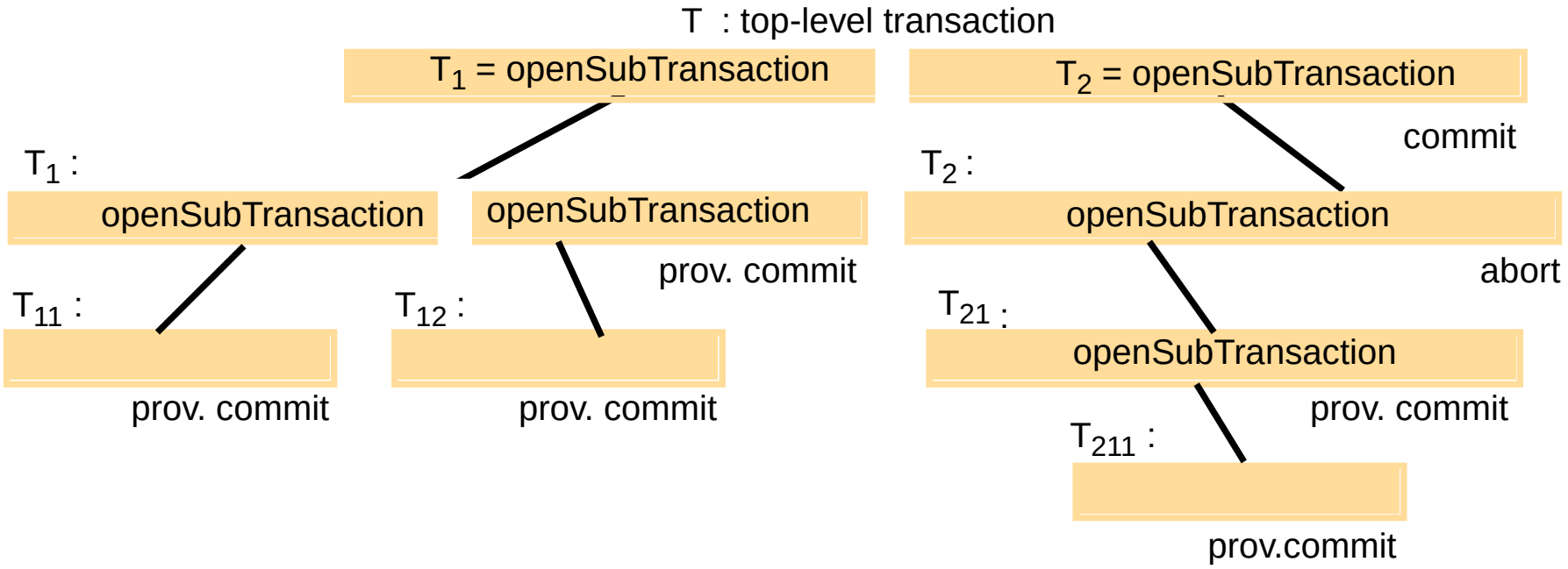
Today’s topics

- Nested transactions
- Expectations on final reports
- Locks

Nested transactions



- transactions may be composed of other transactions
  - several transactions may be started from within a transaction
  - we have a top-level transaction and subtransactions which may have their own subtransactions

Nested transactions

- To a parent, a subtransaction is atomic with respect to failures and concurrent access
- transactions at the same level (e.g. *T*1 and *T*2) can run concurrently but access to common objects is serialised
- a subtransaction can fail independently of its parent and other subtransactions
  - when it aborts, its parent decides what to do, e.g. start another subtransaction or give up
- The CORBA transaction service supports both flat and nested transactions

Advantages of nested transactions (over *flat* ones)

- Subtransactions may run concurrently with other subtransactions at the same level.
  - this allows additional concurrency in a transaction.
  - when subtransactions run in different servers, they can work in parallel.
    - e.g. consider the *branchTotal* operation
    - it can be implemented by invoking *getBalance* at every account in the branch.
      - these can be done in parallel when the branches have different servers
- Subtransactions can commit or abort independently.
  - this is potentially more robust Why?
  - a parent can decide on different actions according to whether a subtransaction has aborted or not

With a flat transaction, one transaction failure would cause the whole transaction to be restarted. Based on the example of delivering mail, we can see that a set of nested transactions is potentially more robust.

