The picture can't be displayed.

Chapter 14: Transactions

Database System Concepts, 6th Ed.

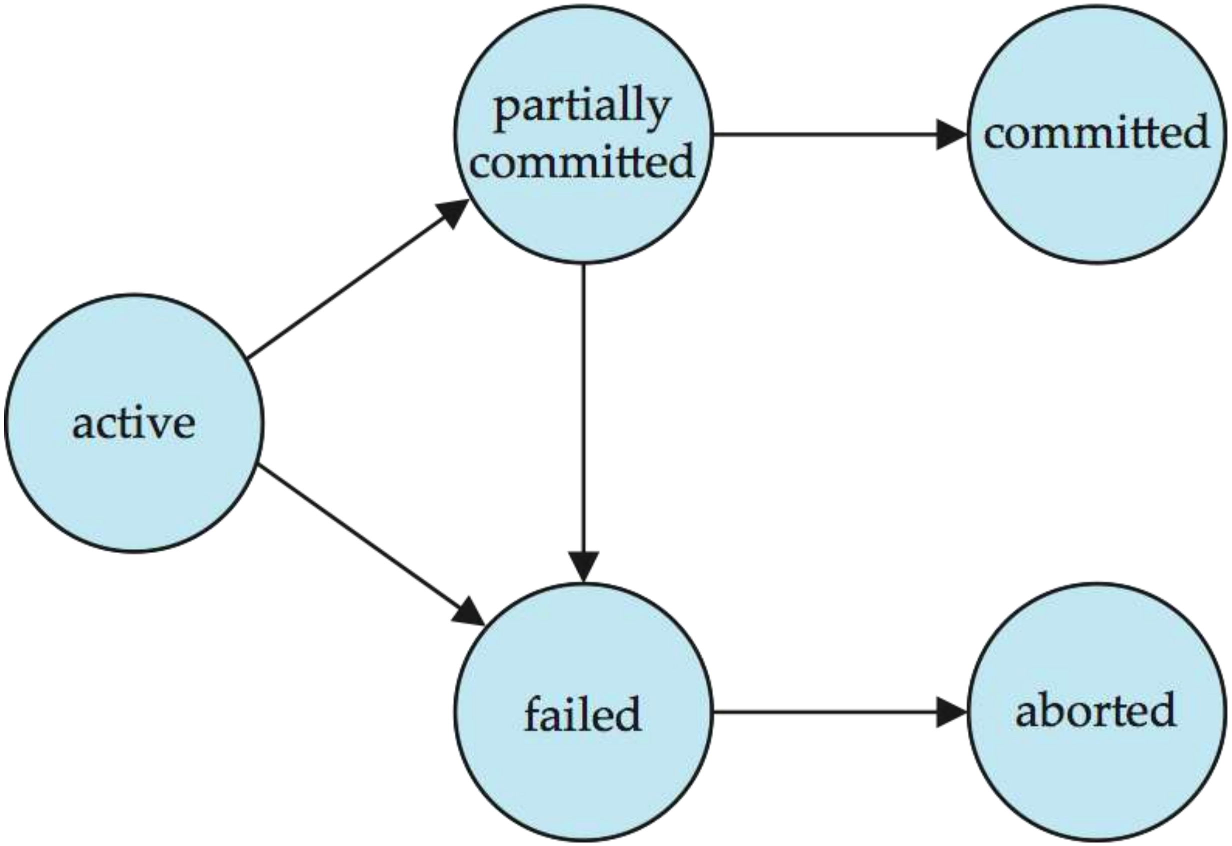
Chapter 14: Transactions

- Transaction Concept
- Transaction State
- Concurrent Executions
- Serializability
- Implementation of Isolation

Transaction State

- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed --** after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - restart the transaction
    - can be done only if no internal logical error
  - kill the transaction
- **Committed** – after successful completion.

Transaction State (Cont.)



