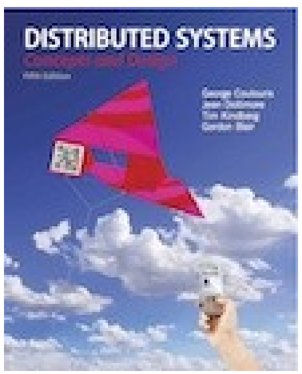
Slides for Chapter 16: Transactions and Concurrency Control



From **Coulouris, Dollimore, Kindberg and Blair**

Distributed Systems: Concepts and Design

Edition 5, © Addison-Wesley 2012

Figure 16.1
Operations of the *Account* interface

deposit(amount)
deposit amount in the account
withdraw(amount)
withdraw amount from the account
getBalance() -> amount
return the balance of the account
setBalance(amount)
set the balance of the account to amount

Operations of the *Branch* interface

create(name) -> account
create a new account with a given name
lookUp(name) -> account
return a reference to the account with the given name
branchTotal() -> amount
return the total of all the balances at the branch

Figure 16.2
A client's banking transaction

Transaction T:
a.withdraw(100);
b.deposit(100);
c.withdraw(200);
b.deposit(200);

Figure 16.3
Operations in *Coordinator* interface

openTransaction() -> trans;

starts a new transaction and delivers a unique TID *trans*. This identifier will be used in the other operations in the transaction.

closeTransaction(trans) -> (commit, abort);

ends a transaction: a *commit* return value indicates that the transaction has committed; an *abort* return value indicates that it has aborted.

abortTransaction(trans);

aborts the transaction.

Figure 16.4
Transaction life histories

Successful	Aborted by client	Aborted by server	
openTransaction	openTransaction		openTransaction
operation	operation		operation
operation	operation		operation
•	•	server aborts transaction →	•
•	•		•
operation	operation		operation ERROR reported to client
closeTransaction	abortTransaction		

Figure 16.5
The lost update problem

Transaction *T* :		**Transaction *U*:**	
balance = b.getBalance(); *b.setBalance(balance*1.1);* *a.withdraw(balance/10)*		*balance = b.getBalance();* *b.setBalance(balance*1.1);* *c.withdraw(balance/10)*	
balance = b.getBalance();	$200		
		balance = b.getBalance();	$200
		*b.setBalance(balance*1.1);*	$220
*b.setBalance(balance*1.1);*	$220		
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$280

Figure 16.6
The inconsistent retrievals problem

Transaction *V*:		**Transaction *W*:**	
a.withdraw(100) *b.deposit(100)*		*aBranch.branchTotal()*	
a.withdraw(100);	\$100		
		total = a.getBalance()	\$100
		total = total+b.getBalance()	\$300
		total = total+c.getBalance()	
b.deposit(100)	\$300	:	

Figure 16.7
A serially equivalent interleaving of *T* and *U*

Transaction *T*:		**Transaction *U*:**	
balance = b.getBalance()		*balance = b.getBalance()*	
*b.setBalance(balance*1.1)*		*b.setBalance(balance*1.1)*	
a.withdraw(balance/10)		*c.withdraw(balance/10)*	
balance = b.getBalance()	$200		
*b.setBalance(balance*1.1)*	$220		
		balance = b.getBalance()	$220
		*b.setBalance(balance*1.1)*	$242
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$278

Figure 16.8
A serially equivalent interleaving of *V* and *W*

Transaction *V*:		**Transaction *W*:**	
a.withdraw(100); *b.deposit(100)*		*aBranch.branchTotal()*	
a.withdraw(100);	$100		
b.deposit(100)	$300		
		total = a.getBalance()	$100
		total = total+b.getBalance()	$400
		total = total+c.getBalance()	
		...	

Figure 16.9
Read and *write* operation conflict rules

Operations of different transactions		*Conflict*	*Reason*
read	*read*	No	Because the effect of a pair of *read* operations does not depend on the order in which they are executed
read	*write*	Yes	Because the effect of a *read* and a *write* operation depends on the order of their execution
write	*write*	Yes	Because the effect of a pair of *write* operations depends on the order of their execution

Figure 16.10
A non-serially equivalent interleaving of operations of transactions *T* and *U*

Transaction *T*:	**Transaction *U*:**
x = read(i)	
write(i, 10)	
	y = read(j)
	write(j, 30)
write(j, 20)	
	z = read (i)

Figure 16.11
A dirty read when transaction *T* aborts

Transaction *T*:		**Transaction *U*:**	
a.getBalance() *a.setBalance(balance + 10)*		*a.getBalance()* *a.setBalance(balance + 20)*	
balance = a.getBalance()	$100		
a.setBalance(balance + 10)	$110		
		balance = a.getBalance()	$110
		a.setBalance(balance + 20)	$130
		commit transaction	
abort transaction			

