Kapitel 13

Mehrbenutzersynchronisation

13.1 Multiprogramming

Unter $Multiprogramming$ versteht man die nebenläufige, verzahnte Ausführung mehrerer Programme. Abbildung 13.1 zeigt exemplarisch die dadurch erreichte bessere CPU-Auslastung.

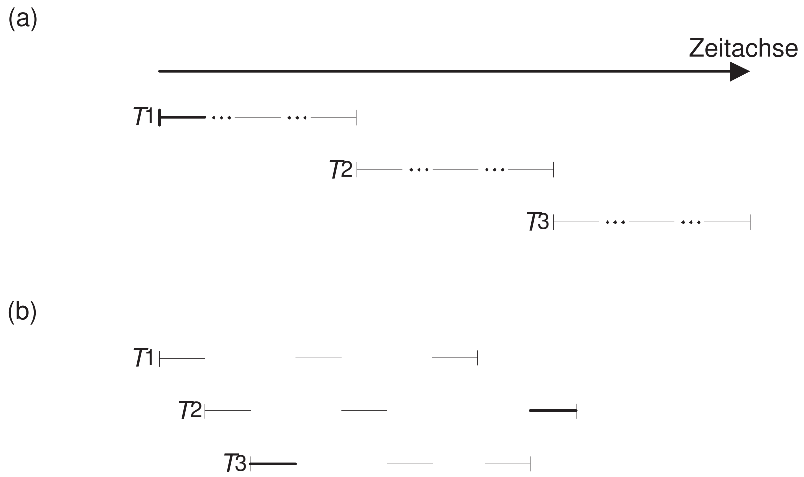

Abbildung 13.1: Einbenutzerbetrieb (a) versus Mehrbenutzerbetrieb (b)

13.2 Fehler bei unkontrolliertem Mehrbenutzerbetrieb

13.2.1 Lost Update

Transaktion T_1 transferiert 300,- DM von Konto A nach Konto B,
Transaktion T_2 schreibt Konto A die 3 % Zinseinkünfte gut.

Den Ablauf zeigt Tabelle 13.1. Die im Schritt 5 von Transaktion T_2 gutgeschriebenen Zinsen gehen verloren, da sie in Schritt 6 von Transaktion T_1 wieder überschrieben werden.

Schritt	T_1	T_2
1.	read(A, a_1)	
2.	$a_1 := a_1$ - 300	
3.		read(A, a_2)
4.		$a_2 := a_2$ * 1.03
5.		write(A, a_2)
6.	write(A, a_1)	
7.	read(B, b_1)	
8.	$b_1 := b_1 + 300$	
9.	write(B, b_1)	

Tabelle 13.1: Beispiel für Lost Update

13.2.2 Dirty Read

Transaktion T_2 schreibt die Zinsen gut anhand eines Betrages, der nicht in einem konsistenten Zustand der Datenbasis vorkommt, da Transaktion T_1 später durch ein **abort** zurückgesetzt wird. Den Ablauf zeigt Tabelle 13.2.

Schritt	T_1	T_2
1.	read(A, a_1)	
2.	$a_1 := a_1 - 300$	
3.	write(A, a_1)	
4.		read(A, a_2)
5.		$a_2 := a_2 * 1.03$
6.		write(A, a_2)
7.	read(B, b_1)	
8.	...	
9.	**abort**	

Tabelle 13.2: Beispiel für Dirty Read

13.2.3 Phantomproblem

Während der Abarbeitung der Transaktion T_2 fügt Transaktion T_1 ein Datum ein, welches T_2 liest. Dadurch berechnet Transaktion T_2 zwei unterschiedliche Werte. Den Ablauf zeigt Tabelle 13.3.

T_1	T_2
	select sum(KontoStand) **from** Konten;
insert into Konten **values** (C, 1000, ...);	
	select sum(KontoStand) **from** Konten;

Tabelle 13.3: Beispiel für das Phantomproblem

13.3 Serialisierbarkeit

Eine $Historie$, auch genannt $Schedule$, für eine Menge von Transaktionen ist eine Festlegung für die Reihenfolge sämtlicher relevanter Datenbankoperationen. Ein Schedule heißt $seriell$, wenn alle Schritte einer Transaktion unmittelbar hintereinander ablaufen. Wir unterscheiden nur noch zwischen $read$- und $write$-Operationen.

