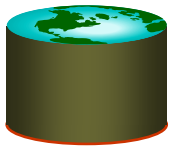
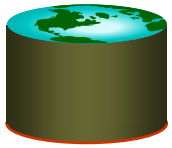
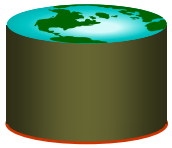


Upravljanje Transakcijama



Upravljanje Transakcijama

- Koncept Transakcije
- Stanja Transakcije
- Izvršavanje Transakcija
- Serijalizovanost
- Kontrola Serijalizovanosti
- Protokoli Bazirani na Zaključavanju
  - Dvofazni Protokol Zaključavanja
  - Protokol u Obliku Stabla
- Protokol Vremenskog Markiranja

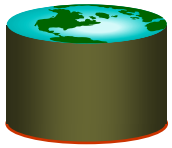


Koncept Transakcije

Start
Read (Poruka sa terminala)
Read (Stanje na računu)
Write (Stanje na računu)
Write (Dnevnik izvršenih transakcija)
Read (Stanje ATM uređaja)
Write (Stanje ATM uređaja)
Read (Stanje ekspoziture)
Write (Stanje ekspoziture)
Write (Poruka na terminal)
Commit

Program za podizanje i ulaganje novca sa *ATM* uređaja mora zadovoljiti više zahteva:

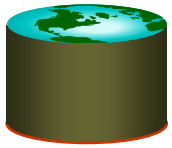
- Izvršenje programa se sastoji od više operacija
- Radi zadovoljenja performansi sistema, program se mora izvršavati konkurentno sa drugim programima
- Ako program počne sa izvršavanjem, mora se i završiti
- Rezultati rada programa, moraju biti trajni i dokumentovani
- Program se mora izvršavati, saglasno specifikacijama, i u distribuiranom okruženju



Definicija Transakcije

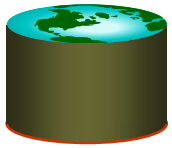
Transakcija je izvršenje programa koji predstavlja neku interakciju u realnom okruženju, i obezbeđuje očuvanje konzistentnosti baze podataka, kojom je okruženje modelirano. Transakcija mora biti:

- **Atomska.** *Transakcija* se mora izvršiti ili u potpunosti, ili nimalo. Ne sme se dozvoliti mogućnost izvršenja samo dela programa transakcije, jer bi to moglo dovesti do nekonzistentnog stanja baze.
- **Konzistentna.** Izvršenje transakcije nad bazom podataka koja je konzistentna, mora prevesti bazu u takođe konzistentno stanje, odnosno očuvati konzistentnost baze. Očuvanje konzistentnosti baze podrazumeva poštovanje svih ograničenja integriteta.
- **Nezavisna.** Kaže se da je skup transakcija nezavistan, ako je rezultat njihovog konkurentnog izvršavanja isti kao da su se izvršavale jedna po jedna, bez preklapanja izvršavanja. Saglasno tome, osobina *nezavisnosti* obezbeđuje konkurentno izvršavanje transakcija.
- **Trajna.** Pod *trajnošću* se podrazumeva osobina da kada se transakcija završi, sva ažuriranja podataka su smeštena u memoriju, tipično na disk, koji može sačuvati podatke i u slučaju kvara sistema. Posle oporavka sistema od kvara, podaci nisu izgubljeni, a time i rezultati rada transakcije, nego se mogu naći na odgovarajućem mestu na disku.



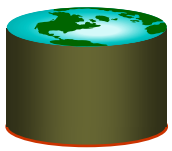
Primer

- Transaction prenosa $50 sa računa A na račun B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)
- **Atomska** – ako se transakcija prekine posle koraka 3 i pre koraka 6, sistem mora obezbediti da ažuriranja nisu upisana u bazu, jer će baza preći u nekonzistentno stanje.
- **Konzistentna** – izvršenjem transakcije suma A i B nije promenjena.
- **Nezavisna** – ako se između koraka 3 i 6 dozvoli drugoj transakciji pristup delimično ažuriranoj bazi, ona će videti nekonzistentnu bazu (suma *A* + *B* će biti manja nego što treba).
- **Trajna** – posle obaveštavanja korisnika da je transakcija izvršena (to jest, prenos $50 izvršen), ažurirana baza mora zadržati stanje uprkos mogućim kvarovima.