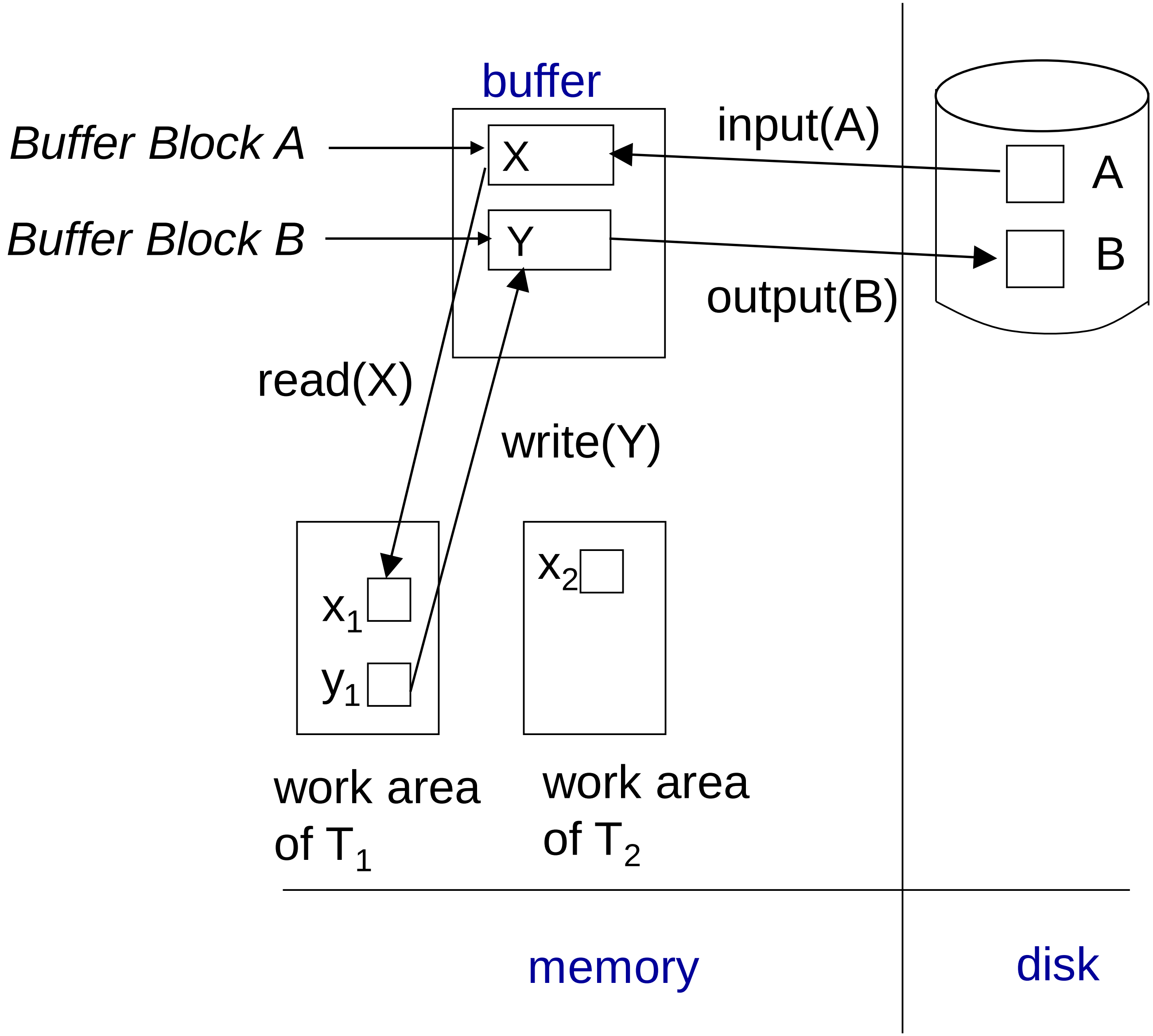
Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- E.g. transaction to transfer \$50 from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B)*
- Two main issues to deal with:
  - **Failures** of various kinds, such as hardware failures and system crashes
  - **Concurrent** execution of multiple transactions

Example of Fund Transfer

- Transaction to transfer $50 from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)
- **Atomicity requirement**
  - **if the transaction fails** after step 3 and before step 6, **money will be "lost"** leading to an inconsistent database state
    - Failure could be due to software or hardware
  - the **system should ensure** that updates of a partially executed **transaction** are **not reflected in the database**
- **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the **updates to the database by the transaction must persist** even if there are software or hardware failures.