Zum Beispiel transferiere T_1 einen bestimmten Betrag von A nach B und T_2 transferiere einen Betrag von C nach A. Eine mögliche Historie zeigt Tabelle 13.4.

Schritt	T_1	T_2
1.	**BOT**	
2.	read(A)	
3.		**BOT**
4.		read(C)
5.	write(A)	
6.		write(C)
7.	read(B)	
8.	write(B)	
9.	**commit**	
10.		read(A)
11.		write(A)
12.		**commit**

Tabelle 13.4: Serialisierbare Historie

Offenbar wird derselbe Effekt verursacht, als wenn zunächst T_1 und dann T_2 ausgeführt worden wäre, wie Tabelle 13.5 demonstriert.

Schritt	T_1	T_2
1.	**BOT**	
2.	read(A)	
3.	write(A)	
4.	read(B)	
5.	write(B)	
6.	**commit**	
7.		**BOT**
8.		read(C)
9.		write(C)
10.		read(A)
11.		write(A)
12.		**commit**

Tabelle 13.5: Serielle Historie

Wir nennen deshalb das (verzahnte) Schedule $serialisierbar$.

Tabelle 13.6 zeigt ein Schedule der Transaktionen T_1 und T_3, welches nicht serialisierbar ist.

Schritt	T_1	T_3
1.	**BOT**	
2.	read(A)	
3.	write(A)	
4.		**BOT**
5.		read(A)
6.		write(A)
7.		read(B)
8.		write(B)
9.		**commit**
10.	read(B)	
11.	write(B)	
12.	**commit**	

Tabelle 13.6: Nicht-serialisierbares Schedule

Der Grund liegt darin, daß bzgl. Datenobjekt A die Transaktion T_1 vor T_3 kommt, bzgl. Datenobjekt B die Transaktion T_3 vor T_1 kommt. Dies ist nicht äquivalent zu einer der beiden möglichen seriellen Ausführungen T_1T_3 oder T_3T_1.

Im Einzelfall kann die konkrete Anwendungssemantik zu einem äquivalenten seriellen Schedule führen, wie Tabelle 13.7 zeigt.

Schritt	T_1	T_3
1.	**BOT**	
2.	read(A, a_1)	
3.	$a_1 := a_1 - 50$	
4.	write(A, a_1)	
5.		**BOT**
6.		read(A, a_2)
7.		$a_2 := a_2 - 100$
8.		write(A, a_2)
9.		read(B, b_2)
10.		$b_2 := b_2 + 100$
11.		write(B, b_2)
12.		**commit**
13.	read(B, b_1)	
14.	$b_1 := b_1 + 50$	
15.	write(B, b_1)	
16.	**commit**	

Tabelle 13.7: Zwei verzahnte Überweisungen

In beiden Fällen wird Konto A mit 150,- DM belastet und Konto B werden 150,- DM gutgeschrieben.

Unter einer anderen Semantik würde T_1 einen Betrag von 50,- DM von A nach B überweisen und Transaktion T_2 würde beiden Konten jeweils 3 % Zinsen gutschreiben. Tabelle 13.8 zeigt den Ablauf.

Schritt	T_1	T_3
1.	**BOT**	
2.	read(A, a_1)	
3.	$a_1 := a_1 - 50$	
4.	write(A, a_1)	
5.		**BOT**
6.		read(A, a_2)
7.		$a_2 := a_2 * 1.03$
8.		write(A, a_2)
9.		read(B, b_2)
10.		$b_2 := b_2 * 1.03$
11.		write(B, b_2)
12.		**commit**
13.	read(B, b_1)	
14.	$b_1 := b_1 + 50$	
15.	write(B, b_1)	
16.	**commit**	

Tabelle 13.8: Überweisung verzahnt mit Zinsgutschrift

Offenbar entspricht diese Reihenfolge keiner möglichen seriellen Abarbeitung $T_1 T_3$ oder $T_3 T_1$, denn es fehlen in jedem Falle Zinsen in Höhe von 3 % von 50,- DM = 1,50 DM.

13.4 Theorie der Serialisierbarkeit

Eine $Transaktion\ T_i$ besteht aus folgenden elementaren Operationen:

- r_i(A) zum Lesen von Datenobjekt A,
- w_i(A) zum Schreiben von Datenobjekt A,
- a_i zur Durchführung eines **abort**,
- c_i zur Durchführung eines **commit**.

