α DATENBANKSYSTEME II

β Ergänzende Literatur zur Vorlesung

[30.04.03]

- G. Gardarin, P. Valduriez — **Relational Databases and Knowledge Bases.** Addison-Wesley 1989.
- G. Weikum — **Transaktionen in Datenbanksystemen.** Addison-Wesley, 1988.
- J.D. Ullman — **Principles of Database and Knowledge-Base Systems, Vol. I.** Computer Science Press, 1989.
- P. Lockemann, J. Schmidt — **Datenbank-Handbuch.** Springer 1993.
- C.J. Date — **An Introduction to Database Systems.** 7. Auflage, Addison-Wesley, 2000.
- G. Schlageter, W. Stucky — **Datenbanksysteme: Konzepte und Modelle.** Teubner, 1983.
- G. Vossen, M. Groß-Hardt — **Grundlagen der Transaktionsverarbeitung.** Addison- Wesley, 1993.
- R. Elmasri, S. Navathe — **Fundamentals of Database Systems.** Addison-Wesley, 2000.
- J. Gray, A. Reuter — **Transaction Processing. Concepts and Techniques.** Morgan Kaufmann, 1993.

Seite 1

- Bernstein, P.A., Newcomer, E. — **Principles of Transaction Processing for the Systems Professional.** Morgan Kaufmann, 1997 ["Transaktion" Kernbegriff der Vorlesung, als eine Lösung für "Nebenläufigkeit"]
- Bernstein, P.A, Hadzilacos, V., Goodman, N. — **Concurrency Control and Recovery in Database Systems** Addison-Wesley, 1987 http://research.microsoft.com/pubs/ccontrol ["Concurrency control" / Kontrolle der Nebenläufigkeit; Außerdem "Recovery" / Erholung, nachdem die Datenbank "krank" geworden ist (Normalerweise kann man hier nur zurücksetzen; Probleme können immer auftreten, z.B. wenn der Benutzer eine Transaktion abbricht); Im Internet downloadbar]
- Cellary, W., Gelenbe, E., Morzy, T. — **Concurrency Control in Distributed Database Systems.** North Holland, 1989 [CC bei verteilten Datenbanksystemen, noch komplexer, verschiedene nicht-synchropne Uhren, in Vorlesung nicht viel davon]
- Date, C.J — **An introduction to Database Systems, 7. Aufl.** Addison-Wesley, 2000 [In guten DB-Büchern gibt es auch Kapitel über Transkation]
- Elmasri, R., Navathe, S. — **Fundamentals of Database Systems** Benjamin/Cummings, 2000 [Dito]
- Garcia-Molina, H., Ullmann, J., Widom, J. — **Database System Implementation** Prentice Hall, 2000 [Implementation]
- Gardarin, Georges, Valduriez, Patrick — **Relational Databases and Knowledge Bases,** Addison Wesley, 1989
- Gray, J., Reuter, A. — **Transaction Processing. Concepts and Techniques.** Morgan Kaufmann, 1993 ["Bibel", Konzepte und Techniken, 1500 Seiten]
- Härder, T., Rahm, E. — **Datenbanksysteme: Konzepte und Techniken der Implementierung.** Springer,1999 ["Sehr empfehlenswertes Buch", auch Konzepte und Techniken; Man braucht Systeme, die beweisbar mit allen Fällen zurechtkommen (Theorie)]
- Jajodia, S., Kerschberg, L. — **Advanced Transaction Models and Architectures.** Kluwer Academic Publishers, 1997 [Geschachtelte Transaktion, hier nicht (Z.B. Flug buchen ⇒ Hotel buchen ⇒ Theater-Tickets, alles Rücksetzen)]
- Kemper, A., Eickler, A. — **Datenbanksysteme. Eine Einführung,** Oldenbourg, 1999
- Kumar, V. — **Recovery in Database Management Systems**

Hsu, M. — Prentice Hall, 1998

- Lewis, M.L. / Bernstein, A. / Kifer, M. — **Databases and Transaction Processing** **An Application-Oriented Approach** Addison Wesley, 2002 [Bernstein ist "Papst" auf diesem Gebiet]
- Lynch, N. / Merritt, M. / Weihl, W. / Fekete, A. — **Atomic Transactions** Morgan Kaufmann, 1999 [Transaktionen sind atomar bzw. müssen atomar behandelt werden; Allerdings ist es komplex, wie man etwas, was nicht atomar ist, so behandelt]
- Schlageter, G. / Stucky, W.: — **Datenbanksysteme: Konzepte und Modelle,** Teubner, 1983 [Weniger dick, 1983, aber vieles steht bereits in diesem Buch]

