {x = x+1;};
```

Dabei sei x mit dem Wert 1 vorinitialisiert. Der Wert von x kann als die Anzahl verfügbarer Exemplare eines bestimmten Betriebsmittels interpretiert werden. Wenn ein Prozess einen Wert >0 sieht, kann er ein Exemplare dieses Betriebsmittels akquirieren. Durch Initialisierung von x mit einem Wert >1 wird aus dem binären Semaphor ein zählender Semaphor. Essentiell an den Semaphor-Operationen P und V ist, dass beide unteilbar sind. Ein Prozess, der gerade eine P-Operation ausführt, kann also nicht zwischen dem Lesen von x und dem Erniedrigen von x unterbrochen werden. Diese Semantik ist durch das Betriebssystem und die Rechner-Hardware sicherzustellen.

Mit Semaphoren lässt sich wechselseitiger Ausschluss elegant realisieren. Betrachten wir ein Programm p mit Schritten p1, p2, p3 und p4; p2 und p3 sollen zusammen einen kritischen Abschnitt bilden. Wir führen eine Semaphor-Variable mutex ein und erweitern das Programm wie folgt:

```
p1; while (P(mutex)!=False) { }; p2; p3; V(mutex); p4;
```

Bei der nebenläufigen Ausführung zweier Instantiierungen p und p’ dieses erweiterten Programms sind dann beispielsweise die folgenden Reihenfolgen möglich:

p1, p2, p3, p1’, p4, p2’, p3’, p4’
p1, p1’, p2, p3, p2’, p3’, p4, p4’
p1, p1’, p2’, p3’, p2, p3, p4’, p4
usw.,

nicht aber die folgenden verbotenen Reihenfolgen:

p1, p2, p2’, p3, p3’, p4, p4’
p1, p2’, p2, p3’, p3, p4, p4’
p1, p2’, p2, p3, p3’, p4, p4’

Erlaubte und verbotene Reihenfolge der Schritte solcher nebenläufiger Ausführung lassen sich auch durch Ausdrücke sog. *Prozessalgebren* (z.B. CSP - Calculus for Communicating Sequential Processes - von Sir Anthony Hoare) formal charakterisieren.

Implementierung von Semaphoren

Semaphore können auf 3 Arten realisiert werden:

- durch eine *unteilbare Maschineninstruktion* vom Typ *Test-and-Set* (oder einer ähnlichen Instruktion): Test-and-Set <addr> <val> testet den Inhalt der Speicheradresse addr (ein Byte oder ein Wort); falls der Wert gleich 0 ist, wird ein Condition-Code-Register (CC) auf 1 gesetzt und gleichzeitig wird der Inhalt der Speicheradresse auf den Wert val gesetzt; das CC-Register kann von nachfolgenden Sprungbefehlen abgefragt werden
- durch eine *ununterbrechbare Prozedur des Betriebssystemskerns* und einen entsprechenden *System-Call*: Diese Prozedur benötigt auf einem Multiprozessor-Rechner ihrerseits eine unteilbare Maschineninstruktion vom Test-and-Set-Typ, ist also mit Variante 1 implementiert.
- durch eine selbst implementierte Prozedur, die nur dann System-Calls durchführt, wenn ein Prozess warten muss.

Grundsätzlich ist bei allen 3 Varianten zu überlegen, wie man verfährt, wenn ein Prozessor bei einer P-Operation keinen Erfolg hat. Es gibt dafür 2 Möglichkeiten:

- *Busy Wait*: die P-Operation wird so oft wiederholt, bis sie Erfolg hat. Dies macht nur auf einem Multiprozessor-Rechner Sinn; der potentielle Nachteil ist, dass beim Warten CPU-Zeit verbraucht wird. Wenn kritische Abschnitte sehr kurz sind und nur selten auftreten, ist Busy Wait akzeptabel.
- *Lazy Wait*: bei einer nicht erfolgreichen P-Operation legt sich ein Prozess schlafen (deaktiviert sich mittels System-Call selbst). Entweder muss er sich vom Betriebssystem periodisch wecken lassen und dann erneut die P-Operation versuchen, oder die anderen Prozesse müssen bei der V-Operation schlafende Prozesse, die auf die Freigabe des betreffenden Semaphors warten, durch System-Calls wecken (lassen). Der potentielle Nachteil von Lazy Wait ist, dass System-Calls sehr aufwändig sind; ein System-Call benötigt einige Hundert oder sogar einige Tausend Maschineninstruktionen.