Eine Transaktion kann nur eine der beiden Operationen **abort** oder **commit** durchführen; diese müssen jeweils am Ende der Transaktion stehen. Implizit wird ein **BOT** vor der ersten Operation angenommen. Wir nehmen für die Transaktion eine feste Reihenfolge der Elementaroperationen an.

Eine $Historie$, auch genannt $Schedule$, ist eine Festlegung der Reihenfolge für sämtliche beteiligten Einzeloperationen.

Gegeben Transaktionen T_i und T_j, beide mit Zugriff auf Datum A. Folgende vier Fälle sind möglich:

- r_i(A) und r_j(A) : kein Konflikt, da Reihenfolge unerheblich

- r_i(A) und w_j(A) : Konflikt, da Reihenfolge entscheidend

- w_i(A) und r_j(A) : Konflikt, da Reihenfolge entscheidend

- w_i(A) und w_j(A) : Konflikt, da Reihenfolge entscheidend

Von besonderem Interesse sind die $Konfliktoperationen$.

Zwei Historien H_1 und H_2 über der gleichen Menge von Transaktionen sind äquivalent (in Zeichen $H_1 \equiv H_2$), wenn sie die Konfliktoperationen der nicht abgebrochenen Transaktionen in derselben Reihenfolge ausführen. D. h., für die durch H_1 und H_2 induzierten Ordnungen auf den Elementaroperationen $<_{H_1}$ bzw. $<_{H_2}$ wird verlangt: Wenn p_i und q_j Konfliktoperationen sind mit $p_i <_{H_1} q_j$, dann muß auch $p_i <_{H_2} q_j$ gelten. Die Anordnung der nicht in Konflikt stehenden Operationen ist irrelevant.

13.5 Algorithmus zum Testen auf Serialisierbarkeit:

Input: Eine Historie H für Transaktionen $T_1, \ldots, T_k$.

Output: entweder: „nein, ist nicht serialisierbar" oder „ja, ist serialisierbar" + serielles Schedule

Idee: Bilde gerichteten Graph G, dessen Knoten den Transaktionen entsprechen. Für zwei Konfliktoperationen p_i, q_j aus der Historie H mit $p_i <_H q_j$ fügen wir die Kante $T_i \rightarrow T_j$ in den Graph ein.

Es gilt das **Serialisierbarkeitstheorem:**
Eine Historie H ist genau dann serialisierbar, wenn der zugehörige Serialisierbarkeitsgraph azyklisch ist. Im Falle der Kreisfreiheit läßt sich die äquivalente serielle Historie aus der topologischen Sortierung des Serialisierbarkeitsgraphen bestimmen.

Als Beispiel-Input für diesen Algorithmus verwenden wir die in Tabelle 13.9 gezeigte Historie über den Transaktionen T_1, T_2, T_3 mit insgesamt 14 Operationen.

Schritt	T_1	T_2	T_3
1.	r_1(A)		
2.		r_2(B)	
3.		r_2(C)	
4.		w_2(B)	
5.	r_1(B)		
6.	w_1(A)		
7.		r_2(A)	
8.		w_2(C)	
9.		w_2(A)	
10.			r_3(A)
11.			r_3(C)
12.	w_1(B)		
13.			w_3(C)
14.			w_3(A)

Tabelle 13.9: Historie H mit drei Transaktionen

Folgende Konfliktoperationen existieren für Historie H:

$$w_2(B) < r_1(B),$$

$$w_1(A) < r_2(A),$$

$$w_2(C) < r_3(C),$$

$$w_2(A) < r_3(A).$$

Daraus ergeben sich die Kanten

$$T_2 \rightarrow T_1,$$

$$T_1 \rightarrow T_2,$$

$$T_2 \rightarrow T_3,$$

$$T_2 \rightarrow T_3.$$

Den resultierenden Graph zeigt Abbildung 13.2

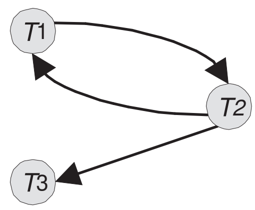

Abbildung 13.2: Der zu Historie H konstruierte Serialisierbarkeitsgraph

Da der konstruierte Graph einen Kreis besitzt, ist die Historie nicht serialisierbar.

13.6 Sperrbasierte Synchronisation

Bei der sperrbasierten Synchronisation wird während des laufenden Betriebs sichergestellt, daß die resultierende Historie serialisierbar bleibt. Dies geschieht durch die Vergabe einer $Sperre$ (englisch: $lock$).