Seite 2

- Thomasian, A. — **Database concurrency control: methods, performance, analysis.** Kluwer Academic Publishers, 1996 [Ohne Performance ist alles trivial, dann kann man die DB sperren, bis die Transaktion fertig ist]
- Vossen, G. / Gross-Hardt, M. — **Grundlagen der Transaktionsverarbeitung,** Addison-Wesley, 1993 [Bücher von Vossen und Weikum relativ modern, empfehlswert]
- Weikum, G. — **Transaktionen in Datenbanksystemen,** Addison Wesley, 1988
- Weikum, G. / Vossen, G. — **Transactional Information Systems** **Theory, Algorithms and the Practice of Concurrency Control and Recovery** Morgan Kaufmann, 2002
- McGee, W.C. — **The Information Management Systems IMS/VS** **Part I: General structure and operation** **Part II: Database Facilities** IBM System Journal 16, S. 84-95, 96-122, 1977 [Die Vorlesung gliedert sich in 2 Teile: (1) Synchronisation (Concurrency Control), (2) Pre-Relationale Systeme (knapp, letzte 2 Wochen); Da man heute noch damit beschäftigt ist, alte Datenbanksysteme umzustellen, muß man sich auch mit den anderen Modellen (Hierarchisches Modell, Netzwerk-Modell) beschäftigen]
- Conference on Data System Languages (CODASYL)/ Data Base Task Group (DBTG) — **Report of CODASYL Data Description Language Committee** Information Systems 3, 247-320,1978
- Meier, A. / Dippold, R. — **Migration und Koexistenz heterogener Datenbanken** **Praktische Lösungen zum Wechsel in die relationale Datenbanktechnologie.** Informatik Spektrum 15, S. 157-166, 1992 [Migration]
- Brodie, M. / Stonebraker, M. — **Migrating Legacy Systems: Gateways, Interfaces, and the Incremental Approach,** Morgan Kaufmann, 1995 ["Legacy" hier Altlast, man muß langsam / inkrementell umstellen]
- Fong, J. / Huang, S. — **Information Systems Reengineering** Springer - Singapore, 1997 [Reengineering, z.B. von Cobol in neuer Sprache]
- Kim, W. (Ed.) — **Modern Database Systems** ACM Press and Addison Wesley 1995 Insbesondere Teil 2: Technology for Interoperating Legacy Databases ["Modern" und "Legacy" zusammenbringen]
- Ullman, J.D. — **Principles of Database Systems** Computer Science Press 1981 [Ullman hervorragend]
- Ullman, J.D. — **Principles of Database and Knowledge Base Systems** Vol. 1,2, Computer Science Press, 1988, 1989
- Lang, S.M. / Lockemann, P.C. — **Datenbankeinsatz,** Springer 1995 [Praktische Fragen]
- Dürr, M. / Radermacher, K. — **Einsatz von Datenbanksystemen,** Springer 1990

Seite 3

- Lockemann, P.C. / Schmidt, J. — **Datenbankhandbuch,** Springer unveränderte 2. Aufl. 1993
- Paton, N.W. (ed.) — **Active Rules in Database Systems** Springer 1999

[DB wird hier selber aktiv (z.B. wenn ein Gehalt 3 Jahre nicht mehr erhöht wurde), hier nicht]

- Ozsu, M.T. Valduriez, P. **Principles of Distributed Database Systems**
  Prentice Hall, 1999
  [Verteilte Datenbanksysteme, hier nicht]
- Dadam, P. **Verteilte Datenbanken und Client/Server Systeme. Grundlagen, Konzepte und Realisierungsformen**
  Springer 1996
  [Verteilte Datenbanksysteme, Client/Server]
- Chamberlin, D. **A Complete Guide to DB2 Universal Database**
  Morgan Kaufmann, 1998
  [Zum Praktikum]
- Janacek, C. Snow, D. **DB 2 Universal Database Certification Guide**
  Prentice Hall, 1997
  [Dito]

Übersicht

(1) **Synchronisation (Concurrency Control)**
- (1.1) **Probleme beim Mehrbenutzerbetrieb (Diagnose) und Lösungsansätze (Therapie)**
  [Mehrbenutzerbetrieb≙Concurrency; Diese Ideen sind Quick&Dirty, aber man benötigt allgemeine Ansätze, die man beweisen kann]
- (1.2) **Grundlegende Definitionen für Synchronisation, insbesondere Transaktionskonzept und "Serialisierbarkeit"**
  [Was ist eigentlich "gesund"?; Die Prozesse müssen sich gegenseitig beeinflussen können, das läßt sich nicht verhindern; Wenn 2 Personen die letzte Karte in einem Flugzeug möchten, daß man dafür gesorgt werden, daß nur 1 sie bekommt (nicht keiner, nicht beide) ⇒ "Serialisierbarkeit" (der Grundbegriff der Vorlesung); DB kennt Semantik des Problems nicht, d.h. man braucht "ruppige" Verfahren]
- (1.3) **Transaktionssteuerung**
  [Wenn man weiß, was "gesund" ist, dann muß man sich überlegen, mit welchen Methoden (Algorithmen) man dafür sorgt]
- (1.4) **Zwei-Phasen-Sperrprotokoll**
  [Erste Methode]
- (1.5) **Deadlock-Behandlung, insbesondere Vermeidung**
  [Deadlock: Z.B. 2 Prozesse benötigen die Ressourcen A und B benötigen; Der erste Prozeß belegt A, der zweite B, nun warten sie endlos aufeinander, bis der jeweils andere seine Resource freigibt; Behandlung und insbesondere Vermeidung]
- (1.6) **Zeitstempel-Verfahren**
  [Anträge werden nach der eingehenden Reihenfolge behandelt, ineffizient]
  - (1.6.0) **Einleitung, Serialisierung bei Transaktionsbeginn, bei Konflikt, bei Transaktionsende**
    [Bei Beginn ≙ Zeitstempel; Bei Konflikt ≙ Man vergibt nur eine Reihenfolge, falls die Prozesse in Konflikt geraten; Beim Ende ≙ Optimistisch, durchlaufen lassen, bei Problem zurücksetzen, z.B. Ethernet); Je nachdem, was der Normalfall ist, lohnt sich verschiedenes]
  - (1.6.1) **Totale Anordnung der Zeitstempel**
  - (1.6.2) **Partielle Anordnung der Zeitstempel**
    [Besser, nur von denen, die in Konflikt stehen]
  - (1.6.3) **Mehrfach-Versionen und Zeitstempel**
    [Ressource verdoppeln, aber beim Zurückschreiben muß man dann aufpassen]