Figure 16.12
Overwriting uncommitted values

Transaction *T*:		**Transaction *U*:**	
a.setBalance(105)		*a.setBalance(110)*	
	$100		
a.setBalance(105)	$105		
		a.setBalance(110)	$110

Figure 16.13
Nested transactions

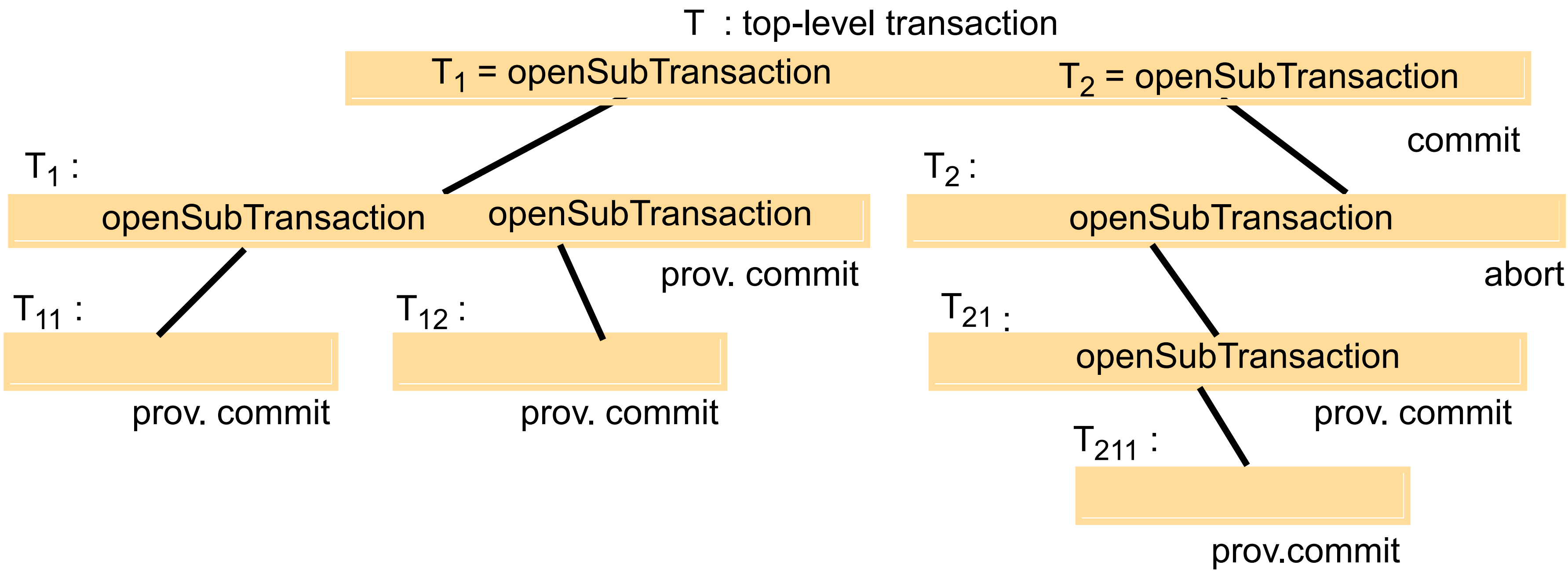


Figure 16.14
Transactions *T* and *U* with exclusive locks

Transaction *T*:		**Transaction *U*:**	
balance = b.getBalance() *b.setBalance(bal*1.1)* *a.withdraw(bal/10)*		*balance = b.getBalance()* *b.setBalance(bal*1.1)* *c.withdraw(bal/10)*	
Operations	Locks	Operations	Locks
openTransaction			
bal = b.getBalance()	lock *B*		
*b.setBalance(bal*1.1)*		*openTransaction*	
a.withdraw(bal/10)	lock *A*	*bal = b.getBalance()*	waits for *T*'s lock on *B*
closeTransaction	unlock *A*, *B*	• • •	
			lock *B*
		*b.setBalance(bal*1.1)*	
		c.withdraw(bal/10)	lock *C*
		closeTransaction	unlock *B*, *C*

Figure 16.15
Lock compatibility

For one object		*Lock requested*	
		read	*write*
Lock already set	*none*	OK	OK
	read	OK	wait
	write	wait	wait

Figure 16.16
Use of locks in strict two-phase locking

1. When an operation accesses an object within a transaction:
   (a) If the object is not already locked, it is locked and the operation proceeds.
   (b) If the object has a conflicting lock set by another transaction, the transaction must wait until it is unlocked.
   (c) If the object has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds.
   (d) If the object has already been locked in the same transaction, the lock will be promoted if necessary and the operation proceeds. (Where promotion is prevented by a conflicting lock, rule (b) is used.)
2. When a transaction is committed or aborted, the server unlocks all objects it locked for the transaction.