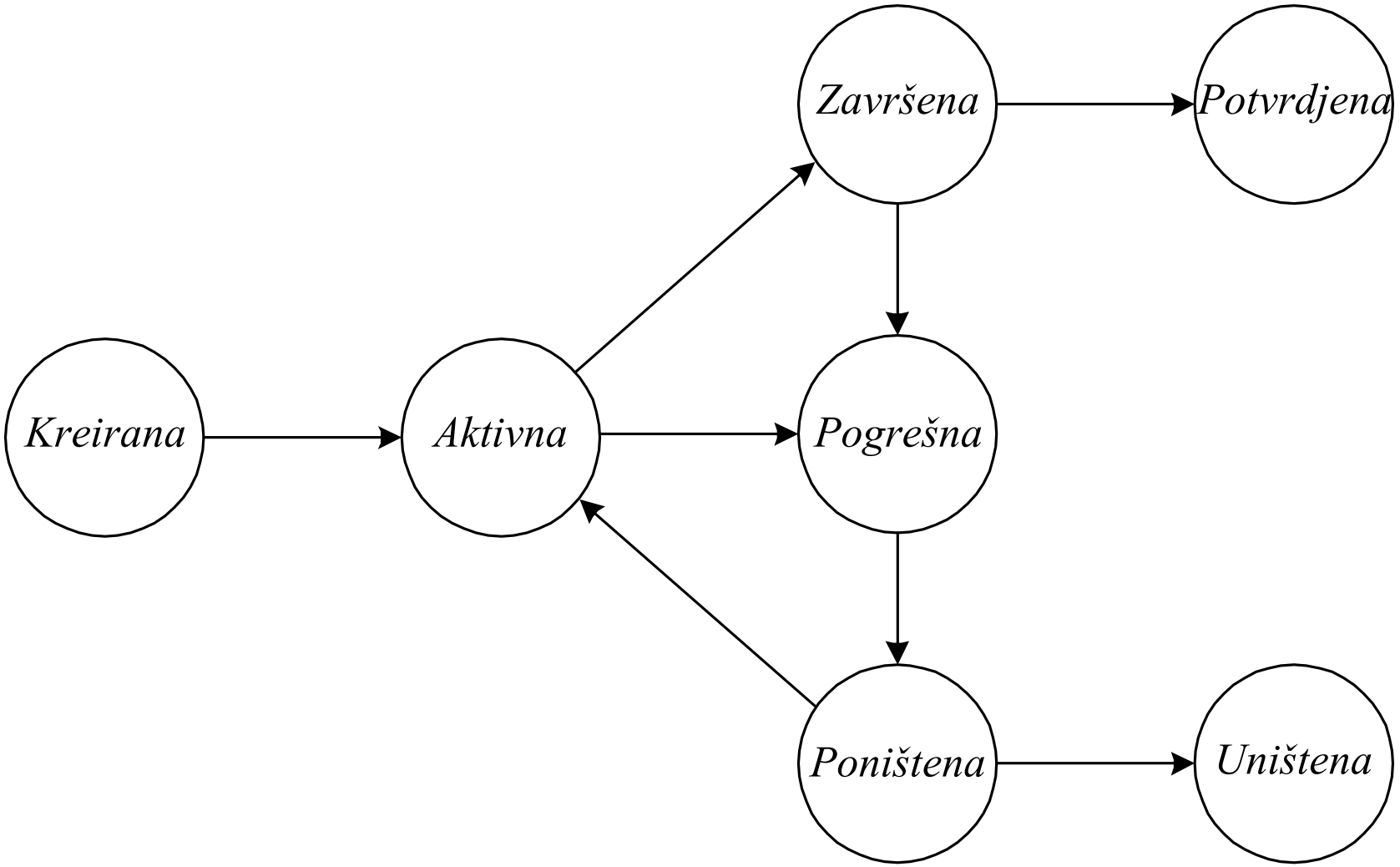


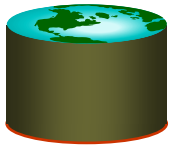
Stanja Transakcije

- **Kreirana** – spremna za aktiviranje
- **Aktivna** – počne da se izvršava
- **Završena** – pošto se izvrši zadnja instrukcija.
- **Pogrešna** – ako se inicirani upisi u bazu ne završe usled kvara sistema. U ovo stanje, transakcija može preći i iz aktivnog stanja, ukoliko u toku izvršavanja transakcije dođe do kvara sistema, bilo hardverskog, bilo softverskog.
- **Poništena** – Pošto se nije uspešno završila, transakciju je potrebno prevesti u stanje ***Poništena***, i vratiti bazu u prethodno konzistentno stanje. Posle restauracije baze, transakciju treba ponovo startovati, ako je hardverski ili softverski kvar, zbog koga se transakcija nije mogla završiti, u potpunosti otklonjen. Ukoliko kvar ne može biti u potpunosti otklonjen, ponovno startovanje transakcije nema smisla, već se transakcija prevodi u stanje ***Uništena***.
- **Uništena** – Tipičan slučaj je prisustvo logičke greške u transakciji. Takva greška može biti ispravljena samo ponovnim pisanjem programa transakcije.
- **Potvrđena** – svi upisi, inicirani u toku izvršenja transakcije, u bazu završeni i očuvana konzistentnost baze.



Stanja Transakcije - Dijagram





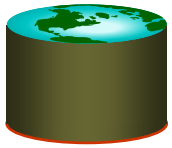
Izvršavanje Transakcija