α ABLAUFSTEUERUNG BEI NEBENLÄUFIGKEIT (1)

β Probleme bei Nebenläufigkeit (1.1)

"Unerwünschte" Interaktion, wenn mehrere Benutzer diesselben Daten benötigen.

Beispiele

(1) Lost Update (Verlorene Operation)

Durch unglückliche Verzahnung kann die Wirkung eines Prozesses unbemerkt [!] verloren gehen. P_1, P_2 verringern den Wert einer Datenbankvariable X (X Datenbankobjekt, x_i Hauptspeichervariable (i=1,2)) [Z.B. X Anzahl vorhandener Bücher von Knuth] ...

P_1 (Buche eine Einheit von Konto X ab)	P_2 (Buche eine Einheit von Konto X ab)
read X→x_1	
[Leitung hat Schluckauf]	**read X→x_2**
	x_2-1→x_2 [Falls negativ Vormerkung]
	write x_2→X
x_1-1→x_1	
write x_1→X	
[Falls zu Beginn X=2, jetzt X=1]	

D.h. die Wirkung von P_2 ist verloren gegangen [Egal was P_2 macht, Ergebnis ist danach weg]. DB konsistent [Die Datenbank sieht nur die einzelnen Anweisungen, "Buche eine Einheit von Konto X ab" existiert nur auf der Metaebene, X=1 völlig legaler Wert]

In SQL ...

```
select KONTOSTAND into :kontostand
from KONTEN
where KONTONR = 4711;
:kontostand = :kontostand - 1;
update KONTEN
set KONTOSTAND = :kontostand
where KONTONR = 4711;
```

[Das erstes select-Statement entspricht dem Read und das zweite dem Write; ":" bezeichnet eine Hauptspeichervariable (http://www.cs.man.ac.uk/~horrocks/Teaching/cs2312/Lectures/Handouts/dbpl.pdf)]

(2) Currency Confusion (Verwirrung über aktuellen Zustand)

Beispiel (X Anzahl der freien Plätze auf dem Flug) ...

P_1 (Flug-Reservierung)	P_2 (Streiche den Flug)
read X→x_1	
if x_1>0 then x_1-1→x_1	
	delete X (Beim Löschen ist unbekannt, daß der Platz gerade vergeben wurde)
≪ print Bestätigung ≫	
write x_1→X	

Scheinbare Lösung ...

write x_1→X
≪ print Bestätigung ≫

[In diesem Fall wird erst versucht, in die Variable X zu schreiben und dann die Bestätigung ausgedruckt; Da die Variable X gelöscht ist, schlägt dies natürlich fehl; D.h. diese Lösung ist besser programmiert]

Aber ...

write x_1→X	
	delete X
≪ print Bestätigung ≫	

[Aber auch das hilft in diesem Fall nicht; Außerdem möchte man auch nicht von einer guten Programmierung abhängig sein]

Weiterer Lösungsversuch ...

write x_1→X	
	delete X
	≪ print Stornierung ≫
≪ print Bestätigung ≫	

[Angenomen P_1 bucht in Deutschland bei der Lufthansa und P_2 löscht den Flug in Hongkong wegen SARS; Nun kann man zumindest erwarten, daß P_2 der Lufthansa bescheid gibt und eine Stornierung an alle Fluggäste verteilt; Aber auch das hilft in obigen Fall nicht, da erst die Stornierung herausgeschickt wird und dann die Bestätigung; D.h. das System macht dein Eindruck, daß es nicht weiß, was es tut]

[Generell ist das Problem zwar lösbar, aber nur mit Methoden, die man nicht haben will (z.B. muß man auch mit "doofen" Programmen zurechtkommen); D.h. während X in einem "fragilen" Zustand ist, muß es gesperrt werden; Dies ist in beiden Beispielen zwischen dem Lesen und Schreiben von X der Fall; In (2) muß dann während dieser Zeit das Löschen abgewehrt werden, d.h. ersten buchen und dann löschen; Das ist zwar nicht sehr sinnvoll, allerdings gibt es kein Gesamtsystem, das dies erkennen könnte; Alternativ kann man

zwar den jeweiligen Anforderungen Prioritäten zuweisen (z.B. **`delete`** kann eine Buchung unterbrechen), aber da man bei einer Datenbank schlecht solche Prioritäten abschätzen kann, werden alle gleich behandelt]