Figure 16.17
Lock class

```
public class Lock {
        private Object object;          // the object being protected by the lock
        private Vector holders;         // the TIDs of current holders
        private LockType lockType;      // the current type
        public synchronized void acquire(TransID trans,   LockType aLockType ){
                while(/*another transaction holds the lock in conflicing mode*/) {
                        try {
                                wait();
                        }catch ( InterruptedException e){/*...*/ }
                }
                if(holders.isEmpty()) { // no TIDs  hold lock
                        holders.addElement(trans);
                        lockType  = aLockType;
                } else if(/*another transaction holds the lock, share it*/ ) ){
                        if(/* this transaction not a holder*/) holders.addElement(trans);
                } else if (/* this transaction is a holder but needs a more exclusive lock*/)
                                lockType.promote();
                }
        }
```

Continues on next slide

Figure 16.17 continued

```
public synchronized void release(TransID trans ){
                    holders.removeElement(trans);      // remove this holder
                    // set locktype to none
                    notifyAll();
          }
}
```

Figure 16.18
LockManager class

```
public class LockManager {
    private Hashtable theLocks;

    public  void setLock(Object object, TransID trans,  LockType lockType){
            Lock foundLock;
            synchronized(this){
                    // find the lock associated with object
                    // if there isn't one, create it and add to the hashtable
            }
        foundLock.acquire(trans, lockType);
    }

   // synchronize this one because we want to remove all entries
    public synchronized void unLock(TransID trans) {
            Enumeration e = theLocks.elements();
            while(e.hasMoreElements()){
          Lock aLock = (Lock)(e.nextElement());
          if(/* trans is a holder of this lock*/ ) aLock.release(trans);
        }
    }
}
```

Figure 16.19
Deadlock with write locks

Transaction *T*		Transaction *U*	
Operations	**Locks**	**Operations**	**Locks**
a.deposit(100);	write lock *A*		
		b.deposit(200)	write lock *B*
b.withdraw(100)			
• • •	waits for *U*'s lock on *B*	*a.withdraw(200);*	waits for *T*'s lock on *A*
		• • •	
• • •		• • •	
• • •		• • •	