Je nach Operation (**read** oder **write**) unterscheiden wir zwei Sperrmodi:

- **S** (shared, read lock, Lesesperre):
  Wenn Transaktion T_i eine S-Sperre für Datum A besitzt, kann T_i **read**(A) ausführen. Mehrere Transaktionen können gleichzeitig eine S-Sperre auf dem selben Objekt A besitzen.
- **X** (exclusive, write lock, Schreibsperre):
  Ein write(A) darf nur die eine Transaktion ausführen, die eine X-Sperre auf A besitzt.

Tabelle 13.10 zeigt die Kompatibilitätsmatrix für die Situationen NL (no lock), S (read lock) und X (write lock).

	NL	S	X
S	$\surd$	$\surd$	-
X	$\surd$	-	-

Tabelle 13.10: Kompatibilitätsmatrix

Folgendes Zwei-Phasen-Sperrprotokoll ($two\ phase\ locking$, $2PL$) garantiert die Serialisierbarkeit:

1. Jedes Objekt muß vor der Benutzung gesperrt werden.
2. Eine Transaktion fordert eine Sperre, die sie schon besitzt, nicht erneut an.
3. Eine Transaktion respektiert vorhandene Sperren gemäß der Verträglichkeitsmatrix und wird ggf. in eine Warteschlange eingereiht.
4. Jede Transaktion durchläuft eine $Wachstumsphase$ (nur Sperren anfordern) und dann eine $Schrumpfungsphase$ (nur Sperren freigeben).
5. Bei Transaktionsende muß eine Transaktion alle ihre Sperren zurückgeben.

Abbildung 13.3 visualisiert den Verlauf des 2PL-Protokolls. Tabelle 13.11 zeigt eine Verzahnung zweier Transaktionen nach dem 2PL-Protokoll.

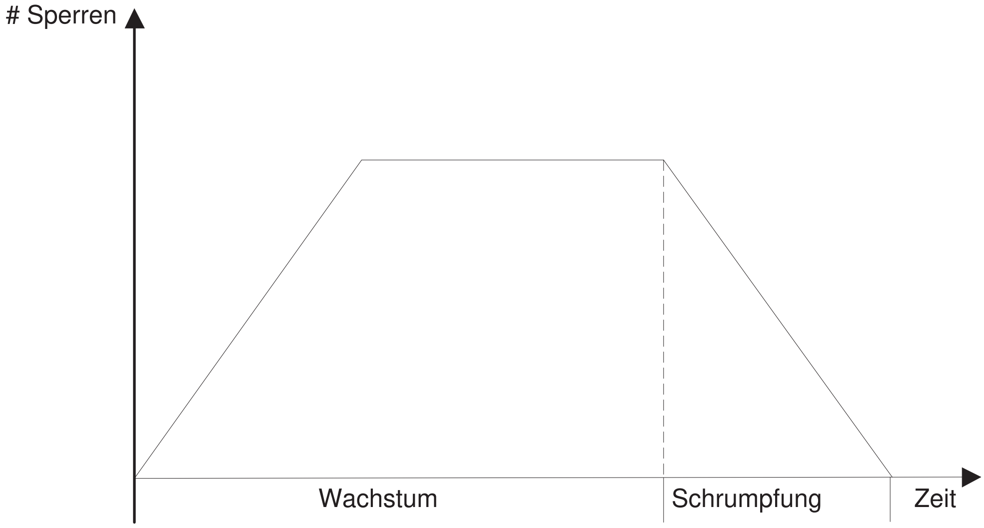

Abbildung 13.3: 2-Phasen-Sperrprotokoll

Schritt	T_1	T_2	Bemerkung
1.	**BOT**		
2.	**lockX**(A)		
3.	read(A)		
4.	write(A)		
5.		**BOT**	
6.		**lockS**(A)	T_2 muß warten
7.	**lockX**(B)		
8.	read(B)		
9.	**unlockX**(A)		T_2 wecken
10.		read(A)	
11.		**lockS**(B)	T_2 muß warten
12.	write(B)		
13.	**unlockX**(B)		T_2 wecken
14.		read(B)	
15.	**commit**		
16.		**unlockS**(A)	
17.		**unlockS**(B)	
18.		**commit**	

Tabelle 13.11: Beispiel für 2PL-Protokoll

13.7 Verklemmungen (Deadlocks)

Ein schwerwiegendes Problem bei sperrbasierten Synchronisationsmethoden ist das Auftreten von Verklemmungen (englisch: deadlocks). Tabelle 13.12 zeigt ein Beispiel.