(3) **Dirty Reads (Lesen von unsauberen Datenbankzuständen)** [07.05.03]

[(http://www.iaws.sowi.uni-bamberg.de/forschung/projekte/bwl_interaktiv/lexikon/23.HTM) **Bilanz** "Die Bilanz wird am Ende eines jeden Geschäftsjahres im Rahmen des Jahresabschlusses erstellt. In der Bilanz werden Vermögenspositionen (=Aktiva) sowie Eigenkapital und Schuldpositionen (=Passiva) des Unternehmens einander gegenübergestellt."; (http://www.iaws.sowi.uni-bamberg.de/forschung/projekte/bwl_interaktiv/lexikon/27.HTM) **Bilanzverlängerung** "Eine Bilanzveränderung, bei der durch einen Geschäftsvorfall eine oder mehrere Positionen sowohl auf der Aktivseite als auch auf der Passivseite der Bilanz zunehmen, heißt Bilanzverlängerung. Es wird bei einer Bilanzverlängerung auch von Aktiv-/Passivmehrung gesprochen. Eine Bilanzverlängerung erhöht die Bilanzsumme. Diese Erhöhung kann erfolgsneutral oder erfolgswirksam vonstatten gehen."]

A, B seien Aktiva bzw. Passiva eines Unternehmens A=B ...

- P_1 erhöht A und B (Bilanzverlängerung) [Z.B. nimmt die Firma einen Kredit auf]
- P_2 gibt A und B aus

P_1 (Inkrementiere A, B)	P_2 (Beobachte A,B)	{ A = B }
`read A→a1`		
`a1+1→a1`		
`write a1→A`		
	`print Balt`	{ A = B+1 }
	`print Aneu`	inkonsistenter schmutziger Zwischenzustand
`read B→b1`		
`b1+1→b1`		
`write b1→B`		
konsistenzerhaltend [Zusammen]	*konsistenzerhaltend* [Nur lesend]	{ A = B }

[Die Datenbank hat den Index $_{alt}$ bzw. $_{neu}$ nicht zur Verfügung; Falls ein Wirtschaftsprüfer an der obigen Stelle lesen würde, hätte er eine Unterschlagung gefunden; Einzeln sind die beiden Operationen konsistenzerhaltend, aber nicht mehr bei Nebenläufigkeit]

Daten können "dirty" sein, weil ...

(a) Noch nicht fertig, z.B. approximativer Zwischenwert [Z.B. der Nullstellenalgorithmus von Newton, erst die 5te Iteration ist brauchbar] oder Ansatz, der erst noch getestet werden muß [Z.B. Vermutung eines Heuristik-Programmes].

(b) Zwar fertig, aber noch nicht endgültig, weil evtl. später noch zurückgesetzt [Z.B. werden mit großen Aufwand alle Abbuchungen von einem Konto berechnet, aber wenn das Konto am Ende nicht gedeckt ist, wird alles verworfen; Oder z.B. allen Mitarbeitern soll eine Prämie ausgezahlt werden, aber diese soll weniger als 2% vom Gehalt sein (am Ende der Prämienberechnung abgebrochen falls nicht <2%)].

(c) Zwar fertig und endgültig, aber nicht zu anderen Daten passend. Dann ist der Datenbankzustand dirty und nicht die einzelnen Daten [Z.B. passen oben B_{alt} und A_{neu} oben nicht zueinander].

Therapie

[Es gibt verschiedene Therapien für die Probleme]

(0) **Sperren von A und B, jeweils solange an ihnen gearbeitet wird**

P_1	P_2	{ A = B }
`lock A`		
`update A`		
`unlock A`		
	`lock B`	
	`lock A`	
	`print B`	{ A = B+1 }
	`print A`	
	`unlock A`	
	`unlock B`	
`lock B`		
`update B`		
`unlock B`		{ A = B }

[**`lock A`** besteht genauer gesagt aus **`request lock A`** und **`wait for lock A`**; Das Ergebnis zeigt aber, daß es nach der Therapie noch schlimmer ist als vorher: Man muß sperren und dirty reads werden dennoch nicht verhindert]

Diese Sperre verhindert lost update und concurrency confusion, aber nicht dirty reads. Therapie 0 heilt nicht.

($\frac{1}{2}$) **Sperre alles "zusammengehörende" und halte Sperren bis zum Schluß**

[Die Datenbank weiß nicht, daß nach A noch B kommt, daher sperre alles von Anfang an]

Im Extremfall: Sperre die ganze Datenbank ⇒ zu viel und zu lange.