Figure 16.20
The wait-for graph for Figure 16.19

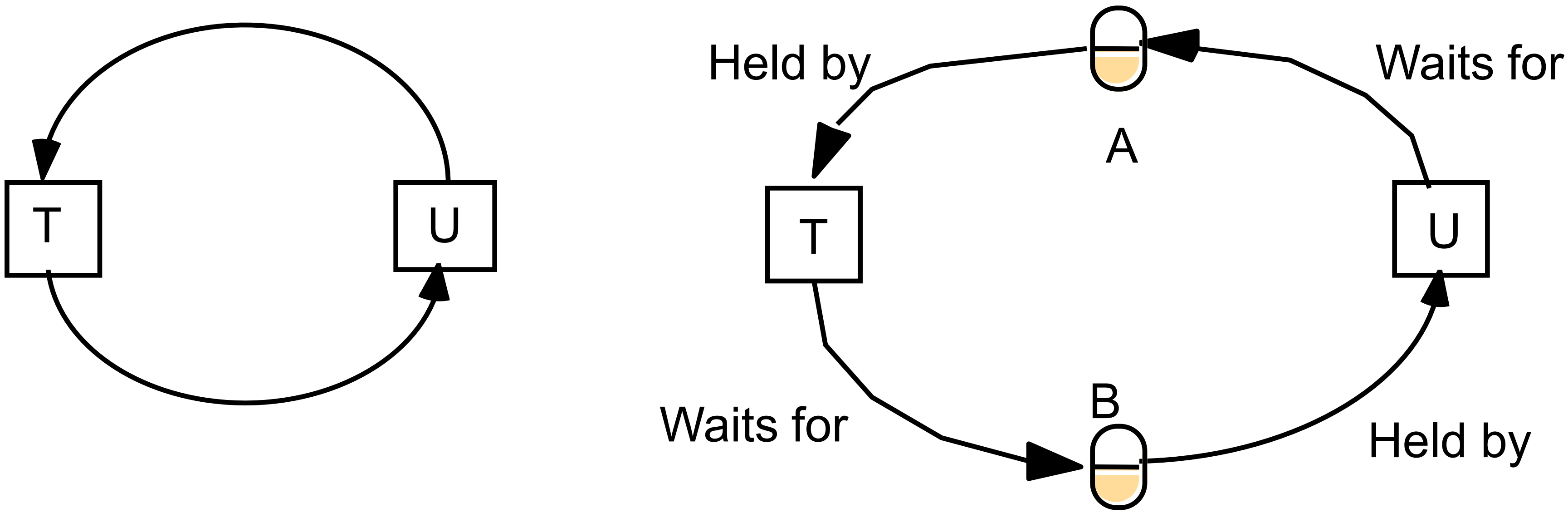


Figure 16.21
A cycle in a wait-for graph

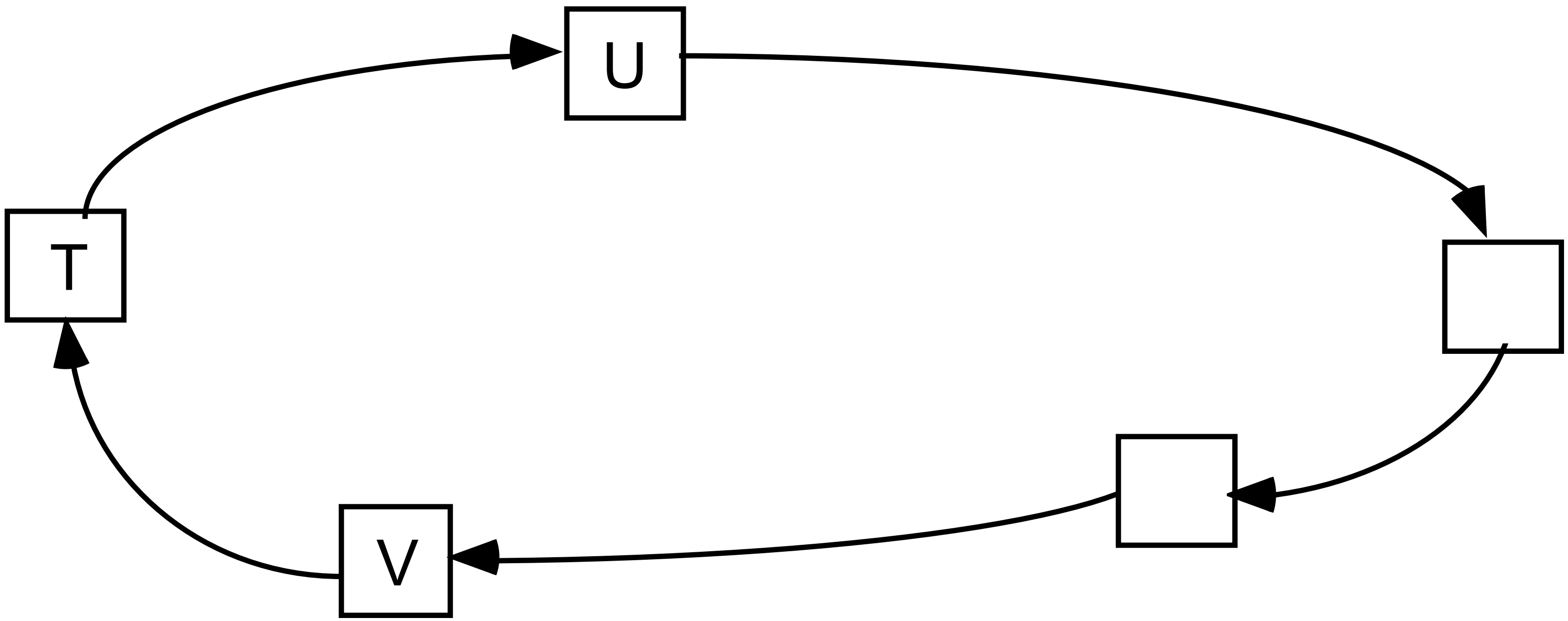


Figure 16.22
Another wait-for graph

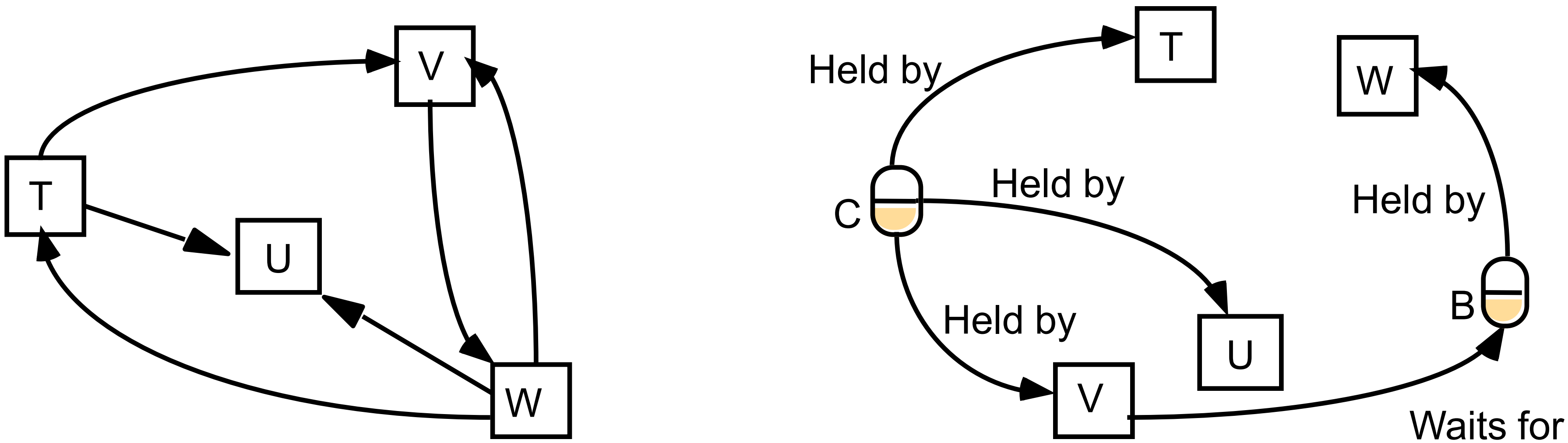


Figure 16.23
Resolution of the deadlock in Figure 15.19

Transaction T		Transaction U	
Operations	**Locks**	**Operations**	**Locks**
a.deposit(100);	write lock *A*		
		b.deposit(200)	write lock *B*
b.withdraw(100)			
• • •	waits for *U*'s lock on *B*	*a.withdraw(200);*	waits for T's lock on *A*
	(timeout elapses)	• • •	
T's lock on *A* becomes vulnerable,	unlock *A*, abort T	• • •	
		a.withdraw(200);	write locks *A* unlock *A*, *B*

Figure 16.24
Lock compatibility (*read*, *write* and *commit* locks)

For one object		*Lock to be set*		
		read	*write*	*commit*
Lock already set	*none*	OK	OK	OK
	read	OK	OK	wait
	write	OK	wait	
	commit	wait	wait	