Schritt	T_1	T_2	Bemerkung
1.	**BOT**		
2.	**lockX**(A)		
3.		**BOT**	
4.		**lockS**(B)	
5.		read(B)	
6.	read(A)		
7.	write(A)		
8.	**lockX**(B)		T_1 muß warten auf T_2
9.		**lockS**(A)	T_2 muß warten auf T_1
10.	...	...	$\Rightarrow Deadlock$

Tabelle 13.12: Ein verklemmter Schedule

Eine Methode zur Erkennung von Deadlocks ist die $Time - out-$Strategie. Falls eine Transaktion innerhalb eines Zeitmaßes (z. B. 1 Sekunde) keinerlei Fortschritt erzielt, wird sie zurückgesetzt. Allerdings ist die Wahl des richtigen Zeitmaßes problematisch.

Eine präzise, aber auch teurere - Methode zum Erkennen von Verklemmungen basiert auf dem sogenannten $Wartegraphen$. Seine Knoten entsprechen den Transaktionen. Eine Kante existiert von T_i nach T_j, wenn T_i auf die Freigabe einer Sperre von T_j wartet. Bild 13.4 zeigt ein Beispiel.

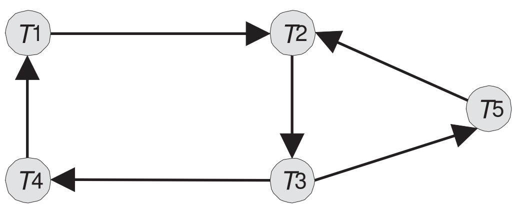

Abbildung 13.4: Wartegraph mit zwei Zyklen

Es gilt der Satz: Die Transaktionen befinden sich in einem Deadlock genau dann, wenn der Wartegraph einen Zyklus aufweist.

Eine Verklemmung wird durch das Zurücksetzen einer Transaktion aufgelöst:

- Minimierung des Rücksetzaufwandes: Wähle jüngste beteiligte Transaktion.
- Maximierung der freigegebenen Resourcen: Wähle Transaktion mit den meisten Sperren.
- Vermeidung von Verhungern (engl. Starvation): Wähle nicht diejenige Transaktion, die schon oft zurückgesetzt wurde.
- Mehrfache Zyklen: Wähle Transaktion, die an mehreren Zyklen beteiligt ist.

13.8 Hierarchische Sperrgranulate

Bisher wurden alle Sperren auf derselben $Granularität$ erworben. Mögliche Sperrgranulate sind:

- Datensatz $\widehat{=}$ Tupel
- Seite $\widehat{=}$ Block im Hintergrundspeicher
- Segment $\widehat{=}$ Zusammenfassung von Seiten
- Datenbasis $\widehat{=}$ gesamter Datenbestand

Abbildung 13.5 zeigt die hierarchische Anordnung der möglichen Sperrgranulate.

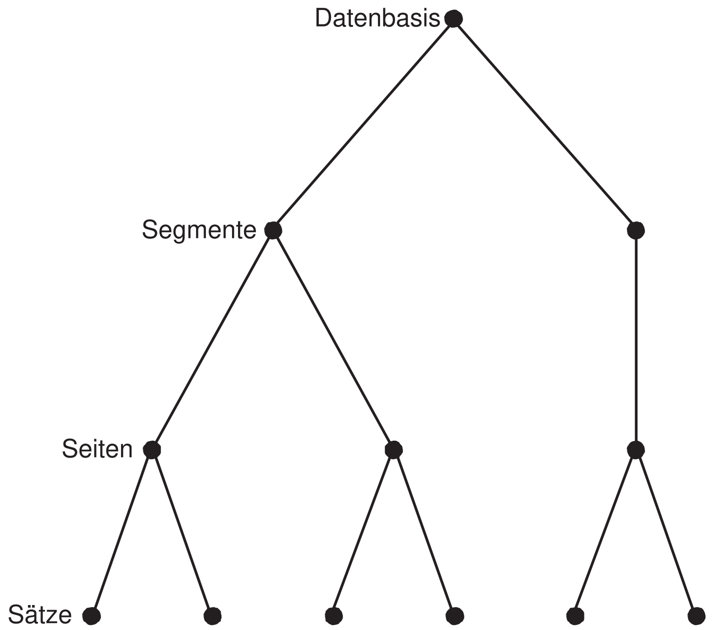

Abbildung 13.5: Hierarchie der Sperrgranulate

Eine Vermischung von Sperrgranulaten hätte folgende Auswirkung. Bei Anforderung einer Sperre für eine Speichereinheit, z.B. ein Segment, müssen alle darunterliegenden Seiten und Sätze auf eventuelle Sperren überprüft werden. Dies bedeutet einen immensen Suchaufwand. Auf der anderen Seite hätte die Beschränkung auf nur eine Sperrgranularität folgende Nachteile:

- Bei zu kleiner Granularität werden Transaktionen mit hohem Datenzugriff stark belastet.
- Bei zu großer Granularität wird der Parallelitätsgrad unnötig eingeschränkt.

Die Lösung des Problems besteht im *multiple granularity locking (MGL)*. Hierbei werden zusätzliche *Intentionssperren* verwendet, welche die Absicht einer weiter unten in der Hierarchie gesetzten Sperre anzeigen. Tabelle 13.13 zeigt die Kompatibilitätsmatrix. Die Sperrmodi sind:

- **NL:** keine Sperrung (no lock);

- **S:** Sperrung durch Leser,
- **X:** Sperrung durch Schreiber,
- **IS:** Lesesperre (S) weiter unten beabsichtigt,
- **IX:** Schreibsperre (X) weiter unten beabsichtigt.

	NL	S	X	IS	IX
S	$\surd$	$\surd$	-	$\surd$	-
X	$\surd$	-	-	-	-
IS	$\surd$	$\surd$	-	$\surd$	$\surd$
IX	$\surd$	-	-	$\surd$	$\surd$

Tabelle 13.13: Kompatibilitätsmatrix beim Multiple-Granularity-Locking

Die Sperrung eines Datenobjekts muß so durchgeführt werden, daß erst geeignete Sperren in allen übergeordneten Knoten in der Hierarchie erworben werden:

1. Bevor ein Knoten mit *S* oder *IS* gesperrt wird, müssen alle Vorgänger vom Sperrer im *IX*- oder *IS*-Modus gehalten werden.
2. Bevor ein Knoten mit *X* oder *IX* gesperrt wird, müssen alle Vorgänger vom Sperrer im *IX*-Modus gehalten werden.
3. Die Sperren werden von unten nach oben freigegeben.