Die beiden Varianten busy wait und lazy wait können entweder direkt in die Implementierung der P- und V-Operationen integriert werden, oder die Wahl kann den Prozessen selbst überlassen bleiben.

Die selbst implementierten Prozeduren zur Realisierung der P- und V-Operationen könnten mit Lazy Wait beispielsweise auf der folgenden Datenstruktur aufbauen:

```
struct latch {
     int Status;
     int NumProcesses;
     int Pids[MaxProcesses]; };
```

Diese Datenstruktur implementiert sozusagen einen leichtgewichtigen Semaphore, im Industriejargon auch “Latch” genannt, inklusive einer Warteschlange wartender Prozesse. Der Prozess auf Positions Pids[0] ist der jeweils aktuelle Besitzer des Semaphors. Zur Manipulation der Latch-Datenstruktur selbst wird die Test-and-Set-Methode auf dem Status-Feld verwendet. Der Code für P und V sieht dann folgendermaßen aus:

```
int P (latch) {
     while ( Test-and-Set(latch.Status, 1) != True ) { };
     latch.Pids[latch.NumProcesses] = getOwnProcessId();
     latch.NumProcesses++;
     if (latch.NumProcesses == 1)
        {latch.Status = 0}
     else {latch.Status = 0; sleep ( )};
     return True; };

int V (latch) {
     while ( Test-and-Set(latch.Status, 1) != True) { };
     for (i=0; i < latch.NumProcesses; i++)
        {latch.Pids[i] = latch.Pids[i+1]};
     latch.NumProcesses--;
     if (latch.NumProcesses == 0)
        {latch.Status = 0}
     else {latch.Status = 0, wakeup (Pids[0])};
     return True;};
```

Datenbanksysteme, Web-Server (z.B. Apache) und Betriebssysteme, die intern mit nebenläufigen Prozessen (häufig leichtgewichtigen Prozessen, sog. Threads) arbeiten, verwenden derartige Low-Level-Routinen zur Synchronisation der Zugriffe auf interne Datenstrukturen.

13.2 Synchronisation nebenläufiger Transaktionen

13.2.1 Transaktionskonzept

Eine *Transaktion* ist eine Folge von (Lese- und Änderungs-) Operationen auf einer oder mehreren Datenbanken, für die die zugrundeliegenden Datenbanksysteme die folgenden Eigenschaften (häufig *ACID-Eigenschaften* genannt) garantieren:

- *Atomarität (engl.: atomicity):*
  Die Änderungen einer Transaktion werden entweder vollständig oder gar nicht durchgeführt (Alles-oder-nichts-Eigenschaft). Im Fehlerfall oder auf "Wunsch" eines Programms werden bereits durchgeführte Änderungen einer Transaktion rückgängig gemacht.

- *Dauerhaftigkeit / Persistenz (engl.: durability / persistence):*
  Die Änderungen einer abgeschlossenen Transaktion gehen auch im Fall von Fehlern der Hardware oder Systemsoftware nicht verloren.

- *Isolation (engl.: isolation):*
  Die Transaktion sieht die Datenbanken so, als gäbe es keine parallelen Transaktionen, und die Änderungen der Transaktion werden bis zum Abschluß der Transaktion vor parallelen Transaktionen verborgen (Äquivalenz zum Einbenutzerbetrieb).

- *Integritätserhaltung (engl.: consistency-preservation, integrity-preservation):*
  Die Transaktion transformiert die Datenbanken von einem - bezüglich der spezifizierten Integritätsbedingungen - konsistenten Zustand in einen anderen konsistenten Zustand.

Die Eigenschaften der Atomarität und Persistenz bedeuten, daß Datenbankanwendungsprogramme größtenteils so geschrieben werden können, als gäbe es keine Fehler der Hardware oder Systemsoftware. Die Transaktionsverwaltung eines DBS stellt mit der **Recovery-Komponente** weitreichende Fehlertoleranzmaßnahmen zur Verfügung, und zwar transparent für die Anwendungsentwicklung (und natürlich erst recht transparent für den Endbenutzer).