Figure 16.25
Lock hierarchy for the banking example

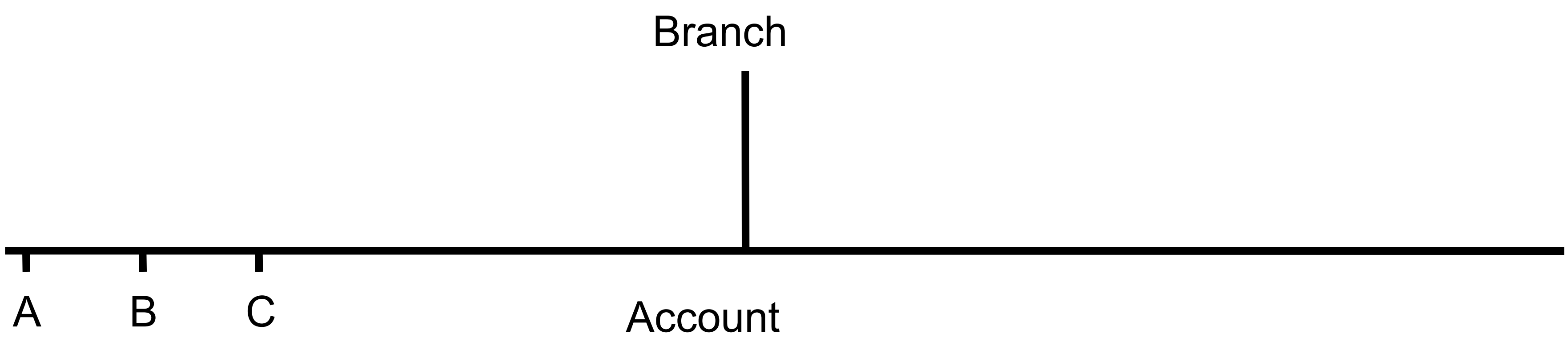


Figure 16.26
Lock hierarchy for a diary

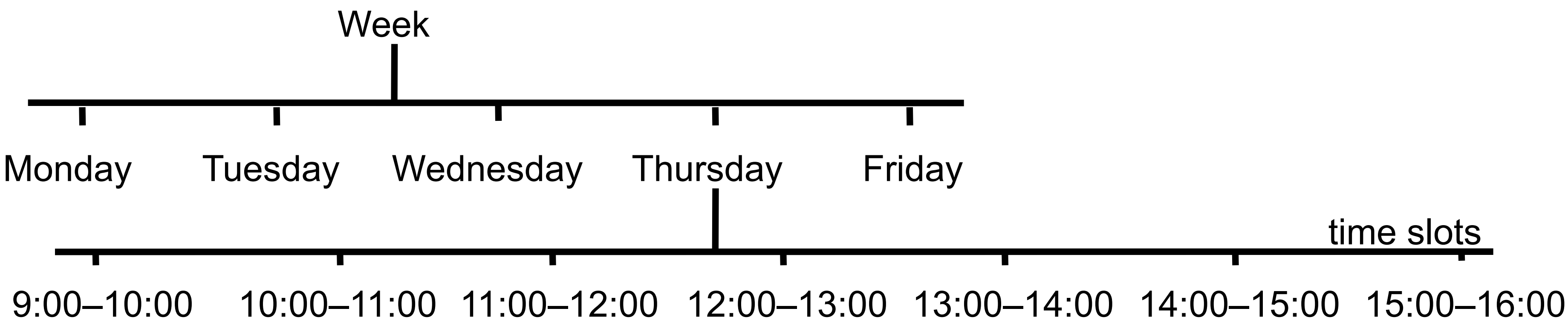


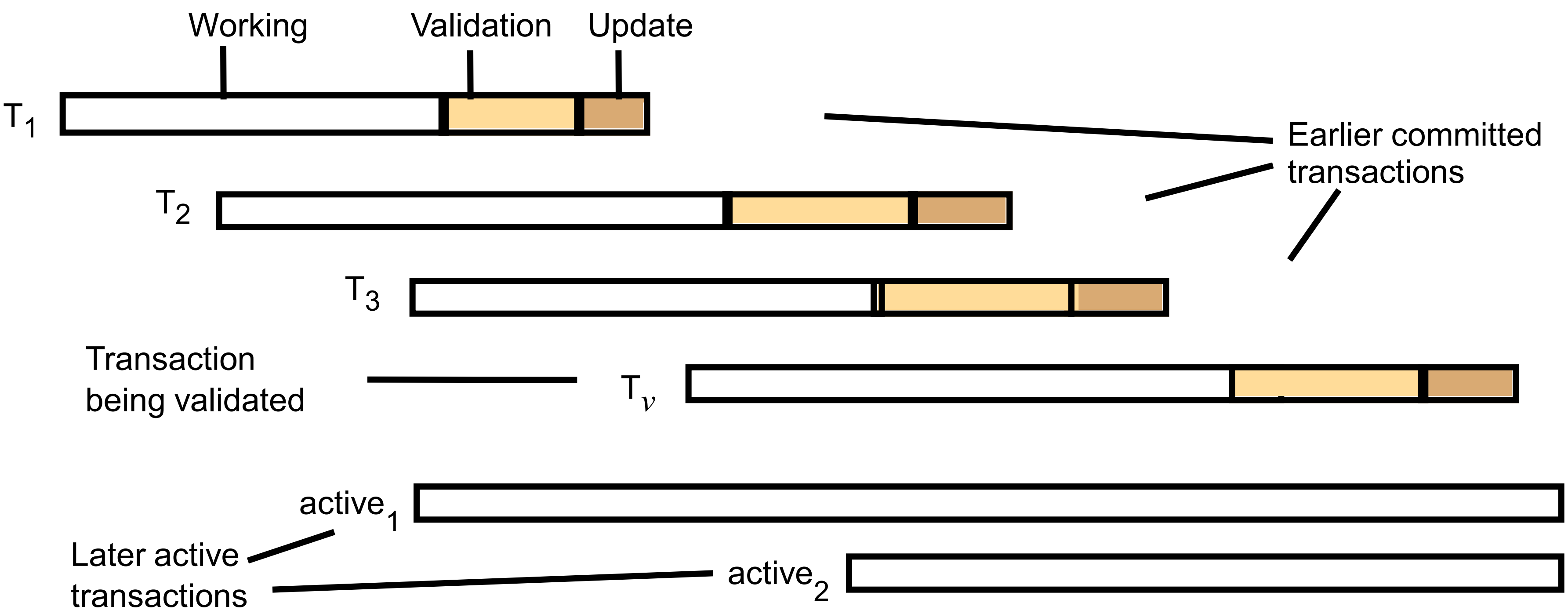
Figure 16.27
Lock compatibility table for hierarchic locks

For one object		*Lock to be set*			
		read	*write*	*I-read*	*I-write*
Lock already set	*none*	OK	OK	OK	OK
	read	OK	wait	OK	wait
	write	wait	wait	wait	wait
	I-read	OK	wait	OK	OK
	I-write	wait	wait	OK	OK

Table on page 708
Serializability of transaction *T* with respect to transaction T_i

T_v	T_i	*Rule*	
write	*read*	1.	T_i must not read objects written by T_v
read	*write*	2.	T_v must not read objects written by T_i
write	*write*	3.	T_i must not write objects written by T_v and T_v must not write objects written by T_i

Figure 16.28
Validation of transactions



Validation of Transactions

Backward validation of transaction T_v

```
boolean valid = true;
for (int Ti = startTn+1; Ti <= finishTn; Ti++){
    if (read set of Tv intersects write set of Ti) valid = false;
}
```