Pretpostavimo da u multiprogramskom računarskom sistemu postoji skup od *n* aktivnih transakcija, $T=\{T_1, \ldots, T_n\}$. Transakcije iz ovog skupa se mogu izvršavati tako, da se prvo izvrši kompletno jedna transakcija, zatim druga, pa treća, i tako redom dok se kompletno ne izvrši i poslednja transakcija iz skupa *T*. Transakcije iz ovog skupa se mogu izvršavati i tako, da se pre kompletnog izvršenja jedne transakcije, startuje druga, pre kompletnog završetka druge, startuje neka treća transakcija, ili, nastavi prva, i tako naizmenično, sve do kompletnog završetka svih transakcija. Saglasno tome, kako su se transakcije iz skupa *T* izvršavale, kaže se da su se transakcije izvršavale:

- Serijski, ili
- Konkurentno (Neserijski)

Zavisno od toga, da li izvršavanje različitim redosledima, bilo serijskim ili konkurentnim, daje isti rezultat ili ne, kao i da li konkurentni redosled daje isti rezultat kao neki serijski redosled, definišu se i sledeći redosledi:

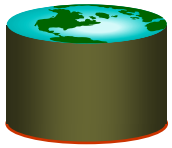
- Ekvivalentni i
- Serijalizovani redosled



Redosled 1

- Neka T_1 prenosi \$50 sa računa *A* na račun *B*, a T_2 prenosi 10% stanja računa *A* u *B.*
- Serijski redosled u kome iza T_1 sledi T_2:

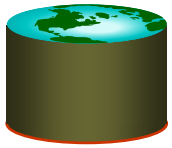
T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)



Redosled 2

- Serijski redosled u kome iza T_2 sledi T_1:

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	

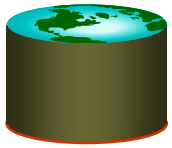


Redosled 3

- Neka su T_1 i T_2 prethodno definisane transakcije. Sledeći redosled nije serijski, ali je *ekvivalentan* Redosledu 1.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

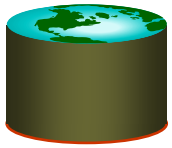
U Redosledima 1, 2 i 3, suma A + B ostaje nepromenjena.