Die Eigenschaft der Isolation bedeutet, daß Datenbankanwendungsprogramme so geschrieben werden können, als würde das DBS (ggf. sogar ein verteiltes DBS oder mehrere DBS in einem verteilten System) im Einbenutzerbetrieb arbeiten, obwohl ein DBS in der Regel aus Leistungsgründen effektiv im Mehrbenutzerbetrieb arbeitet. Die Transaktionsverwaltung eines DBS stellt mit der **Concurrency-Control-Komponente** automatische Synchronisationsmaßnahmen zur Verfügung, und zwar transparent für die Anwendungsentwicklung (und natürlich erst recht für den Endbenutzer).

Die Eigenschaft der Integritätserhaltung wird durch die Transaktionsverwaltung nur unterstützt, nicht aber voll erzwungen. Sie kann nur in Verbindung mit entsprechenden Maßnahmen der Anwendungsentwicklung erzwungen werden. Es wird erwartet, daß jedes Anwendungsprogramm bei Beendigung einer Transaktion entsprechende Integritätsprüfungen durchführt oder anstößt (z.B. über Trigger). Falls dann eine Transaktion im fehlerlosen Einbenutzerbetrieb integritätserhaltend ist, garantiert die Transaktionsverwaltung, daß die Integritätsbewahrung auch im Mehrbenutzerbetrieb und bei Auftreten von Hardware- und Systemsoftwarefehlern gilt.

Transaktionen werden dynamisch durch entsprechende DBS-Aufrufe des Anwendungsprogramms erzeugt. In SQL gibt es dafür drei Anweisungen:

- CONNECT ... (BOT = Begin-of-Transaction):
  kennzeichnet den Beginn einer Transaktion.
- COMMIT WORK (EOT = End-of-Transaction):
  kennzeichnet das ("erfolgreiche") Ende einer Transaktion
  und erklärt alle Änderungen der Transaktion für gültig.
- ROLLBACK WORK (RBT = Rollback-Transaction, häufig auch Abort genannt):
  kennzeichnet das ("erfolglose") Ende einer Transaktion
  und erklärt alle Änderungen der Transaktion für ungültig.

Implizit kennzeichnen EOT und RBT außerdem den Beginn einer neuen Transaktion, die bis zum nächsten EOT, RBT oder DISCONNECT dauert.

Beispiele für simple Transaktionsprogramms mit Embedded SQL:

DebitCredit:

```
void main ( ) {
   EXEC SQL BEGIN DECLARE SECTION
      int b /*balance*/, a /*accountid*/, amount;
   EXEC SQL END DECLARE SECTION;
   /* read user input */
   scanf („%d %d", &a, &amount);
   /* read account balance */
   EXEC SQL Select Balance into :b From Account
      Where Account_Id = :a;
   /* add amount (positive for debit, negative for credit) */
   b = b + amount;
   /* write account balance back into database */
   EXEC SQL Update Account
      Set Balance = :b Where Account_Id = :a;
   EXEC SQL Commit Work;
}
```

FundsTransfer:

```
void main ( ) {
   /* read user input */
   scanf („%d %d %d", &sourceid, &targetid, &amount);
   /* subtract amount from source account */
   EXEC SQL Update Account
      Set Balance = Balance - :amount Where Account_Id = :sourceid;
   /* add amount to target account */
   EXEC SQL Update Account
      Set Balance = Balance + :amount Where Account_Id = :targetid;
   EXEC SQL Commit Work;
}
```

13.2.2 Probleme bei unkontrollierter (Quasi-) Parallelität

Die (quasi-) parallele Ausführung von Transaktionen ist wünschenswert, um möglichst gute Antwortzeiten zu erzielen. Dazu müssen Plattenzugriffe und CPU-Arbeit für verschiedene Transaktionen parallel erfolgen. Um aber auch korrekte Resultate zu gewährleisten, muß die Parallelität u.U. wieder eingeschränkt werden.