Commitment of nested transactions

- A transaction may commit or abort only after its child transactions have completed.
- A subtransaction decides independently to *commit provisionally* or to *abort*. Its decision to abort is final.
- When a parent aborts, all of its subtransactions are aborted.
- When a subtransaction aborts, the parent can decide whether to abort or not.
- If the top-level transaction commits, then all of the subtransactions that have provisionally committed can commit too, provided that none of their ancestors has aborted.

Summary on transactions

- We consider only transactions at a single server, they are:
- atomic in the presence of concurrent transactions
  - which can be achieved by serially equivalent executions
- atomic in the presence of server crashes
  - they save committed state in permanent storage
  - they use **strict executions** to allow for aborts
  - they use tentative versions to allow for commit/abort
- nested transactions are structured from sub-transactions
  - they allow concurrent execution of sub-transactions
  - they allow independent recovery of sub-transactions

What I’m expecting in your final report (1)

- 4~6 pages in the IEEE 2-column format
- Please do use the IEEE template DOC file
- The number of references should be more than 10
- Source code of your program should not be included in the 4~6 pages

What I’m expecting in your final report (2)

- I will focus on your real work and efforts
- Next I will look at your results (How good they are? Is it a new idea?)
- Survey papers cannot receive high scores
- If you want to compare some existing algorithms or designs, at least you need to design and conduct experiments to support your conclusions

What I’m expecting in your final report (3)

- No or few obvious grammar errors and typos.
- Correct format (e.g., italic for variables and Abstract).
- Sufficient experimental results in figures/tables.
- Source code can only be attached as Appendix of your final report.
- If majority parts of your report come from an existing article, you will receive zero point.
- Send me a separate document, which clearly states each person’s contributions if there are two students in one group.

Can you recall the definition of serial equivalence?

- Transactions must be scheduled so that their effect on shared objects is serially equivalent

- A server can achieve serial equivalence by serialising access to objects, e.g. by the use of locks

for serial equivalence,

(a) all access by a transaction to a particular object must be serialized with respect to another transaction's access.

(b) all pairs of conflicting operations of two transactions should be executed in the same order.

to ensure (b), a transaction is not allowed any new locks after it has released a lock

- *Two-phase locking* - has a 'growing' and a 'shrinking' phase

the use of the lock on *B* effectively serialises access to *B*

when *T* is about to use *B*, it is locked for *T*

when *U* is about to use *B*, it is still locked for *T* and *U* waits

U can now continue

when *T* commits, it unlocks *B*

Transaction *T*:		**Tr**	
balance = b.getBalance()		*ba*	
*b.setBalance(bal*1.1)*		*b.setBalance(bal*1.1)*	
a.withdraw(bal/10)		*c.withdraw(bal/10)*	
Operations	Locks	Operations	Locks
openTransaction			
bal = b.getBalance()	lock *B*		
		openTransaction	
*b.setBalance(bal*1.1)*			
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*

- initially the balances of A, B and C unlocked

What are dirty reads?

Strict two-phase locking

How can they be prevented?

- strict executions prevent dirty reads and premature writes (if transactions abort).

Dirty read - a transaction reads a value set by another transaction that subsequently aborts

Prevent dirty reads by delaying reads until the transaction that wrote the object they want to read has committed or aborted.

What decides whether a pair of operations conflict?

- concurrency control protocols are designed to deal with *conflicts* between operations in different transactions on the same object
- we describe the protocols in terms of *read* and *write* operations, which we assume are atomic
- read operations of different transactions do not conflict

- It uses read locks and write locks

A pair of operations conflict if their combined effect depends on the order in which they were executed

Lock compatibility

to enforce 1, a request for a write lock is delayed by the presence of a read lock belonging to another transaction

to enforce 2, a request for a read lock or write lock is delayed by the presence of a write lock belonging to another transaction

- The operation conflict rules tell us that:
  1. If a transaction *T* has already performed a *read* operation on a particular object, then a concurrent transaction *U* must not *write* that object until it commits or aborts.
  2. If a transaction *T* has already performed a *write* operation on a particular object, then a concurrent transaction *U* must not *read* or *write* that object until *T* commits or aborts.

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

Lock promotion

Why not allow demotion of locks?