(1) **Zwei-Phasen-Sperrprotokoll (Kompromiß)**

Nach der ersten Sperrenrückgabe können keine weiteren Sperren mehr angefordert werden (≙ Es gibt 2 getrennte Phasen [Erwerbs- und Freigabephase]) ...

```
P1                                                          P2
lock A (checkout)   ┬ Hier ist Lesen     ┬ Lesen von Balt   lock B
update A            │ von A durch P2     ┴ durch P2 möglich. print B
lock B              │ unmöglich, weil    ┬ Hier ist Lesen   unlock B
unlock A (check in) ┴ sperren exklusiv.  │ von B durch P2   lock A
update B            ┬ Lesen von Aneu     │ unmöglich, weil  print A
unlock B            ┴ durch P2 möglich.  ┴ sperren exklusiv. unlock A
```

P_2 genügt nicht 2-Phasen-Sperrprotokoll! [Deshalb ist das Ergebnis auch wieder falsch]
Aber mit 2-Phasen-Sperrprotokoll ...

```
lock A      lock B
  ⋮           ⋮
lock B      lock A
```

Deadlock! ["Nebenwirkung"]

Behauptung
Falls kein Deadlock auftritt, gilt: Wenn P_1 und P_2 das 2-Phasen-Sperrprotokoll einhalten, muß P_1 logisch ganz vor P_2 kommen oder umgekehrt, d.h. P_2 sieht nur neue Werte von P_1 oder nur alte, aber nichts gemischtes.
Beweis: Später.

Ergebnis
So fair gegenüber P_1' wie möglich, unter der Prämisse daß P_2' anfängt [mit 2 Phasen-Sperrprotokoll] ...

```
P1'             P2'
                lock A
                print A
                lock B
                unlock A
lock A
update A
                print B
                unlock B
lock B
update B
unlock B
```

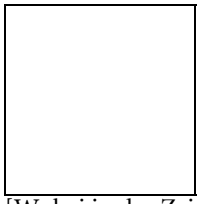
P_2' logisch vor P_1'? Ja.
[Die Anweisung **`update A`** benötigt evtl. mehr Zeit, wenn dafür z.B. erst auf die Platte zugegriffen werden muß; Aber in diesem Fall kann dann bereits P_2 weitermachen, dann P_1 und P_2 streiten sich nur um den Prozessor; Außerdem haben dadurch die Benutzer von P_1 und P_2 das Gefühl, "daß sich etwas tut"; **`update A`** setzt nur den Befehl ab; D.h. es ist falsch, daß es einfach nur 11 Befehle sind, deren Verschränkung zeitlich egal ist; Auch **`update A`** kann während **`print B`** arbeiten]

Diagramme
2-Phasen-Sperrprotokoll ...

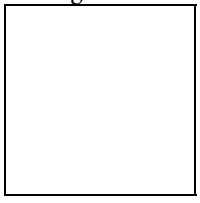
PASSAGIER-LISTE

NAME	FLUGNO
-	-

[Es gibt nur 1 Gipfel]
Lösung $\frac{1}{2}$ ...



[Wobei in der Zeichnung angenommen wird, man kann über eine Liste alle benötigten Variablen auf einmal sperren; Auch wenn 1000 Objekte gesperrt werden müssen, ist dies noch einfach, denn die Datenbank kennt die Semantik von Sperren]
Lösung 0 ...



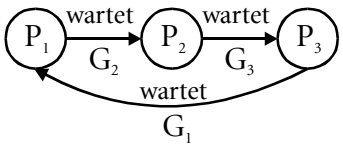
["Dirty" in Zwischenzuständen; D.h. man möchte eine möglichst kleine Fläche unter der gezeigten Kurve, damit so wenig wie möglich gesperrt wird, aber dennoch soll das Ergebnis richtig sein; Auf der anderen Seite nimmt die Deadlock-Wahrscheinlichkeit mit der Größe der Fläche zu (Wobei 1/2 hier eine Aussnahme ist, falls alle Sperren auf einmal akquiriert werden, denn dann können keine Deadlocks auftreten; Aber bei schrittweiser Akquirierung der Sperren gilt dies auch für 1/2 und dies möchte man im Normalfall)]

(4) Verklemmung: Dining philosophers

[Fortsetzung der Problem-Beispiele nach "(3) Dirty Reads"]

[ ; 3 Philosophen essen Spaghetti, wobei jeder hier 2 Gabeln braucht, obwohl nur 3 Gabeln insgesamt vorhanden sind]

Jeder Philosoph greift nach der Gabel zu seiner Rechten ⇒ Verklemmung ...



Präzedenzgraph ≠ Wartegraph

[Später kommt ein Satz, der zeigt, daß ein Problem nicht serialisierbar ist, falls es zu einem Zyklus im Präzedenzgraphen kommt; Präzedenz bedeutet "P_i muß fertig sein, bevor P_j anfängt" bzw. "P_i präzediert P_j".; Im allgemeinen ist Präzedenzgraph ≠ Wartegraph; Oben ist der Wartegraph abgebildet]

[- **(ElmasriNavathe696) Präzedenzgraph**

Prüfen eines Ausführungsplans S auf Konfliktserialisierbarkeit ...

(1) Erzeuge für jede Transkation T_i, die in Ausführungsplan S auftritt, einen Knoten T_i im Präzedenzgraphen.

(2) Erzeuge für jeden Fall in S, bei dem T_j ein **`read_item(X)`** ausführt, nachdem T_i ein **`write_item(X)`** ausgeführt hat, eine Kante ($T_i \rightarrow T_j$) im Präzedenzgraph.

(3) Erzeuge für jeden Fall in S, bei dem T_j ein **`write_item(X)`** ausführt, nachdem T_i ein **`read_item(X)`** ausgeführt hat, eine Kante ($T_i \rightarrow T_j$) im Präzedenzgraph.