Redosled 4

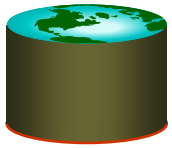
- Sledeći konkurentni redosled dovodi do promene sume (*A* + *B*).

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)



Serijalizovanost

- **Osnovna Pretpostavka** – Svaka transakcija obezbeđuje konzistentnost baze.
- Tako i serijsko izvršavanje skupa transakcija obezbeđuje konzistentnost baze.
- Neki neserijski redosled je serijalizovan ako je ekvivalentan serijskom redosledu. Definišu se dve vrste ekvivalentnosti redosleda pa saglasno tome i serijalizovanosti:
  1. **”conflict” serijalizovanost**
  2. **”view” serijalizovanost**
- Ignorišemo sve operacije izuzev **read** i **write** instrukcija, i pretpostavljamo da transakcije mogu izvršavati proizvoljna računanja na podacima u lokalnim baferima između operacija read i write. Pojednostavljeni redosledi se prema tome sastoje samo od **read** i **write** instrukcija.

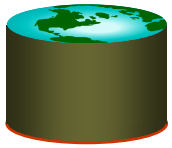


Konfliktne Instrukcije

- Instrukcije I_i i I_j transakcija T_i i T_j respektivno, su **konfliktne** ako i samo ako postoji podatak Q kome pristupaju i I_i i I_j, i najmanje jedna od ovih instrukcija je **write**(Q).

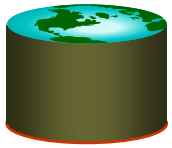
  1. I_i = **read**(Q), I_j = **read**(Q). I_i i I_j nisu konfliktne.
  2. I_i = **read**(Q), I_j = **write**(Q). Konfliktne.
  3. I_i = **write**(Q), I_j = **read**(Q). Konfliktne.
  4. I_i = **write**(Q), I_j = **write**(Q). Konfliktne.

- Konflikt između I_i i I_j forsira redosled njihovog izvršavanja.
  - Ako su I_i i I_j nekonfliktne, rezultat njihovog izvršavanja ostaje isti bez obzira kojim se redosledom izvršavale.



Konfliktna Serijalizovanost

- Ako se redosled *S* može transformisati u redosled *S´* promenom redosleda ne-konfliktnih instrukcija, kažemo da su redosledi *S* i *S´* **konfliktno ekvivalentni**.
- Kažemo da je redosled *S* **konfliktno serijalizovan** ako je konfliktno ekvivalentan serijskom redosledu



Konfliktna Serijalizovanost (nastavak)

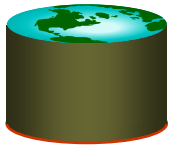
- Redosled 3 se može transformisati u Redosled 6, serijski redosled u kome T_2 sledi T_1, promenom redosleda ne-konfliktnih instrukcija.
  - Prema tome Redosled 3 je konfliktno serijalizovan.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Redosled 3

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Redosled 6

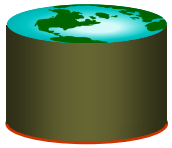


Konfliktna Serijalizovanost (nastavak)

- Primer redosleda koji nije konfliktno serijalizovan:

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

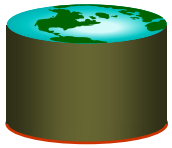
- Ne možemo promeniti redosled instrukcija da bi dobili bilo serijski redosled < T_3, T_4 >, ili serijski redosled < T_4, T_3 >.



View Serijalizovanost

- Neka su *S* i *S´* dva redosleda sa istim skupom transakcija. *S* i *S´* su **view ekvivalentni** ako važe sledeća tri uslova:
  1. Za svaki podatak *Q,* ako transakcija T_i čita početnu vrednost *Q* u redosledu *S,* tada transakcija T_i mora, u redosledu *S´*, takođe čitati početnu vrednost *Q.*
  2. Za svaki podatak *Q,* ako transakcija T_i izvršava **read**(*Q*) u redosledu *S*, a tu vrednost je upisala transakcija T_j , tada transakcija T_i mora i u redosledu *S´* takođe čitati vrednost *Q* koju je upisala transakcija T_j .
  3. Za svaki podatak *Q*, transakcija koja zadnja izvršava **write**(*Q*) u redosledu *S* mora zadnja izvršavati **write**(*Q*) i u redosledu *S´*.