Example of Data Access



Example of Fund Transfer (Cont.)

- Transaction to transfer $50 from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B)*
- **Consistency requirement** in above example:
  - the **sum of A and B is unchanged** by the execution of the transaction
- In general, consistency requirements include
  - **Explicitly** specified **integrity constraints** such as primary keys and foreign keys
  - **Implicit** integrity constraints
    - e.g. **sum of balances of all accounts**, minus sum of loan amounts **must equal value of cash-in-hand**
  - A transaction must see a consistent database.
  - During transaction execution the database may be temporarily inconsistent.
  - When the transaction completes successfully the database must be consistent
    - Erroneous transaction logic can lead to inconsistency

Example of Fund Transfer (Cont.)

- **Isolation requirement** — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum *A* + *B* will be less than it should be).

T1	**T2**
1. **read**(*A*)	
2. *A* := *A* – 50	
3. **write**(*A*)	
	read(A), read(B), print(A+B)
4. **read**(*B*)	
5. *B* := *B* + 50	
6. **write**(*B*	

- Isolation can be ensured trivially by running transactions **serially**
  - that is, one after the other.
- **However**, executing multiple transactions concurrently has **significant benefits**, as we will see later.

ACID Properties

A **transaction** is a unit of program execution that accesses and possibly updates various data items.To preserve the integrity of data the database system must ensure:

- **Atomicity.** Either **all** operations of the transaction are properly reflected in the database **or none are**.
- **Consistency.** Execution of a transaction in isolation **preserves the consistency** of the database.
- **Isolation.** Although multiple transactions may execute concurrently, each **transaction** must be **unaware of** other **concurrently executing transactions**. Intermediate transaction results must be hidden from other concurrently executed transactions.
  - That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.
- **Durability.** After a transaction completes successfully, the changes it has made to the database **persist**, even if there are system failures.

Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **increased processor and disk utilization**, leading to better transaction *throughput*
    - E.g. one transaction can be using the CPU while another is reading from or writing to the disk
  - **reduced average response time** for transactions: short transactions need not wait behind long ones.
- **Concurrency control schemes** – mechanisms to achieve isolation
  - that is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database
    - Will study in Chapter 16, after studying notion of correctness of concurrent executions.

Schedules

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a commit instructions as the last statement
  - by default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

Schedule 1

- Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B.
- A serial schedule in which T_1 is followed by T_2 :

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

Schedule 2

- A serial schedule where T_2 is followed by T_1

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

Schedule 3

- Let T_1 and T_2 be the transactions defined previously. The following schedule is **not a serial schedule**, but it is ***equivalent*** to Schedule 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)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

In Schedules 1, 2 and 3, the sum **A + B is preserved**.

Schedule 4

- The following concurrent schedule **does not preserve** the value of ***(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)	
commit	
	$B := B + temp$
	write (B)
	commit

Serializability

- **Basic Assumption** – Each transaction preserves database consistency.
- Thus serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) **schedule is serializable if it is equivalent to a serial schedule**. Different forms of schedule equivalence give rise to the notions of:
  1. **conflict serializability**
  2. **view serializability**

Simplified view of transactions

- We **ignore** operations **other** than **read** and **write** instructions
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
- Our **simplified schedules** consist of only **read** and **write** instructions.

Conflicting Instructions

- Instructions l_i and l_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both l_i and l_j, and at least one of these instructions wrote Q.
  1. l_i = **read**(Q), l_j = **read**(Q). l_i and l_j **don't conflict**.
  2. l_i = **read**(Q), l_j = **write**(Q). They **conflict**.
  3. l_i = **write**(Q), l_j = **read**(Q). They **conflict**
  4. l_i = **write**(Q), l_j = **write**(Q). They **conflict**
- Intuitively, a conflict between l_i and l_j forces a (logical) temporal order between them.
  - If l_i and l_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflict Serializability

- If a schedule *S* can be transformed into a schedule *S*´ by a series of swaps of non-conflicting instructions, we say that *S* and *S*´ are **conflict equivalent**.
- We say that a schedule *S* is **conflict serializable** if it is conflict **equivalent to a serial schedule**

- Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable.