(4) Erzeuge für jeden Fall in S, bei dem T_j ein **`write_item(X)`** ausführt, nachdem T_i ein **`write_item(X)`** ausgeführt hat, eine Kante ($T_i \rightarrow T_j$) im Präzedenzgraph.

(5) Der Ausführungsplan S ist serialisierbar, falls der Präzedenzgraph keinen Zyklus aufweist.

- **(ElmasriNavathe720) Wartegraph**

Eine einfache Möglichkeit für das System, einen Verklemmungszsutand zu erkennen, ist die Bildung und Führung eines Wartegraphen (Wait-for Graph). Dabei wird für jede Transkation, die momentan ausgeführt wird, ein Knoten erzeugt. Wenn eine Transaktion T_i wartet, um ein Objekt X zu sperren, das momentan von einer Transaktion T_j gesperrt ist, wird eine gerichtete Kante ($T_i \rightarrow T_j$) im Graphen eingeführt. Wenn T_j die Sperre(n) auf die Objekte, auf die T_i wartet, freigibt, wird die gerichtete Kante wieder entfernt. Ein Verklemmungsszustand entsteht nur, wenn der Wartegraph einen Zyklus aufweist.]

Gegenmaßnahmen

(1) Zentrale Steuerung, Priorisierung [Will man aber vermeiden]

(2) ALOHA-Protokoll, Ethernet, Zufallsmechanismus [Jeder versucht die Gabeln direkt zu greifen; Falls sie belegt sind, dann werden sie wieder hingelegt und eine zufällige Zeit gewartet]

(3) Time-Out [Z.B. nach 1 Stunde wieder freigeben; Aber ein Time-Out benötigt auch eine zentrale Steuerung (Diese sorgt dafür, daß die Resource auch wieder freigegeben wird)].

(4) Explizite Kontrolle des Wartegraphen [Z.B. wird die Verbindung G_1 oben im Graphen nicht eingetragen, falls dadurch ein Zyklus entsteht; D.h. P_3 darf nicht weiter warten, sondern "fliegt raus"; Ohne Semantik, rein durch Protokolle (Zwar zentrale Steuerung, aber auf ganz einfachem Niveau)].

[14.05.03]

Programmierung für Einbenutzer- und Mehrbenutzerbetrieb

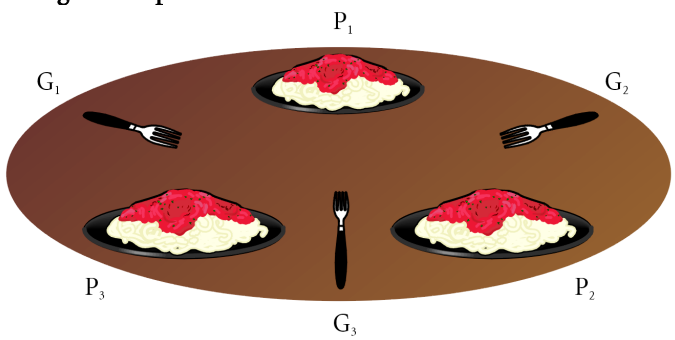
Programmierung für Einbenutzerbetrieb		Programmierung für Mehrbenutzerbetrieb
(1) Partielle Korrektheit [Falls Ende dann Ok]	≈	Konsistenz (einzeln) [Einzelnes Programm korrekt] + Freiheit von unzulässigen Wechselwirkungen (zusammen)
(2) + Termination ⇒ [Totale] Korrektheit	≈	Verklemmungsfreiheit (Scheinbar paradox, denn ... (a) Bei Deadlock bleibt normalerweise nicht alles stehen,

weil nicht alles ineinander verkeilt [Bei Termination dagegen ist das Programm komplett beendet]
(b) Busy Waiting [Statt Ende eigentlich Endlosschleife])

Nichttermination ≈ Verklemmung (unendliche Schleife)

(3) Starvationfreiheit (Fairness) [Kein Prozeß hungert den anderen aus; Ein Problem hierbei ist die "Rücksetzung" (D.h. der Prozeß kommt zwar dran, aber nur halb und wird immer wieder zurückgesetzt), genauer im folgenden]

Dining Philosophers



Verschwörung von P_1 und P_2 gegen P_3 ...
- P_1 hält permanent Gabel G_1
- P_2 hält permanent Gabel G_3
- P_1 und P_2 halten abwechseln Gabel G_2

[Es gibt keinen Zyklus im Wartegraphen, d.h. keine Verklemmung, aber trotzdem ist dieser Zustand nicht gut, da P_3 ausgehungert wird]
Damit ...

P_1	P_2	P_3
lock G_1		
	lock G_3	
		wait for lock G_1 oder G_3
lock G_2		
eat		
unlock G_2		
	lock G_2	
	eat	Starvation
	unlock G_2	
...	...	...