View equivalentnost se takođe bazira na **read** i **write** operacijama.

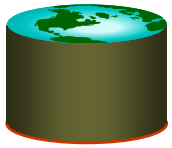


View Serijalizovanost (nastavak)

- A schedule *S* is **view serializable** it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Below is a schedule which is view-serializable but *not* conflict serializable.

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

- What serial schedule is above equivalent to?
- Every view serializable schedule that is not conflict serializable has **blind writes.**

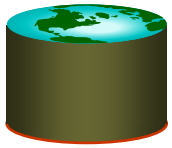


Druge Serijalizovanosti

- The schedule below produces same outcome as the serial schedule < T_1, T_5 >, yet is not conflict equivalent or view equivalent to it.

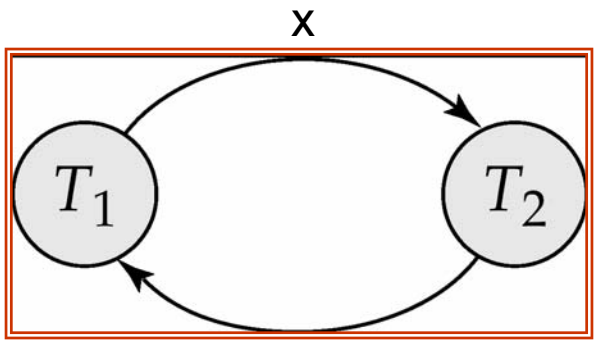
T_1	T_5
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

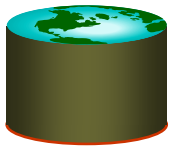
- Determining such equivalence requires analysis of operations other than read and write.



Provera Serijalizovanosti

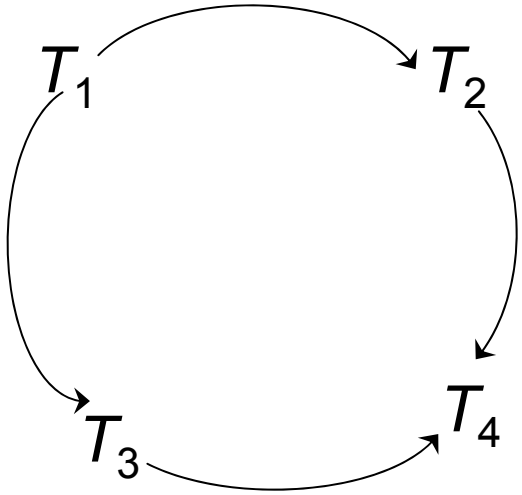
- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- **Precedence graph** — a direct graph where the vertices are the transactions (names).
- We draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.
- **Example 1**