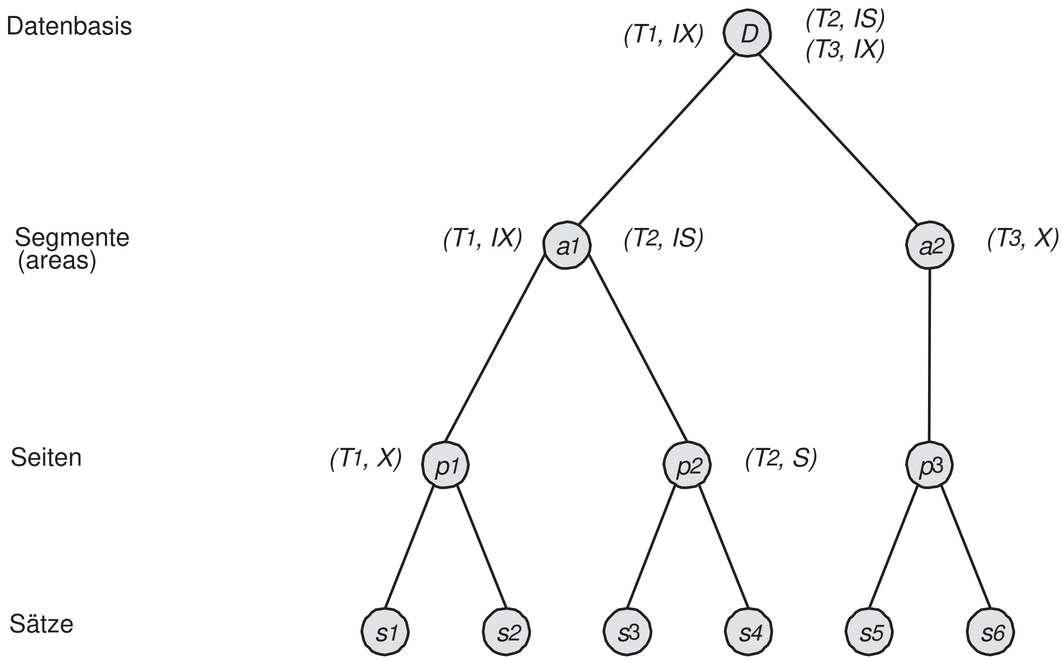

Abbildung 13.6: Datenbasis-Hierarchie mit Sperren

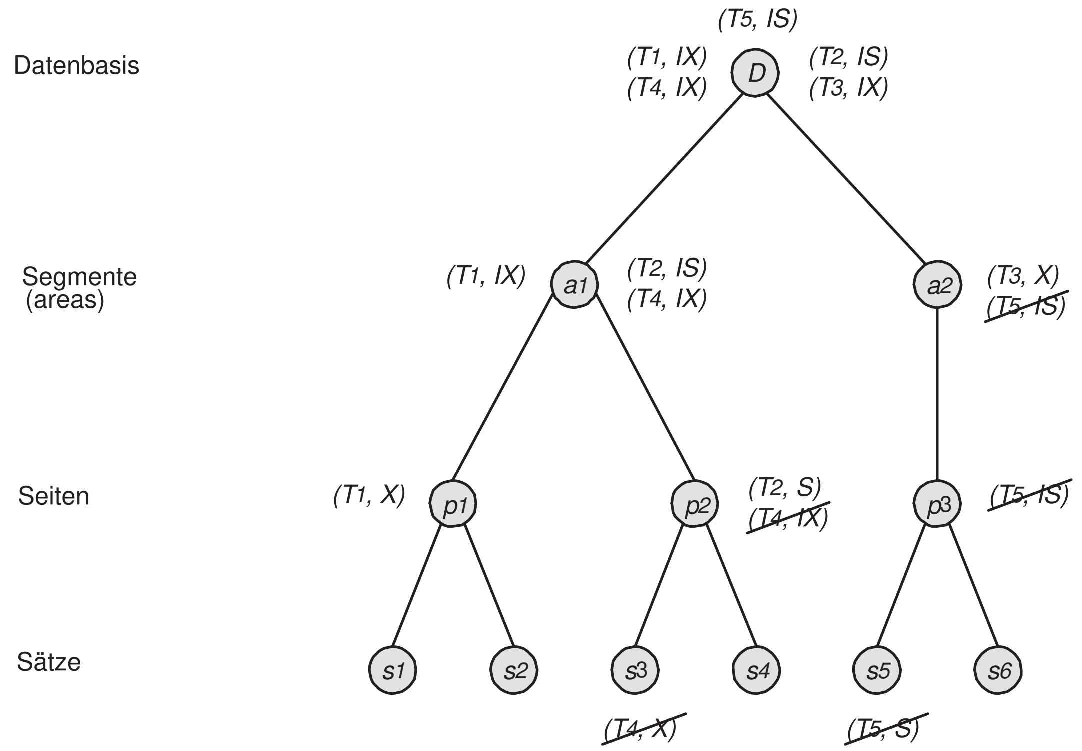

Abbildung 13.7: Datenbasis-Hierarchie mit zwei blockierten Transaktionen

Abbildung 13.6 zeigt eine Datenbasis-Hierarchie, in der drei Transaktionen erfolgreich Sperren erworben haben:

- T_1 will die Seite p_1 zum Schreiben sperren und erwirbt zunächst *IX*-Sperren auf der Datenbasis *D* und auf Segment a_1.
- T_2 will die Seite p_2 zum Lesen sperren und erwirbt zunächst *IS*-Sperren auf der Datenbasis *D* und auf Segment a_1.
- T_3 will das Segment a_2 zum Schreiben sperren und erwirbt zunächst eine *IX*-Sperre auf der Datenbasis *D*.

Nun fordern zwei weitere Transaktionen T_4 (Schreiber) und T_5 (Leser) Sperren an:

- T_4 will Satz s_3 exklusiv sperren. Auf dem Weg dorthin erhält T_4 die erforderlichen *IX*-Sperren für D und a_1, jedoch kann die *IX*-Sperre für p_2 nicht gewährt werden.
- T_5 will Satz s_5 zum Lesen sperren. Auf dem Weg dorthin erhält T_5 die erforderliche *IS*-Sperren nur für D, jedoch können die *IS*-Sperren für a_2 und p_3 zunächst nicht gewährt werden.

Bild 13.7 zeigt die Situation nach dem gerade beschriebenen Zustand. Die noch ausstehenden Sperren sind durch eine Durchstreichung gekennzeichnet. Die Transaktionen T_4 und T_5 sind blockiert, aber nicht verklemmt und müssen auf die Freigabe der Sperren (T_2, S) und $T_3, X)$ warten.