2-Ph asen-Sperrprotokoll hilft!
[Das 2-Phasen-Sperrprotokoll verhindert, daß es endlos so weitergeht; D.h. positiv ist, daß es semantische Inkonsistenzen verhindert (alte und neue Werte getrennt), dafür sorgt es aber für Deadlocks, verhindert aber auf der anderen Seite auch Aushungern]

Fazit

Geeignete Sperr-Protokolle können (hoffentlich) die bei Nebenläufigkeit möglichen Inkonsistenzen verhindern. Ohne zusätzliche Synchronisationsmaßnahmen können Verklemmungen auftreten (sie werden sogar wahrscheinlicher).
[Man kann auch auf die Synchronisation verzichten und während der Ausführung davon ausgehen, daß keine Deadlocks und Verklemmungen aufgetreten sind; Ganz am Ende, bevor man auf die Platte durchschreibt, wird geprüft, ob es bei der Transaktion Probleme gab (jede Transaktion hat ein "read set" und ein "write set"); Im Problemfall wirft man beide Transaktionen aus der Datenbank (und diese werden vom Client dann danach erneut gestartet); "Optimistisches Protokoll", bei geringem Durchsatz ist das die richtige Wahl, wenn es nur bei wenigen Fällen schiefgeht; Wenn aber bereits an der Durchsatzgrenze gearbeitet wird, dann steigt generell die Wahrscheinlichkeit von Problemen durch die vielen Anfragen und das erneute Ausführen verstopft zusätzlich die Leitung]
[Zum Aushungern: Wenn man nur endlich viele Prozesse hat, die in endlicher Zeit abgearbeitet werden, dann kann es generell nicht zu Starvation kommen; Allerdings kann in Realität der Fall auftauchen, daß bei der Lufthansa in jeder Sekunde 1000 neue Anfragen ankommen; D.h. ein alter Prozeß wird immer wieder überholt; Dies muß man über einen Zeitstempel abfangen, daß sich die älteren Prozesse irgendwann durchsetzen können]

Beispiel 4: Permanente Konsistenzverletzung

Es einsteht ein über den Abschluß eines Prozesses andauernder Schaden. Prozesse ...
- P_1 erhöht zwei Datenbankvariablen um 1.
- P_2 verdoppelt zwei Datenbankvariablen.

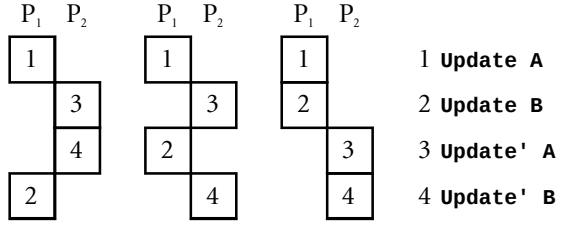
[Generell gilt $(A+1)\cdot 2 \neq A\cdot 2+1$; Was zuerst stattfindet, entscheidet die Datenbank generell nicht; Allerdings muß sie dafür sorgen, daß entweder das eine oder das andere auf beide Variablen angewendet wird]
Damit ...

P_1 (Inkrementiere A und B)	P_2 (Verdopple A und B)	{ A = B }
`lock A`		
`read A→a1`		
`a1+1→a1`		
`write a1→A`		
`unlock A`		{ $A_{neu,1} = B_{alt}+1$ }
	`lock A`	
	`read A→a2`	
	`a2*2→a2`	
	`write a2→A`	{ $A_{neu,1,2} = 2B_{alt}+2$ }
	`lock B`	
	`unlock A`	$A_{neu,1,2}$ sichtbar für Rest der Welt
	`read B→b2`	
	`b2*2→b2`	
	`write b2→B`	
	`unlock B`	{ $A_{neu,1,2} = B_{neu,2}+2$ }
`lock B`		
`read B→b1`		
`b1+1→b1`		
`write b1→B`		
`unlock B`		{ $A_{neu,1,2} = B_{neu,2,1}+1$ }

$A \neq B$, Inkonsistenz ↯.
[Selbst wenn man links sperrt, kommt es links zu einem Fehler (neuer Wert von A mit altem Wert von B gemischt); Links wurde das 2-Phasen-Sperrprotokoll nicht eingehalten, es müßte **`unlock A`** und **`lock B`** vertauscht werden]
Besser ...

P_1	P_2	
`lock A`		
`update A`		
`lock B`		
`unlock A`		
	`lock A`	- Bis hierin muß P_2 warten, falls P_1 als erstes die Sperre
	`update' A`	auf A bekommen hat und P_2 in seinem Code zunächst
	`request lock B`	die Sperre auf A beantragt [2 Voraussetzungen].
`update B`		- P_2 muß wieder warten.
`unlock B`		
	`lock B`	
	`unlock A`	
	`update' B`	
	`unlock B`	{ $A_{neu}=2A_{alt}+2$, $B_{neu}=2B_{alt}+2$ }

[Selbst wenn P_2 immer so früh wie möglich weitermachen kann, muß es zweimal warten (falls P_1 zuvor startet)]
Reihenfolge ...



[**Rechts** Sequentiell, aber ungerecht; **Links** Gerecht, P_2 muß nur einmal warten, aber falsch; **Mitte** Etwas ungerecht, aber geht nicht besser (Wobei natürlich P_2 zu Beginn genauso vor P_1 kommen könnte; Da man nur einen Prozessorkern hat, muß einer gewinnen); Die Zwischenzustände sind hier durchaus sichtbar, aber entweder nur alte oder nur neue Werte]
Oder ...