- Lost updates – two transactions read an object and then use it to calculate a new value (remember 10% increase example?).
- Lost updates are prevented by making later transactions delay their reads until the earlier ones have completed.
- each transaction sets a read lock when it reads and then promotes it to a write lock when it writes the same object
- when another transaction requires a read lock it will be delayed (can anyone see a potential danger which does not exist when exclusive locks are used?)
- Lock promotion: the conversion of a lock to a

because the new weaker lock may allow executions by other transactions that are inconsistent with serial equivalence.

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.

- The sever applies locks when the read/write operations are about to be executed
- the server releases a transaction's locks when it commits or aborts

Lock implementation

- The granting of locks will be implemented by a separate object in the server that we call the *lock manager*.
- the lock manager holds a set of locks, for example in a hash table.
- each lock is an instance of the class *Lock* (Fig 16.17) and is associated with a particular object.
  - its variables refer to the object, the holder(s) of the lock and its type
- the lock manager code uses *wait* (when an object is locked) and *notify* when the lock is released
- the lock manager provides *setLock* and *unLock* operations for use by the server

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

continued

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

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);
      }
   }
}
```

What can a lock manager do about deadlocks?

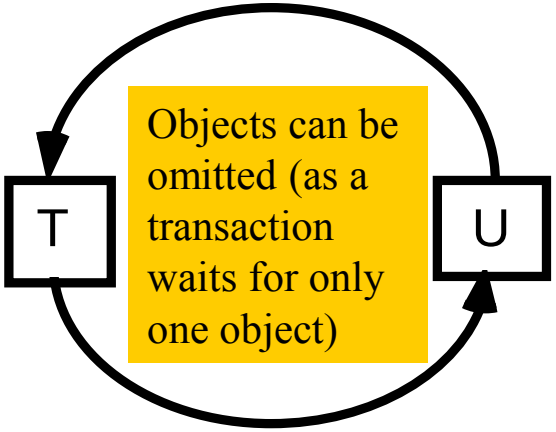