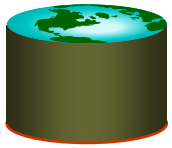




Primer provere Serijalizovanosti

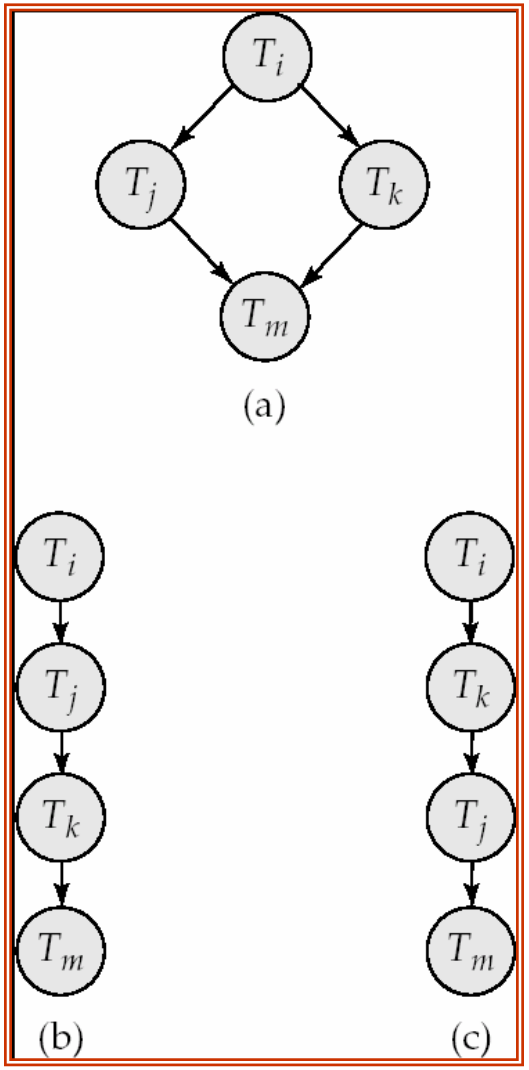
T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				

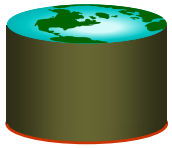




Provera Konfliktne Serijalizovanosti

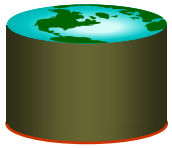
- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph.
  - (Better algorithms take order $n + e$ where e is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.
  - This is a linear order consistent with the partial order of the graph.
  - For example, a serializability order for Schedule A would be $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$
    - Are there others?





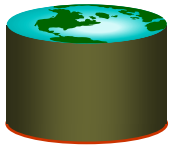
Provera View Serijalizovanosti

- The precedence graph test for conflict serializability cannot be used directly to test for view serializability.
  - Extension to test for view serializability has cost exponential in the size of the precedence graph.
- The problem of checking if a schedule is view serializable falls in the class of *NP*-complete problems.
  - Thus existence of an efficient algorithm is *extremely* unlikely.
- However practical algorithms that just check some **sufficient conditions** for view serializability can still be used.



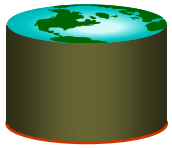
Kontrola Serijalizovanosti

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - are recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
  - Are serial schedules recoverable/cascadeless?
- Testing a schedule for serializability *after* it has executed is a little too late!
- **Goal** – to develop concurrency control protocols that will assure serializability.



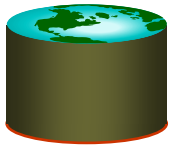
Kontrola vs Provera Serijalizovanosti

- Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless.
- Concurrency control protocols generally do not examine the precedence graph as it is being created
  - Instead a protocol imposes a discipline that avoids nonseralizable schedules.
  - We study such protocols in Chapter 16.
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.
- Tests for serializability help us understand why a concurrency control protocol is correct.



Protokoli Bazirani na Zaključavanju

- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes :
  1. *exclusive (X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
  2. *shared (S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction.
- Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

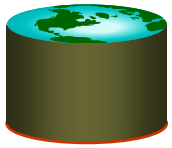


Protokoli Bazirani na Zaključavanju (nast)

- Matrica kompatibilnosti zaključavanja

	S	X
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item,
  - but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.

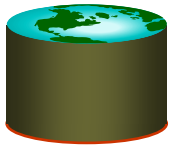


Protokoli Bazirani na Zaključavanju (nast)

- Example of a transaction performing locking:

  T_2: **lock-S**(*A*);
  read (*A*);
  unlock(*A*);
  lock-S(*B*);
  read (*B*);
  unlock(*B*);
  display(*A*+*B*)

- Locking as above is not sufficient to guarantee serializability — if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

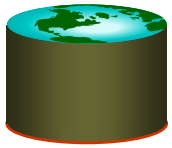


Problem Deadlock-a

- Consider the partial schedule

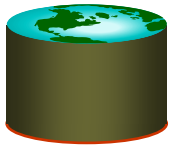
T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

- Neither T_3 nor T_4 can make progress — executing **lock-S**(*B*) causes T_4 to wait for T_3 to release its lock on *B*, while executing **lock-X**(*A*) causes T_3 to wait for T_4 to release its lock on *A*.
- Such a situation is called a **deadlock**.
  - To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.