P_1	P_2
`lock A`	`lock B`

Deadlockgefahr!
[Fordert P_2 erst B an, so können zwar beide weitermachen, dafür kann es aber zu Deadlocks kommen, also hilft dies auch nicht wirklich; Außerdem gibt es keinen "Guru", der P_2 davon verständigen, daß es seinen Code umdrehen soll und zuerst B anfordert]

(5) Non-reproduceable reads (Nicht wiederholbare Lesezugriffe)

[Fortsetzung der Problem-Beispiele nach "(4) Verklemmung"]

Verwandt mit concurrency confusion ...

P_1 (Gibt A zweimal aus, nur lesend)	P_2 (Dekrementiert A, nur 1 Variable beteiligt)
lock A	
read A→a_1	
print a_1	
unlock A	
	lock A
	read A→a_2
	a_2-1→a_2
	write a_2→A
	unlock A
lock A	
read A→a_1	
print a_1	
unlock A	

2-Phasen-Sperrprotokoll bei P_1 verletzt, mit 2-Phasen-Sperrprotokoll würde P_2 verzögert.
[Beides sind sehr einfach Programme (P_1 nur lesen, P_2 nur 1 Variable beteiligt), dennoch Probleme; Ähnlich wie bei concurrency confusion wird kurz später ein anderer Wert ausgegeben, obwohl zweimal dasselbe auf dem Schirm erscheinen sollte]

Konkretisierung des Beispiels (Bibliothek)

Benutzer 1 (Recherche+Ausleihe)	Benutzer 2 (Ausleihe)
Benutzer: Wieviel ausleihbare Bücher über Transaktionen haben wir?	
System: 2 Stück	
	Benutzer: Ausleihen eines dieser Bücher
Benutzer: Reserviere mir die 2 Bücher	
System: Welche 2? Wir haben nur 1	

Oft ist diese Inkonsistenz tolerabel, sogar erwünscht bei Monitoring-Prozeß ["Meßgerät"].
[Normales Problem; Oft wird dies z.B. bei einer Bibliothek auch akzeptiert, da ein DBMS ohne Transaktionsmanagement wesentlich billiger ist; Falls man aus der linken Seite eine Transaktion macht, geht dies natürlich nur, wenn die einzelnen Anfragen schnell genug aufeinander folgen]
Intolerabel, wenn sich der Benutzer auf die erste Auskunft verlassen hat.

Abschottungsgrade (Levels of consistency)

[Eigentlich gibt es nur "konsistent" oder "nicht konsistent", dennoch kann man hier feiner unterteilen]

- **Keine Abschottung**
  Prozesse verwirrt, sehen schmutzige Zwischenzustände.
- **Schwache Abschottung**
  Z.B. non-reproducable reads (Bibliothekssystem) erlaubt, aber Schreibvorgänge abgesichert, lost update verhindert.
  [Lesen+Schreiben ist nicht abgesichert, aber Schreiben+Schreiben]
- **Hohe Abschottung (Thema der Vorlesung)**
  Sogenannte Serialisierbarkeit.
  [Die Prozesse laufen ab, als wenn sie hintereinander ablaufen würden]
- **Zu hohe Abschottung**
  Prozesse merken nichts voneinander, geht nicht ...
  (a) Antwortzeiten werden größer [Das merken die Prozesse dann doch]
  (b) Zulässige Wechselwirkungen [Eine Datenbank ist auch ein Art Kommunikationssystem, z.B. kann man die Frage stellen, ob zuvor jemand dieses Buch ausgeliehen hat]

Genauer ...
J. Gray, A. Reuter, Transaction Processing
[Auch mehr dazu im Praktikum; Bei diesem Thema geht es auch um "Cursor Stability", z.B. erhöht der Cursor in einer Spalte das Gehalt aller Angestellten, gleichzeitig versucht ein zweiter Prozeß von Herrn Maier ein anderes Attribut auszulesen, danach wird fälschlicherweise die Gehaltserhöhung bei Herrn Maier weitergeführt; D.h. auch der Cursor ist selbst eine knappe Resource; In unserem Modell gibt es aber so etwas wie den Cursor noch nicht]
In manchen Anwendungen Abschottung gerade nicht erwünscht.
[Z.B. eine gemeinsame Produktion eines Autos bei BMW, das Chassis und die Tür müssen zueinander passen; Oder falls mehrere Leute an einem Buch schreiben (z.B. jeder schreibt ein Kapitel), aber dann muß man noch viel genauer definieren, was erlaubt ist und was nicht]
Trade off zwischen Aktualität und Konsistenz!
[Ein weiteres Problem ist folgendes: Das statistische Bundesamt sammelt große Menge an Daten; Aber da man diese Daten meist über einen längeren Zeitraum sammelt und sie von verschiedenen Bundesländern auch zu unterschiedlichen Zeiten ankommen können, kommt es zu Inkonsistenzen; Z.B. betrachtet man die Inflationsrate zuerst nur von den Bundesländern, die die Daten bereits geliefert haben, anstatt auf alle Daten zu warten; Hier hat man einen Trade off zwischen Aktualität und Konsistenz der Daten]