T accesses A → B
U accesses B → A

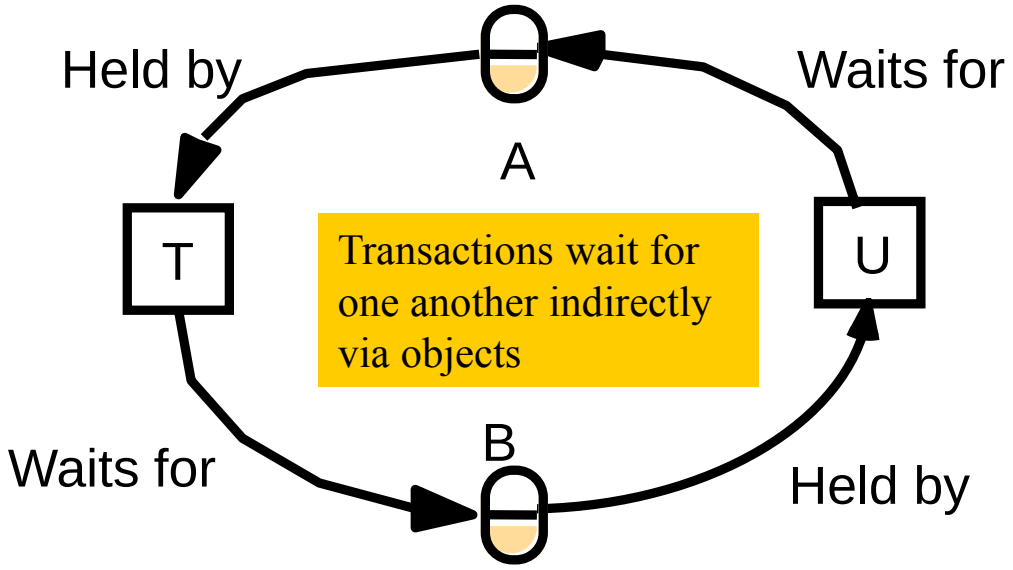
Transaction *T*		Transaction *U*	
Operations	**Locks**	**Operations**	**Locks**
a.deposit(100);	write lock *A*		
		b.deposit(200)	write lock *B*
b.withdraw(100)			
•••	waits for *U*'s	*a.withdraw(200);*	waits for *T*'s
	lock on *B*	•••	lock on *A*
•••		•••	
•••		•••	

When locks are used, each of T and U acquires a lock on one account and then gets blocked when it tries to access the account the other one has locked.
We have a 'deadlock'.
The lock manager must be designed to deal with deadlocks.

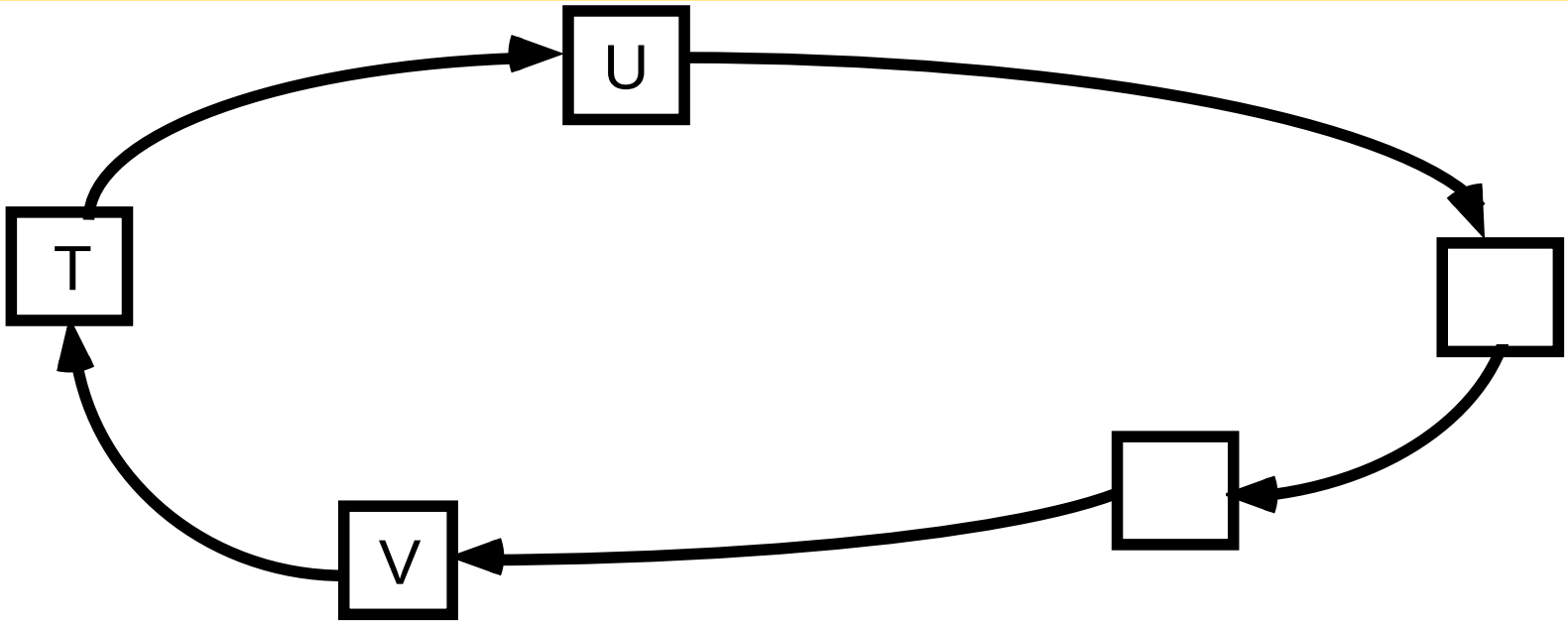
In a wait-for graph the nodes represent transactions and the edges represent wait-for relationships between transactions

- Definition of deadlock
  - deadlock is a state in which each member of a group of transactions is waiting for some other member to release a lock.
  - a *wait-for graph* can be used to represent the waiting relationships between current transactions



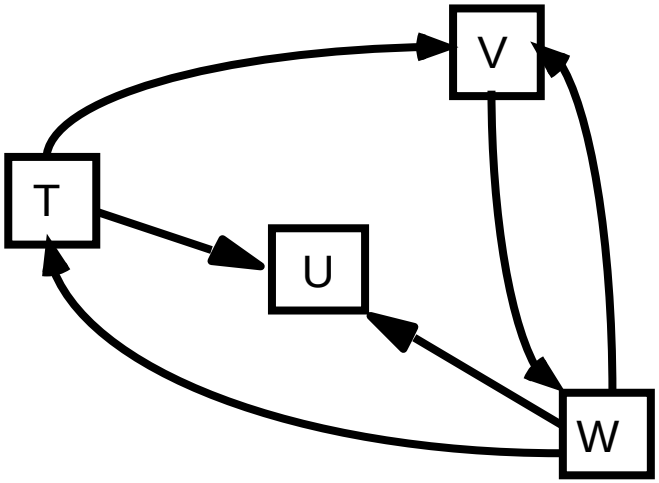


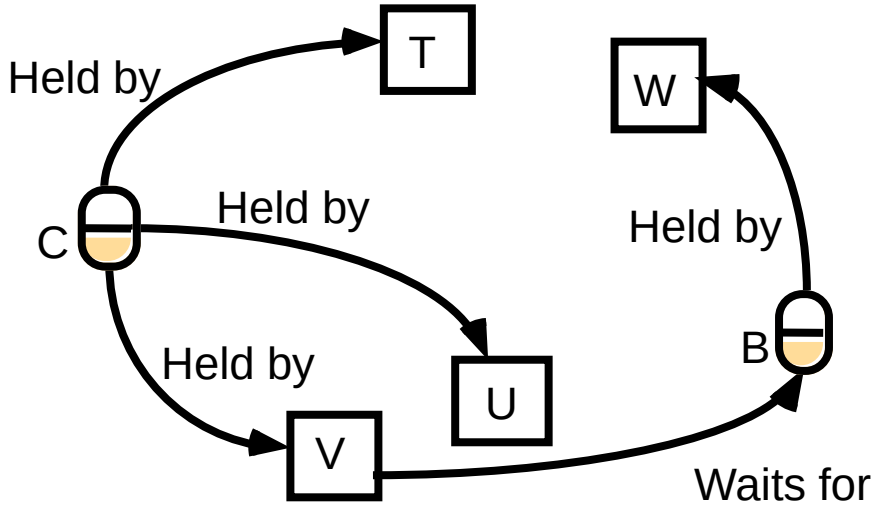
A cycle in a wait-for graph



- Suppose a wait-for graph contains a cycle T …→ U → … → V → T
  - each transaction waits for the next transaction in the cycle
  - all of these transactions are blocked waiting for locks
  - none of the locks can ever be released (the transactions are deadlocked)
  - If one transaction is aborted, then its locks are released and that cycle is broken

Another wait-for graph





- *T*, *U* and *V* share a read lock on *C* and
- *W* holds write lock on *B* (which *V* is waiting for)
- *T* and *W* then request write locks on *C* and deadlock occurs e.g. *V* is in two cycles - look on the left

Deadlock prevention is unrealistic