Problemtyp "verlorene Änderung" (lost update):

P1	Time	P2
	/* x = 100 */	
r (x)	1	
	2	r (x)
x := x+100	4	x := x+200
w (x)	5	
	/* x = 200 */	
	6	w (x)
	/* x = 300 */	

Kern des Problems ist die nebenläufige Ausführung von Read- und Write-Schritten:
R1(x) R2(x) W1(x) W2(x)

Problemtyp "inkonsistentes Lesen":

P1	Time	P2
	1	r (x)
	2	x := x – 10
	3	w (x)
sum := 0	4	
r (x)	5	
r (y)	6	
sum := sum +x	7	
sum := sum + y	8	
	9	r (y)
	10	y := y + 10
	11	w (y)

Kern des Problems ist die nebenläufige Ausführung von Read- und Write-Schritten:
R2(x) W2(x) R1(x) R1(y) R2(y) W2(y)

Problemtyp "Dominoeffekt" (dirty read):

P1	Time	P2
r (x)	1	
x := x + 100	2	
w (x)	3	
	4	r (x)
	5	x := x - 100
failure & rollback	6	
	7	w (x)

Kern des Problems ist die nebenläufige Ausführung von Read-, Write- und Abort-Schritten (sowie potentiellen Commit-Schritten):
R1(x) W1(x) R2(x) W2(x) A1 ... C2 (mit potentiellem C2).

Probleme mit Lösungsansätzen mittels Semaphoren

Man könnte versuchen, die erwähnten Synchronisationsprobleme mittels Semaphoren zu lösen. Dabei wären zwei Varianten verfolgenswert:

- Wir führen einen Semaphor mutex für die gesamt Datenbank ein. Eine Transaktion führt vor ihrem ersten Datenbankzugriff eine P-Operation auf mutex durch und nach ihrem letzten Datenbankzugriff eine V-Operation. Dies wäre korrekt, würde aber den gesamten Betrieb zwangsweise sequentialisieren. Durch Einführung von Lese- und Schreibmodi bzw. je einen Semaphor für Lesen und Schreiben könnte man dies graduell verbessern, bei hoher Last von Änderungstransaktionen aber wäre die Methode völlig unakzeptabel.
- Wir führen pro Datenbankobjekt einen Semaphor ein. Vor dem Zugriff auf Objekt x wird P auf dem Semaphor für x aufgerufen und nach dem Zugriff V. Dies alleine aber genügt nicht, um korrekte Ausführung sicherzustellen, wie der folgende Beispielablauf (mit von oben nach unten fortschreitender Zeit) zeigt:

Prozess 1: FundsTransfer von a nach b	*Prozess 2: Prüfen der Summe von a und b*
shared semaphore mutex_a;	
shared semaphore mutex_b;	
P(mutex_a);	
Update Konto Set Stand = Stand – 3000 Where KontoNr = a;	
V(mutex_a);	
	P(mutex_a);
	Select Stand From Konto Where KontoNr = a;
	V(mutex_a);
	P(mutex_b); ...; V(mutex_b);
P(mutex_b); ...; V(mutex_b);	

Eine korrekte Lösung wäre dagegen, wenn beide Prozesse alle ihre V-Operationen erst nach allen Datenbankzugriffen (also z.B. mit dem Commit Work) aufrufen.
Insgesamt zeigt diese Überlegung, dass Semaphore nur Mechanismen sind, und dass man zusätzliche Überlegungen – insbesondere auch Theoriebildung – für korrekte Synchronisationsstrategien braucht. Die andiskutierten Semaphor-basierten Lösungen haben folgende Nachteile:

- Es ist – ohne weitere Theorie – unklar, wann jeweils ein Semaphor frühestens freigegeben werden darf. Anwendungsprogrammierer wären hinsichtlich des korrekten Umgangs mit Semaphoren überfordert, da man Wechselwirkungen des eigenen Programms mit allen anderen, potentiell nebenläufig ausgeführten Programmen beachten muss (selbst, wenn man den Code dieser Programme gar nicht kennt!).
- Es ist unklar, wie man mit Insert- und Delete-Operationen auf der Datenbank oder mit prädikatorientierten Operationen umgeht.
- Es ist nicht einfach ersichtlich, ob es durch die Verwendung mehrerer Semaphore womöglich zu Deadlocks (Verklemmungen, zyklischen Wartesituationen) kommen kann. Falls es dazu kommt, ist unklar, wie man solche Situationen erkennt und behebt.