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

Schedule 3

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

Schedule 6

Anomalies with Interleaved Execution

- Unrepeatable Reads (RW Conflicts):

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

Anomalies (Continued)

- Overwriting Uncommitted Data (WW Conflicts):

T_1	T_2
write (A)	
	write (A)
	write (B)
	C
write (B)	
C	

The picture can't be displayed.

End of Chapter 14

Database System Concepts, 6th Ed.

Figure 14.01

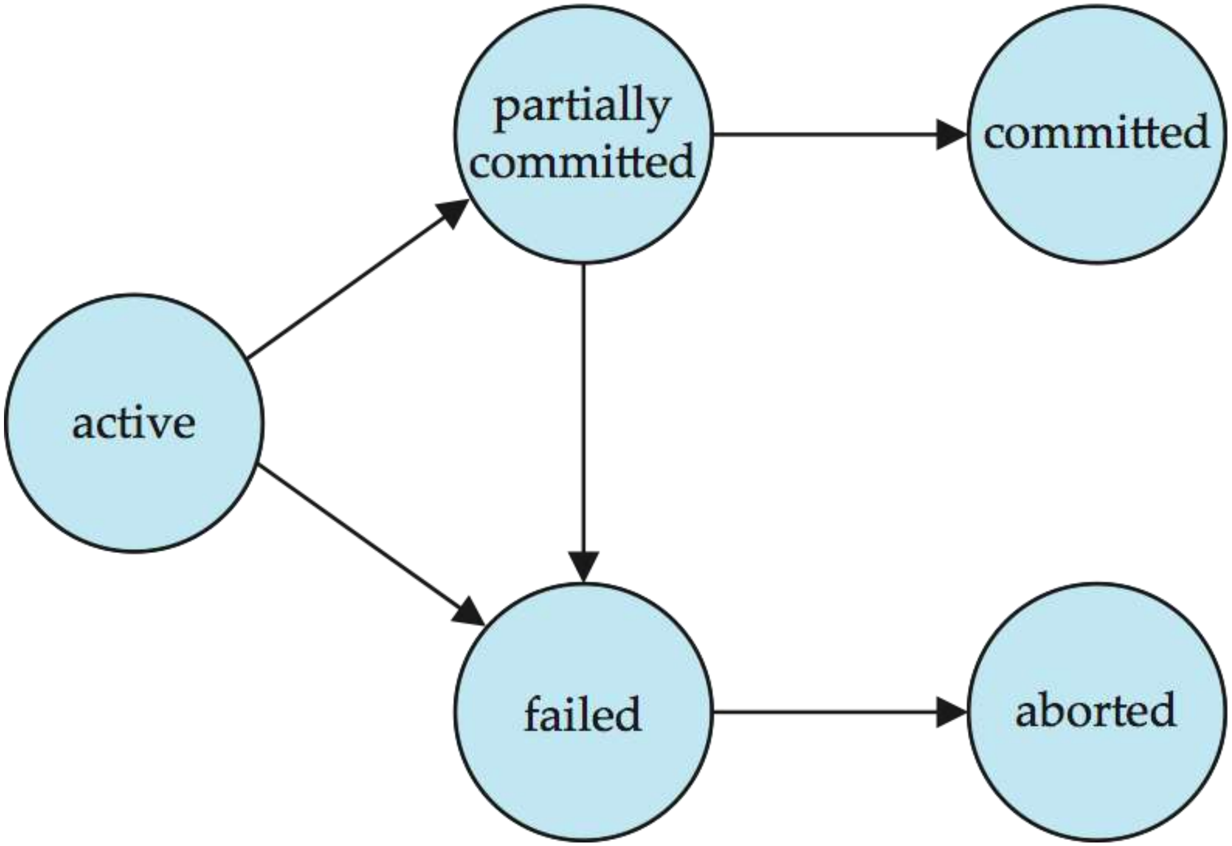


Figure 14.02

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

Figure 14.03

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

Figure 14.04

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)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