13.9 Zeitstempelverfahren

Jede Transaktion erhält beim Eintritt ins System einen eindeutigen Zeitstempel durch die System-Uhr (bei 1 tic pro Millisecunde $\Rightarrow$ 32 Bits reichen für 49 Tage). Das entstehende Schedule gilt als korrekt, falls seine Wirkung dem seriellen Schedule gemäß Eintrittszeiten entspricht.

Jede Einzelaktion drückt einem Item seinen Zeitstempel auf. D.h. jedes Item hat einen

Lesestempel	$\equiv$	höchster Zeitstempel, verabreicht durch eine Leseoperation
Schreibstempel	$\equiv$	höchster Zeitstempel, verabreicht durch eine Schreiboperation

Die gesetzten Marken sollen Verbotenes verhindern:

1. Transaktion mit Zeitstempel t darf kein Item lesen mit Schreibstempel $t_w > t$ (denn der alte Item-Wert ist weg).

2. Transaktion mit Zeitstempel t darf kein Item schreiben mit Lesestempel $t_r > t$ (denn der neue Wert kommt zu spät).

Bei Eintreten von Fall 1 und 2 muß die Transaktion zurückgesetzt zu werden.

Bei den beiden anderen Fällen brauchen die Transaktionen nicht zurückgesetzt zu werden:

3. Zwei Transaktionen können dasselbe Item zu beliebigen Zeitpunkten lesen.

4. Wenn Transaktion mit Zeitstempel t ein Item beschreiben will mit Schreibstempel $t_w > t$, so wird der Schreibbefehl ignoriert.

Also folgt als Regel für Einzelaktion X mit Zeitstempel t bei Zugriff auf Item mit Lesestempel t_r und Schreibstempel t_w:

```
if (X = read) and (t ≥ t_w)
   führe X aus und setze t_r := max{t_r, t}
if (X = write) and (t ≥ t_r) and (t ≥ t_w) then
  führe X aus und setze t_w := t
if (X = write) and (t_r ≤ t < t_w) then tue nichts
else {(X = read and t < t_w) or (X = write and t < t_r)}
  setze Transaktion zurück
```

Tabelle 13.14 und 13.15 zeigen zwei Beispiele für die Synchronisation von Transaktionen mit dem Zeitstempelverfahren.

	T_1	T_2	
	Stempel `150`	`160`	Item `a` hat $t_r = t_w = 0$
`1.)`	`read(a)`		
	t_r `:= 150`		
`2.)`		`read(a)`	
		t_r `:= 160`	
`3.)`	`a := a - 1`		
`4.)`		`a := a - 1`	
`5.)`		`write(a)`	ok, da $160 \geq t_r = 160$ und $160 \geq t_w = 0$
		$t_w := 160$	
`6.)`	`write(a)`		T_1 wird zurückgesetzt, da
			$150 < t_r = 160$

Tabelle 13.14: Beispiel für Zeitstempelverfahren

In Tabelle 13.14 wird in Schritt 6 die Transaktion T_1 zurückgesetzt, da ihr Zeitstempel kleiner ist als der Lesestempel des zu überschreibenden Items *a* ($150 < t_r = 160$). In Tabelle 13.15 wird in Schritt 6 die Transaktion T_2 zurückgesetzt, da ihr Zeitstempel kleiner ist als der Lesestempel von Item *c* ($150 < t_r(c) = 175$). In Schritt 7 wird der Schreibbefehl von Transaktion T_3 ignoriert, da der Zeitstempel von T_3 kleiner ist als der Schreibstempel des zu beschreibenden Items *a* ($175 < t_w(a) = 200$).

	T_1	T_2	T_3	a	b	c
	`200`	`150`	`175`	t_r `= 0`	t_r `= 0`	t_r `= 0`
				t_w `= 0`	t_w `= 0`	t_w `= 0`
`1.)`	`read(b)`				t_r `= 200`	
`2.)`		`read(a)`		t_r `= 150`		
`3.)`			`read(c)`			t_r `= 175`
`4.)`	`write(b)`				t_w `= 200`	
`5.)`	`write(a)`			t_w `= 200`		
`6.)`		`write(c)`				
		Abbruch				
`7.)`			`write(a)`			
			ignoriert			

Tabelle 13.15: Beispiel für Zeitstempelverfahren