- Die Verwaltung sehr vieler Semaphore (z.B. als Unix-Semaphore) wäre extrem aufwändig.

Konsequenz: Das DBS sollte die notwendigen Semaphore und die P- und V-Operationen darauf selbst automatisch generieren!

13.2.3 Korrektheitskriterium Serialisierbarkeit

Intuitives Kriterium:

Es dürfen nur (quasi-) parallele Abläufe von Transaktionen zugelassen werden, die zu einer sequentiellen Ausführung der mit Commit beendeten Transaktionen äquivalent sind. Dadurch werden alle Anomalien wie "verlorene Änderungen", "inkonsistentes Lesen", usw. vermieden.

Formalisierung des intutitiven Korrektheitskriteriums:

- Modellbildung für (quasi-) parallele Abläufe von Transaktionen
- Definition der Äquivalenz von Abläufen
- Entwicklung (beweisbar) korrekter Concurrency-Control-Verfahren zur automatischen Synchronisation durch das DBS

Begriffsbildung und Definitionen:

Einschränkung für den Rahmen der Vorlesung:
Es wird angenommen, daß stets alle Transaktionen mit Commit beendet werden.
(Für den allgemeinen Fall siehe Theorie der Rücksetzbarkeit und verwandte Theorien.)

Ein **Schedule** s ist ein Tripel (T, A, <) mit:
- T ist eine Menge von Transaktionen (eigentlich nur Transaktionsnummern).
- A ist eine Menge von (Datenbankzugriffs-) Aktionen der Form Ri (x) oder Wi (x), wobei
  - R und W für Lesezugriffe (Reads) und Schreibzugriffe (Writes) stehen,
  - x ein Objekt der Datenbank ist, auf das sich der Zugriff bezieht (z.B. eine Seite) und
  - jede Aktion zu genau einer Transaktion Ti gehört.
- < ist eine partielle Ordnung auf der Aktionenmenge A, wobei für jedes Aktionenpaar $\langle a, b\rangle \in A \times A$ gelten soll:
  Wenn a und b auf dasselbe Datenbankobjekt zugreifen und a oder b ein Schreibzugriff ist, dann müssen a und b bezüglich < geordnet sein; es muß also $(a < b) \vee (b < a)$ gelten.

Zwei Aktionen $a, b \in A$ in einem Schedule s = (T, A, <) sind in **Konflikt**, wenn
- a und b auf dasselbe Datenbankobjekt zugreifen und a oder b ein Schreibzugriff ist und
- a und b zu verschiedenen Transaktionen gehören.

Die beiden Aktionen a und b werden auch als "Konfliktpaar" bezeichnet.

Bemerkungen:
- Es ist wichtig, daß die Aktionen eines Konfliktpaars geordnet sind, damit überhaupt eine Aussage über deren Effekt möglich ist.
- Aktionen, die nicht in Konflikt sind, können innerhalb eines Schedules in beliebiger Reihenfolge ausgeführt werden, ohne daß dadurch unterschiedliche Effekte entstehen.
  Mit anderen Worten:
  Um zu entscheiden, ob zwei Schedules bezüglich ihrer Effekte äquivalent sind, ist nur die Reihenfolge der Aktionen in den Konfliktpaaren von Bedeutung.

Ein Schedule s = (T, A, <) heißt **seriell**, wenn für je zwei Transaktionen Ti, Tj ∈ T gilt, daß
entweder alle Aktionen von Ti bezüglich < vor allen Aktionen von Tj liegen
oder alle Aktionen von Tj vor allen Aktionen von Ti.
In einem seriellen Schedule gibt es somit keine (quasi-) parallelen Transaktionen.

Beispiele:
(Es wird in den Beispielen nur A angegeben; T steckt implizit in den Indizes der Aktionen,
und < ist implizit durch die Anordnung der Aktionen von links nach rechts gegeben.
In den Beispielen ist < immer eine totale Ordnung.)