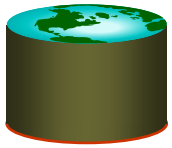
Problem iznurivanja (starvation)

- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
- **Starvation** is also possible if concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
  - The same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control manager can be designed to prevent starvation.



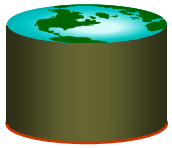
Dvofazni Protokol Zaključavanja

- This is a protocol which ensures conflict-serializable schedules.
- Phase 1: Growing Phase
  - transaction may obtain locks
  - transaction may not release locks
- Phase 2: Shrinking Phase
  - transaction may release locks
  - transaction may not obtain locks
- The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock).



Dvofazni Protokol Zaključavanja (nast)

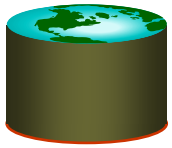
- Two-phase locking *does not* ensure freedom from deadlocks
- Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking**. Here a transaction must hold all its exclusive locks till it commits/aborts.
- **Rigorous two-phase locking** is even stricter: here *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.



Dvofazni Protokol Zaključavanja (nast)

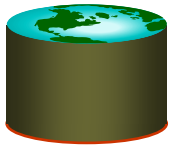
- There can be conflict serializable schedules that cannot be obtained if two-phase locking is used.
- However, in the absence of extra information (e.g., ordering of access to data), two-phase locking is needed for conflict serializability in the following sense:

  Given a transaction T_i that does not follow two-phase locking, we can find a transaction T_j that uses two-phase locking, and a schedule for T_i and T_j that is not conflict serializable.



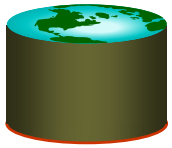
Promene tipa zaključavanja

- Two-phase locking with lock conversions:
  - First Phase:
    - can acquire a lock-S on item
    - can acquire a lock-X on item
    - can convert a lock-S to a lock-X (upgrade)
  - Second Phase:
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (downgrade)
- This protocol assures serializability. But still relies on the programmer to insert the various locking instructions.



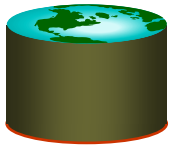
Implementacija Zaključavanja

- A **lock manager** can be implemented as a separate process to which transactions send lock and unlock requests
- The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock)
- The requesting transaction waits until its request is answered
- The lock manager maintains a data-structure called a **lock table** to record granted locks and pending requests
- The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked

