

Kapitel 15: Concurrency Control – Synchronisation von Prozessen und Transaktionen

15.1 Synchronisation nebenläufiger Prozesse

15.2 Synchronisation nebenläufiger Transaktionen

15.2.1 Probleme

15.2.2 Modellbildung

15.2.3 Protokolle

15.1 Synchronisation nebenläufiger Prozesse

Nebenläufigkeit (Quasi-Parallelität, Concurrency):

Verschränkte Ausführung der Schritte mehrerer Prozesse

z.B.: p11, p21, p12, p13, p22 für Prozesse p1, p2

→ Gefährdung der Konsistenz gemeinsamer Datenstrukturen

Lösung:

wechselseitiger Ausschluss (mutual exclusion)

bei kritischen Abschnitten (critical sections)

Beispiel: Programm mit Schritten p1, p2, p3, p4 und
kritischem Abschnitt p2, p3

2 Instantiierungen: Prozesse p, p'

→ p1, p1', p2, p3, p2', p3', p4', p4 ist erlaubt

→ p1, p2, p1', p2', p3, p3', p4, p4' ist nicht erlaubt

Wechselseitiger Ausschluss mit Semaphoren

Semaphor (Ampel):

ganzzahlige, nichtnegative Variable mit unteilbaren Operationen

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

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

Wechselseitiger Ausschluss für kritischen Abschnitt von
Programm p:

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

Implementierung von Semaphoren

- 1) unteilbare Maschineninstruktion Test-and-Set <addr> <val>
- 2) ununterbrechbare Prozedur des Betriebssystemskerns
(z.B. System-Call semop in Unix)
- 3) selbst implementierte leichtgewichtige Prozedur

jeweils mit

- a) busy wait oder
- b) lazy wait

Latches: leichtgewichtige Semaphore

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

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

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

15.2 Synchronisation nebenläufiger Transaktionen

Eine Transaktion ist eine Folge von Server-Aktionen mit ACID-Eigenschaften:

- Atomarität (atomicity): Alles oder Nichts
- Dauerhaftigkeit/Persistenz (durability)
- Isolation (isolation)
- Integritätserhaltung (consistency preservation)

ACID-Vertrag zwischen Client und transaktionalem Server:

- Alle Aktionen vom Beginn einer Transaktion bis zum Commit Work bilden eine ACID-Einheit (Alles-Fall bei Atomarität).
- Alle Aktionen vom Beginn bis zum Rollback Work (Abort) sind isoliert und werden rückgängig gemacht (Nichts-Fall bei Atomarität)

Transaktionsprogramm Debit/Credit

```
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;  
}
```

OLTP Example 1: Concurrent Executions

P1	Time	P2
Select Balance Into :b1 From Account Where Account_Id = :a	1	
/* b1=0, a.Balance=100, b2=0 */		
	2	Select Balance Into :b2 From Account Where Account_Id = :a
b1 = b1-50	/* b1=100, a.Balance=100, b2=100 */	
	3	
	/* b1=50, a.Balance=100, b2=100 */	
	4	b2 = b2 +100
Update Account Set Balance = :b1 Where Account_Id = :a	/* b1=50, a.Balance=100, b2=200 */	
	5	
	/* b1=50, a.Balance=50, b2=200 */	
	6	Update Account Set Balance = :b2 Where Account_Id = :a
	/* b1=50, a.Balance=200, b2=200 */	

*Observation: concurrency or parallelism may cause inconsistencies
requires concurrency control for „isolation“*

Transaktionsprogramm Überweisung

```
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; }  
    ↙
```

*Observation: failures may cause inconsistencies
require recovery for „atomicity“ and „durability“*

Problemtyp „Verlorene Änderung“

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



„lost update“

Kern des Problems: 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	$w(y)$



„inconsistent read“

Kern des Problems: $R2(x) \ W2(x) \ R1(x) \ R1(y) \ R2(y) \ W2(y)$

Problemtyp „Dominoeffekt“

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)



„dirty read“

Kern des Problems: R1(x) W1(x) R2(x) A1 W2(x) ... [C2]

Warum Semaphore nicht genügen

- ein Semaphore für ganze DB führt zur Zwangssequentialisierung
- jeweils ein Semaphore pro Tupel ist (noch) keine Lösung:
 - unklar, wann V-Operation möglich ist
 - unklar, wie Insert-/Delete-Operationen und prädiktorientierte Operationen zu behandeln sind
 - Umgang mit Deadlocks unklar
 - unakzeptabler Overhead bei vielen Semaphoren

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);

Modellbildung und Korrektheitskriterium

Intuitiv:

nur Abläufe zulassen, die äquivalent zu sequentieller Ausführung sind

Formalisierung:

- Modellbildung für (quasi-) parallele Abläufe → Schedules
- Definition der Äquivalenz → Konfliktäquivalenz
- Definition zulässiger Abläufe → (Konflikt-) Serialisierbarkeit
- Entwicklung beweisbar korrekter Concurrency-Control-Verfahren zur automatischen Synchronisation durch das DBS

Schedules und Konflikte

Ein **Schedule** s ist ein Tripel $(T, A, <)$ mit:

- Transaktionsmenge T
- Aktionsmenge A mit Aktionen der Form $R_i(x)$ oder $W_i(x)$
- Partieller Ordnung $<$ auf der Aktionenmenge A ,
wobei für jedes Paar $\langle a, b \rangle \in A \times A$ gilt:
wenn a und b auf dasselbe Objekt zugreifen und a oder b schreibt,
muß $(a < b) \vee (b < a)$ gelten.