Schedule s1: R1 (a) W1 (a) R2 (a) R2 (b) R1 (b) W1 (b)

Schedule s2: R1 (a) W1 (a) R2 (a) R3 (b) R2 (b) W2 (b) R3 (c) W3 (c) R1 (c)

Schedule s2': R3 (b) R3 (c) W3 (c) R1 (a) W1 (a) R1 (c) R2 (a) R2 (b) W2 (b)

Konfliktpaare für s1: <W1(a), R2(a)>, <R2(b), W1(b)>

Konfliktpaare für s2: <W1(a), R2(a)>, <R3(b), W2(b)>, <W3(c), R1(c)>

Konfliktpaare für s2': <W3(c), R1(c)>, <R3(b), W2(b)>, <W1(a), R2(a)>

Definition der Äquivalenz von Schedules:

Zwei Schedules s und s' sind **(konflikt-) äquivalent**, wenn sie dieselben Transaktionsmengen und Aktionsmengen haben und in beiden Schedules dieselben Konfliktpaare (jeweils mit derselben Aktionsreihenfolge) vorliegen.

Definition der Korrektheit von nichtseriellen Schedules:

Ein Schedule s ist **(konflikt-) serialisierbar**, wenn er zu einem seriellen Schedule äquivalent ist.

Bemerkungen:
- Serialisierbarkeit ist die Formalisierung des intuitiven Kriteriums der Äquivalenz zum Einbenutzerbetrieb.
- Damit Serialisierbarkeit als Korrektheitskriterium Sinn macht, sind folgende (nicht beweisbare) Annahmen implizit zugrundegelegt:
  - Ein serieller Schedule ist korrekt (unter der wiederum impliziten Annahme, daß das Transaktionsprogramm an sich korrekt, insbesondere integritätserhaltend ist).
  - Es wird angenommen, daß alle Transaktionen von verschiedenen, unabhängigen Benutzern stammen und daß für diese Benutzer jede beliebige sequentielle Ausführung der Transaktionen akzeptabel ist.

Algorithmische Überprüfung der Serialisierbarkeit eines Schedules:

Sei s = (T, A, <) ein Schedule. Der **Abhängigkeitsgraph** für diesen Schedule ist ein gerichteter Graph mit

- Transaktionen als Knoten und
- einer Kante zwischen Ti und Tj genau dann, wenn es ein Konfliktpaar <a, b> in s gibt, so daß a zu Ti und b zu Tj gehört und a < b gilt.

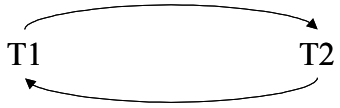
Satz:
Ein Schedule ist serialisierbar genau dann, wenn sein Abhängigkeitsgraph azyklisch ist, d.h. keine Zyklen enthält.

Man erhält einen zum Schedule äquivalenten seriellen Schedule - eine **Serialisierung** - durch topologische Sortierung des Abhängigkeitsgraphen (eine totale Ordnung der Transaktionen, die die durch die Kanten des Graphen gegebene partielle Ordnung beinhaltet):

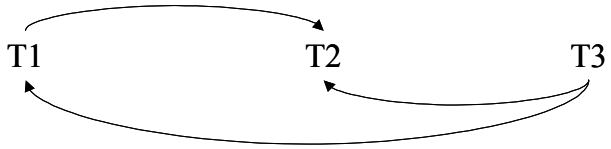
- Man wähle eine beliebige Quelle des Graphen (d.h. einen Knoten ohne eingehende Kanten) und entferne diese sowie alle davon ausgehenden Kanten.
- Falls der Graph noch Knoten hat, wiederholt man diese Prozedur für den verbleibenden Graphen.

Beispiele:

Abhängigkeitsgraph für s1:



Abhängigkeitsgraph für s2:



13.2.4 Das strikte Zweiphasen-Sperrprotokoll

Das DBS fordert für eine Transaktion vor dem Zugriff auf ein Objekt eine Sperre auf dem Objekt an. Sperren sind im Prinzip Semaphore, die durch das DBS selbst implementiert werden. Dazu stellt die Concurrency-Control-Komponente die folgenden zwei Operationen für Sperren zur Verfügung:

- Lock (Transaktion Ti, Objekt x, Modus m) und
- Unlock (Transaktion Ti, Objekt x).