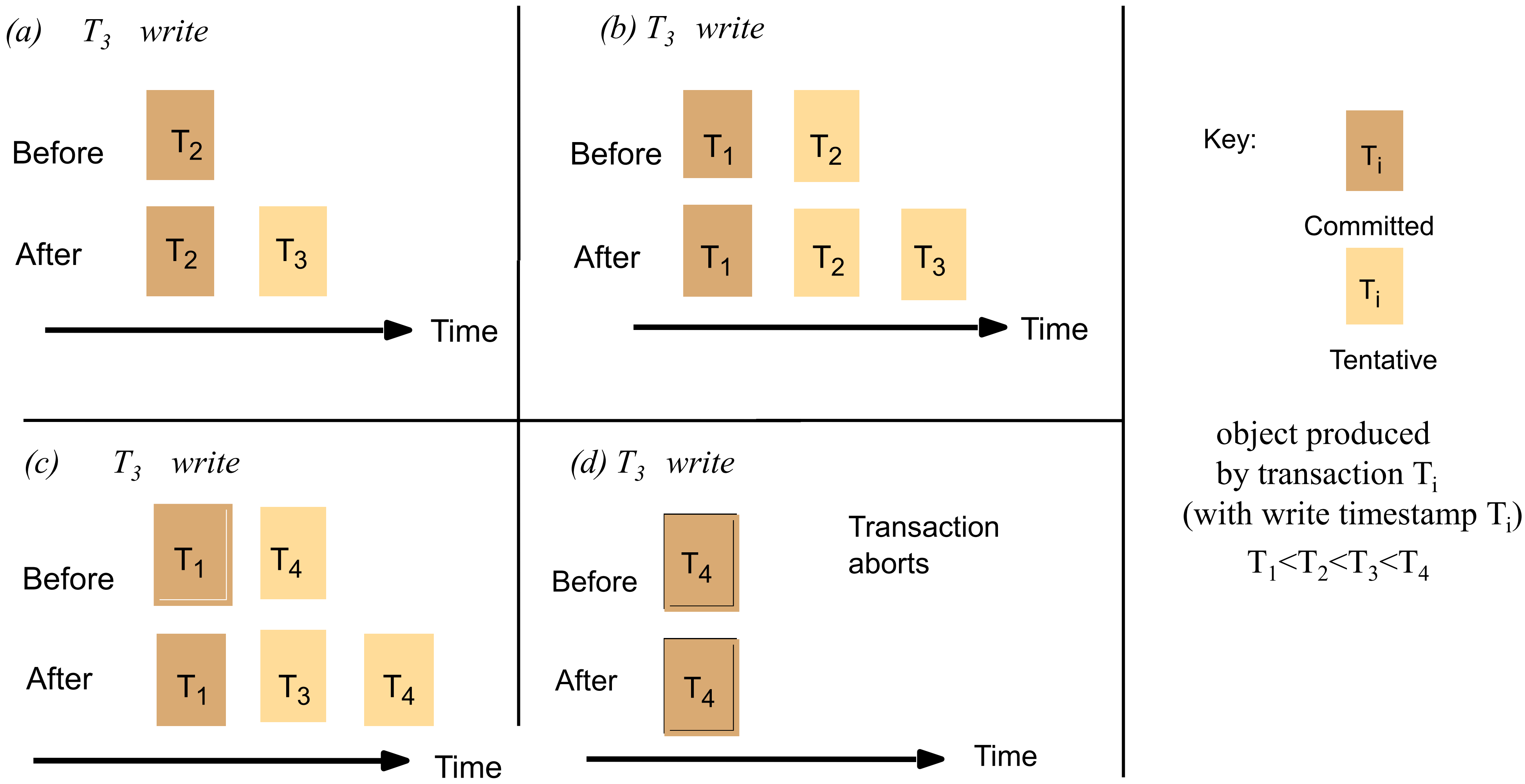
Forward validation of transaction T_v

```
boolean valid = true;
for (int Tid = active1; Tid <= activeN; Tid++){
    if (write set of Tv intersects read set of Tid) valid = false;
}
```

Figure 16.29
Operation conflicts for timestamp ordering

Rule	T_c	T_i	
1.	*write*	*read*	T_c must not *write* an object that has been *read* by any T_i where $T_i > T_c$ this requires that $T_c \geq$ the maximum read timestamp of the object.
2.	*write*	*write*	T_c must not *write* an object that has been *written* by any T_i where $T_i > T_c$ this requires that $T_c >$ write timestamp of the committed object.
3.	*read*	*write*	T_c must not *read* an object that has been *written* by any T_i where $T_i > T_c$ this requires that $T_c >$ write timestamp of the committed object.