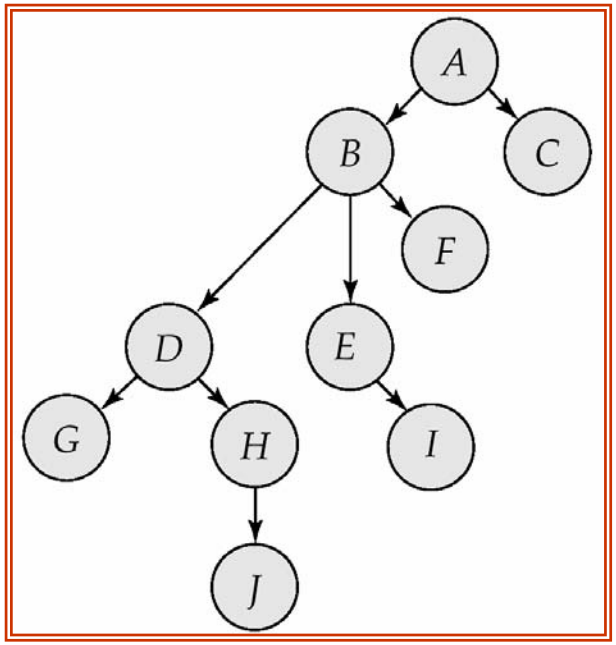


Protokoli Bazirani na Grafovima

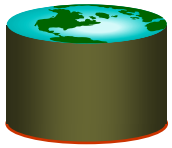
- Graph-based protocols are an alternative to two-phase locking
- Impose a partial ordering $\rightarrow$ on the set **D** = {d_1, d_2 ,..., d_h} of all data items.
  - If $d_i \rightarrow d_j$ then any transaction accessing both d_i and d_j must access d_i before accessing d_j.
  - Implies that the set **D** may now be viewed as a directed acyclic graph, called a *database graph*.
- The *tree-protocol* is a simple kind of graph protocol.



Protokol u Obliku Stabla

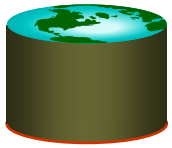


- Only exclusive locks are allowed.
- The first lock by T_i may be on any data item. Subsequently, a data Q can be locked by T_i only if the parent of Q is currently locked by T_i.
- Data items may be unlocked at any time.



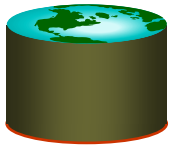
Protokol u Obliku Stabla (nastavak)

- The tree protocol ensures conflict serializability as well as freedom from deadlock.
- Unlocking may occur earlier in the tree-locking protocol than in the two-phase locking protocol.
  - shorter waiting times, and increase in concurrency
  - protocol is deadlock-free, no rollbacks are required
- Drawbacks
  - Protocol does not guarantee recoverability or cascade freedom
    - Need to introduce commit dependencies to ensure recoverability
  - Transactions may have to lock data items that they do not access.
    - increased locking overhead, and additional waiting time
    - potential decrease in concurrency
- Schedules not possible under two-phase locking are possible under tree protocol, and vice versa.



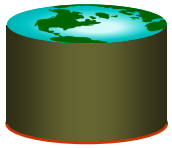
Protokol Vremenskog Markiranja

- Each transaction is issued a timestamp when it enters the system. If an old transaction T_i has time-stamp TS(T_i), a new transaction T_j is assigned time-stamp TS(T_j) such that TS(T_i) <TS(T_j).
- The protocol manages concurrent execution such that the time-stamps determine the serializability order.
- In order to assure such behavior, the protocol maintains for each data Q two timestamp values:
  - **W-timestamp**(Q) is the largest time-stamp of any transaction that executed **write**(Q) successfully.
  - **R-timestamp**(Q) is the largest time-stamp of any transaction that executed **read**(Q) successfully.



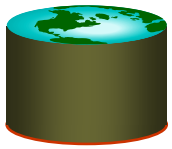
Protokol Vremenskog Markiranja (nast)

- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.
- Suppose a transaction T_i issues a **read**(Q)
  1. If TS(T_i) ≤ **W**-timestamp(Q), then T_i needs to read a value of Q that was already overwritten.
     - Hence, the **read** operation is rejected, and T_i is rolled back.
  2. If TS(T_i)≥ **W**-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(T_i).



Protokol Vremenskog Markiranja (nast)

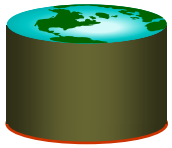
- Suppose that transaction T_i issues **write**(Q).
  1. If TS(T_i) < R-timestamp(Q), then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced.
     - Hence, the **write** operation is rejected, and T_i is rolled back.
  2. If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q.
     - Hence, this **write** operation is rejected, and T_i is rolled back.
  3. Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to TS(T_i).



Primer

A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5

T_1	T_2	T_3	T_4	T_5
				read(X)
	read(Y)			
read(Y)				
		write(Y)		
		write(Z)		
				read(Z)
	read(X)			
	abort			
read(X)				
		write(Z)		
		abort		
				write(Y)
				write(Z)



Korektnost Protokola Vrem. Markiranja

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:

Thus, there will be no cycles in the precedence graph

- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.
- But the schedule may not be cascade-free, and may not even be recoverable.