Das DBS kann nie zwei Transaktionen gleichzeitig eine Sperre auf demselben Objekt (in unverträglichen Sperrmodi) geben. Im Sperrkonfliktfall muß die "spätere" Transaktion bis zur Sperrfreigabe warten. Die Concurrency-Control-Komponente legt in diesem Fall die Transaktion schlafen und weckt sie bei Sperrfreigabe automatisch wieder auf. Zu diesem Zweck verwaltet die Concurrency-Control verschiedene Warteschlangen.

Beispiel:
(mit der Notation Li (x) für Lock (Ti, x, Exclusive) und Ui (x) für Unlock (Ti, x)**:**

```
L1 (a)  W1 (a)                  L1 (c)  W1 (c)          U1 (a)  U1 (c)
              L2 (b)  W2 (b)            L2 (c) - - - - - - - - - - - -          W2 (c) U2 (b)  U2 (c)
```

Zweiphasen-Sperrprotokoll (Two-Phase Locking, 2PL):

Regel 1: Bevor eine Transaktion auf ein Objekt zugreift, muß das Objekt für die Transaktion gesperrt werden ("Wachstumsphase" der Transaktion).

Regel 2: Nachdem eine Transaktion eine Sperre freigegeben hat, darf sie keine weiteren Sperren mehr erwerben ("Schrumpfungsphase" der Transaktion).

Satz:
Das Zweiphasen-Sperrprotokoll garantiert, daß alle entstehenden Schedules serialisierbar sind.

Beweis:

Sei s ein Schedule, der unter dem 2PL entstanden ist.
Annahme: s ist nicht serialisierbar.
$\Rightarrow$ es gibt einen Zyklus im Abhängigkeitsgraphen
$T1 \rightarrow T2 \rightarrow T3 \rightarrow ... \rightarrow Tk \rightarrow T1$,
also Konfliktpaare
<a1 (x1), a2 (x1)>, <a2 (x2), a3 (x2)>, ..., <ak (xk), a1 (xk)>
Da Objekte vor dem Zugriff gesperrt werden und andere Transaktionen erst nach der entsprechenden Sperrfreigabe auf das Objekt zugreifen können, gilt bezüglich der Reihenfolge der Lock- und Unlock-Operationen:
U1 (x1) < L2 (x1), U2 (x2) < L3 (x2), ..., Uk (xk) < L1 (xk)
Wegen Regel 2 des 2PL gilt ferner:
L2 (x1) < U2 (x2), L3 (x2) < U3 (x3), ...
$\Rightarrow$ U1 (x1) < L2 (x1) < U2 (x2) < L3 (x2) < U3 (x3) < ... < Uk (xk) < L1 (xk)
Dann aber wäre wegen U1 (x1) < L1 (xk) die Regel 2 für Transaktion T1 verletzt.
Widerspruch ! $\Rightarrow$ Behauptung

Beobachtungen:

- Das 2PL ist nicht ausreichend, um Domino-Effekte zu verhindern (siehe Beispiel für Domino-Effekte).
- Um das 2PL korrekt anwenden zu können, muß man für jede Transaktion wissen, ab welchem Zeitpunkt im Laufe ihrer Ausführung keine weiteren Objekte mehr benötigt werden, die nicht bereits für die Transaktion gesperrt sind. Ab diesem Zeitpunkt können Sperren freigegeben werden.
  Dieses A-priori-Wissen ist unrealistisch, da die von der Transaktion benötigten Objekte von der Ausführungsdynamik des zugrundeliegenden Transaktionsprogramms abhängen.

Striktes Zweiphasen-Sperrprotokoll (Strict Two-Phase Locking, Strict 2PL):

Regel 1: wie beim allgemeinen 2PL

Regel 2: Halte alle jemals für eine Transaktion erworbenen Sperren bis zum Ende der Transaktion (Abschluß des Commit oder Rollback) und gebe sie dann auf einmal frei. (Die Schrumpfungsphase fällt also zu einem einzigen Zeitpunkt zusammen.)

Bemerkungen:

- Das strikte 2PL gewährleistet die Äquivalenz (in einem weitergehenden Sinne) eines parallelen Ablaufs von Transaktionen (die mit Commit oder Rollback beendet sind) zu einem sequentiellen Ablauf derjenigen Transaktionen, die mit Commit beendet sind (ohne Beweis). Das strikte 2PL verhindert also auch Domino-Effekte.