Figure 14.05

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)	
commit	
	$B := B + temp$
	write (B)
	commit

Figure 14.06

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

Figure 14.07

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

Figure 14.08

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

Figure 14.09

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

Figure 14.10

Figure 14.11

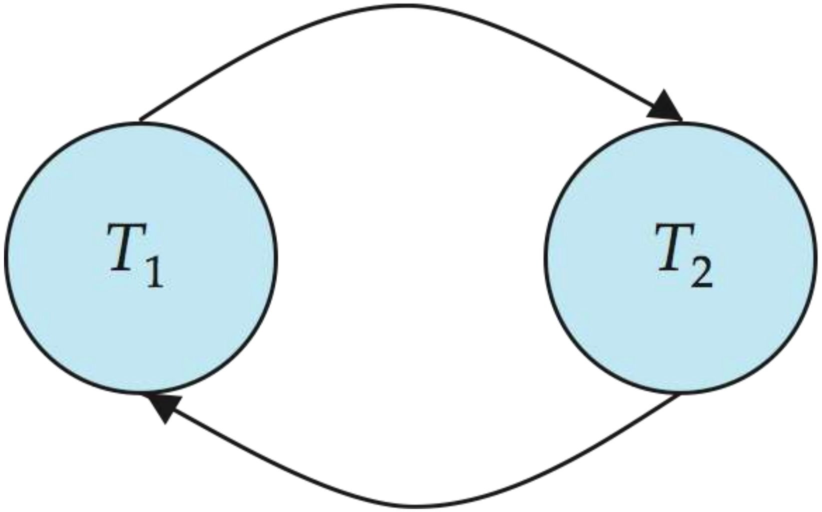


Figure 14.12

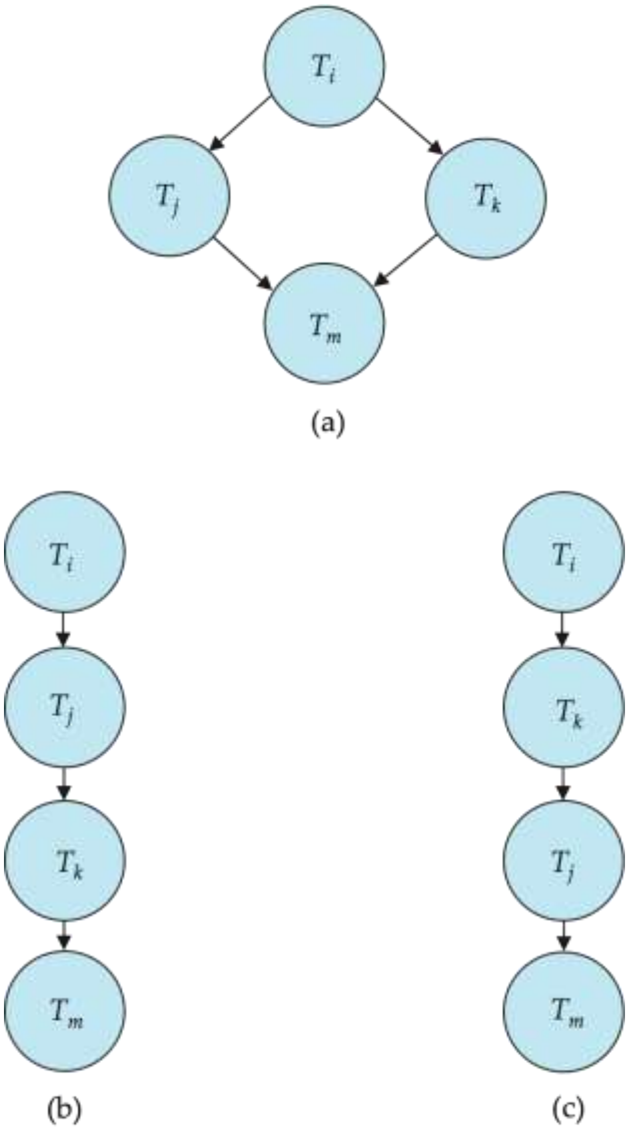


Figure 14.13

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)

Figure 14.14

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

Figure 14.15

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

Figure 14.16