Figure 16.30
Write operations and timestamps



Page 713
Timestamp ordering write rule

if ($T_c \geq$ maximum read timestamp on D &&
    $T_c >$ write timestamp on committed version of D)
        perform write operation on tentative version of D with write timestamp T_c
else /* write is too late */
    Abort transaction T_c

Page 714
Timestamp ordering read rule

if (T_c > write timestamp on committed version of *D*) {

let $D_{selected}$ be the version of *D* with the maximum write timestamp ≤ T_c

if ($D_{selected}$ is committed)

perform *read* operation on the version $D_{selected}$

else

Wait until the transaction that made version $D_{selected}$ commits or aborts

then reapply the *read* rule

} else

Abort transaction T_c

Figure 16.31
Read operations and timestamps

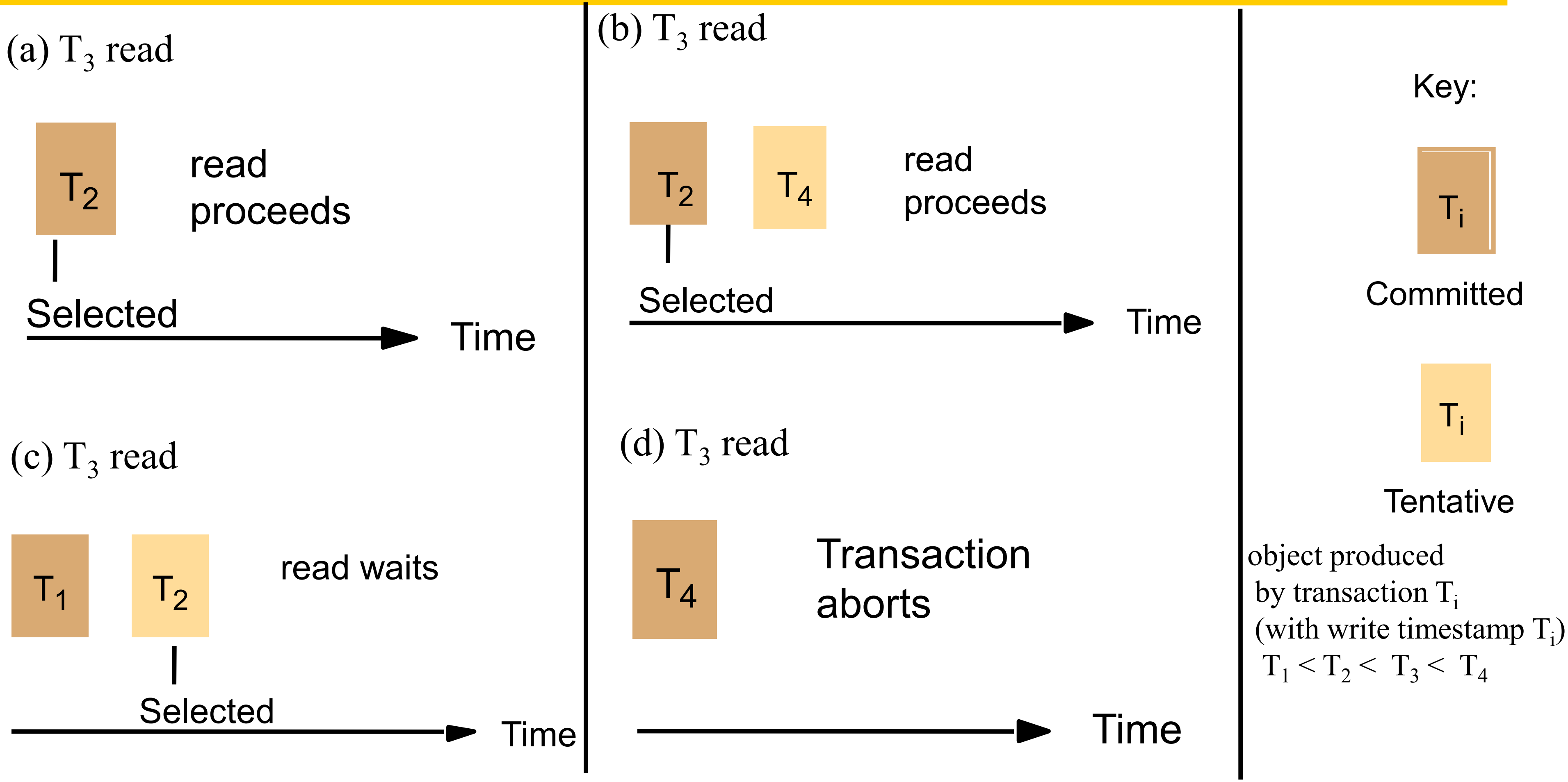


Figure 16.32
Timestamps in transactions *T* and *U*

		Timestamps and versions of objects					
T	*U*	*A*		*B*		*C*	
		RTS	*WTS*	*RTS*	*WTS*	*RTS*	*WTS*
		{}	***S***	{}	***S***	{}	***S***
openTransaction							
bal = b.getBalance()				{*T*}			
	openTransaction						
*b.setBalance(bal*1.1)*					***S***, *T*		
	bal = b.getBalance()						
	wait for T						
a.withdraw(bal/10)	● ● ●		***S***, *T*				
commit	● ● ●		***T***		***T***		
	bal = b.getBalance()			{*U*}			
	*b.setBalance(bal*1.1)*				***T***, *U*		
	c.withdraw(bal/10)						***S***, *U*

Figure 16.33
Late *write* operation would invalidate a *read*