Aktionen $a, b \in A$ in einem Schedule $s = (T, A, <)$ sind in **Konflikt**, wenn beide auf dasselbe Objekt zugreifen und a oder b schreibt und a, b zu verschiedenen Transaktionen gehören.

Schedule $s = (T, A, <)$ heißt **seriell**, wenn für je zwei $T_i, T_j \in T$ gilt: entweder liegen alle Aktionen von T_i bezüglich $<$ vor allen Aktionen von T_j oder umgekehrt.

Konflikt-Äquivalenz und Serialisierbarkeit

Schedules s und s' mit denselben Transaktionen und Aktionen sind **(konflikt-) äquivalent**, wenn sie dieselben Konfliktpaare haben.

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

Sei $s = (T, A, <)$ ein Schedule. Der **Abhängigkeitsgraph** $G(s)$ von s ist ein gerichteter Graph mit

- Knotenmenge T und
- einer Kante von T_i nach T_j , wenn es ein Konfliktpaar $\langle a, b \rangle$ in s gibt, bei dem a zu T_i und b zu T_j gehört.

Satz: s ist serialisierbar g.d.w. $G(s)$ azyklisch ist.

Beispiele

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

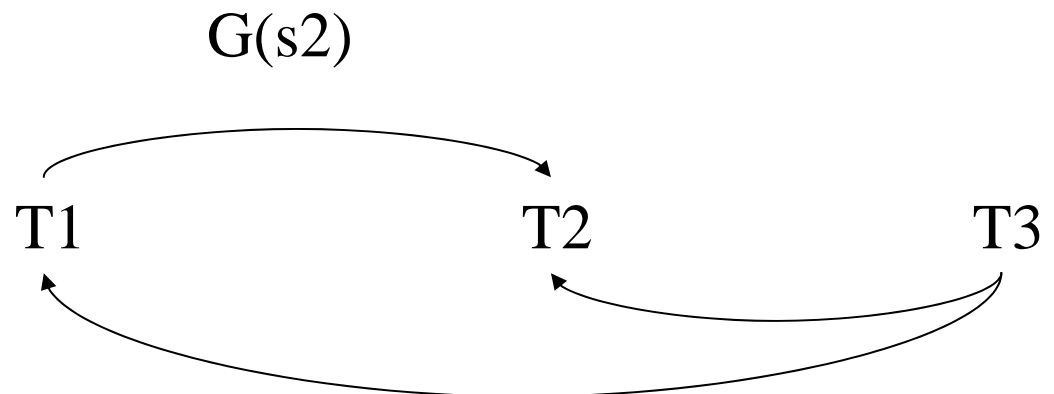
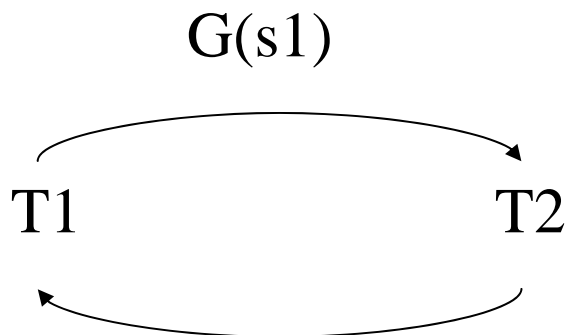
s2: R1 (a) W1 (a) R2 (a) R3 (b) R2 (b) W2 (b) R3 (c) W3 (c) R1 (c)

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

Konfliktpaare s1: $\langle W1(a), R2(a) \rangle, \langle R2(b), W1(b) \rangle$

Konfliktpaare s2: $\langle W1(a), R2(a) \rangle, \langle R3(b), W2(b) \rangle, \langle W3(c), R1(c) \rangle$

Konfliktpaare s3: $\langle W3(c), R1(c) \rangle, \langle R3(b), W2(b) \rangle, \langle W1(a), R2(a) \rangle$



Zweiphasen-Sperrprotokoll

DBS-interne Synchronisationsprimitive:

- Lock (Transaktion T_i , Objekt x , Modus m)
- Unlock (Transaktion T_i , Objekt x)

Beispiel:

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).

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

Satz: 2PL garantiert Serialisierbarkeit.

Striktes Zweiphasen-Sperrprotokoll

Striktes Zweiphasen-Sperrprotokoll (Strict Two-Phase Locking, S2PL):

Regel 1: wie bei 2PL

Regel 2: Alle jemals erworbenen (exklusiven) Sperren müssen bis zum Commit oder Abort der Transaktion gehalten werden.

Satz: S2PL garantiert Serialisierbarkeit und vermeidet Dominoeffekte.

Sperrprotokolle verwenden in der Regel operationsspezifische Sperrmodi (z.B. Shared-Sperren für Lesen und Exclusive-Sperren für Schreiben) mit einer entsprechenden Sperrmoduskompatibilitätsmatrix:

		Transaktion fordert eine Sperre an im Modus	
		<i>Shared</i>	<i>Exclusive</i>
Objekt ist gesperrt im Modus	<i>Shared</i>	+	-
	<i>Exclusive</i>	-	-

Deadlocks

- Zyklische Wartebeziehungen zwischen Transaktionen
- Zykluserkennung auf Waits-For-Graphen
- Opferauswahl und Rücksetzen

Beispiele:

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

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