- Das strikte 2PL ist als Concurrency-Control-Protokoll in praktisch allen kommerziellen DBS implementiert.
- Das 2PL kann auf verschiedenen Sperrgranulaten implementiert werden:
  - Seitensperren (in den meisten DBS)
  - Relationssperren oder Tablespace-Sperren (sehr grob und damit restriktiv)
  - Tupelsperren und Sperren auf Indexeinträgen
    (sehr feines Granulat und daher bezüglich der Mehrbenutzerleistung am besten, aber auch sehr komplex und daher nur in den besseren DBS implementiert)
  - Kombinationen (die meisten Transaktionen sperren z.B. Seiten, einige lange Transaktionen aber sperren ganze Relationen), sog. Mehrgranulats-Sperrverfahren
- Das 2PL kann mit verschiedenen Sperrmodi realisiert werden. In der Regel sind die Sperrmodi
  Exclusive - für Writes - und *Shared* - für Reads -
  vorgesehen.
  Der Sperrmodus für einen Zugriff wird jeweils vom DBS automatisch gewählt.
  Für Sperrmodi muß eine *Kompatibilitätsmatrix (Verträglichkeitsmatrix)* spezifiziert sein, die festlegt, ob eine Sperranforderung im Modus m gewährt werden darf, wenn das zu sperrende Objekt bereits für eine andere Transaktion im Modus k gesperrt ist.

Beispiel einer Kompatibilitätsmatrix:

		Transaktion fordert eine Sperre an im Modus	
		Shared	*Exclusive*
Objekt ist gesperrt im Modus	*Shared*	+	-
	Exclusive	-	-

Deadlocks (Verklemmungen):

Das 2PL kann zu Deadlocks führen, bei denen zwei oder mehr Transaktionen in einem Zyklus wechselseitig auf die Freigabe von Sperren warten. Die Concurrency-Control-Komponente erkennt solche Deadlocks, indem sie über alle Wartebeziehungen in Form eines Wartegraphen Buch führt und - periodisch oder bei jedem Sperrkonflikt - einen Zyklustest für den Wartegraphen durchführt. Falls ein Zyklus gefunden wird, wird eine der am Zyklus beteiligten Transaktionen als Deadlock-Opfer ausgewählt und zurückgesetzt (von der Recovery-Komponente des DBS).

Das Anwendungsprogramm erhält in diesem Fall einen speziellen SQLCODE für die SQL-Anweisung, bei der der Deadlock entstand. Das Programm sollte in einem solchen Fall selbst den erneuten Start der Transaktion veranlassen (ggf. nach erneuten Initialisierungen), damit der Deadlock und seine Behandlung gegenüber dem Endbenutzer transparent ist.

Beispiele:
(mit der Notation Xi (x) für Lock (Ti, x, Exclusive) und Si (x) für Lock (Ti, x, Shared))

Beispiel1:

```
X1 (a)  W1 (a)                          X1 (b) - - - - - - - -
                X2 (b)  W2 (b) X2 (a) - - - - - - - - - - - -
```

Beispiel 2:

```
S1 (a)  R1 (a)                  X1 (a) - - - - - - - - - - - -
                S2 (a)  R2 (a)          X2 (a) - - - - - - - -
```

Der zweite Fall - Deadlock aufgrund von Sperrkonversionen (Verschärfung einer Shared-Sperre in eine Exclusive-Sperre) - tritt in der Praxis potentiell häufig auf. Er kann durch zusätzliche Sperrmodi weitgehend vermieden werden. Diese erfordern aber u.U. spezielle Hinweise durch das Programm (die der Programmierer vorsehen muß). Aus diesem Grund hat SQL spezielle Anweisungszusätze wie z.B. DECLARE C CURSOR FOR SELECT ... *FOR UPDATE* ...

Weiterführende Literatur zu Kapitel 13:

- G. Weikum, G. Vossen: Transactional Information Systems – Theory, Algorithms, and the Practice of Concurrency Control and Recovery, Morgan Kaufmann, 2001
- J. Gray, A. Reuter: Transaction Processing – Concepts and Techniques, Morgan Kaufman, 1993