- e.g. lock all of the objects used by a transaction when it starts
  - unnecessarily restricts access to shared resources.
  - it is sometimes impossible to predict at the start of a transaction which objects will be used.
- Deadlock can also be prevented by requesting locks on objects in a predefined order
  - but this can result in premature locking and a reduction in concurrency

Deadlock detection

- by finding cycles in the wait-for graph.
  - after detecting a deadlock, a transaction must be selected to be aborted to break the cycle
  - the software for deadlock detection can be part of the lock manager
  - it holds a representation of the wait-for graph so that it can check it for cycles from time to time
  - edges are added to the graph and removed from the graph by the lock manager's *setLock* and *unLock* operations
  - when a cycle is detected, choose a transaction to be aborted and then remove from the graph all the edges belonging to it
  - it is hard to choose a victim - e.g. choose the oldest or the one in the most cycles

What are the problems with lock timeouts?

Timeouts on locks

- Lock timeouts can be used to resolve deadlocks
  - each lock is given a limited period in which it is invulnerable.
  - after this time, a lock becomes vulnerable.
  - provided that no other transaction is competing for the locked object, the vulnerable lock is allowed to remain.
  - but if any other transaction is waiting to access the object protected by a vulnerable lock, the lock is broken
    - (that is, the object is unlocked) and the waiting transaction resumes.
  - The transaction whose lock has been broken is normally aborted

- problems with lock timeouts
- locks may be broken when there is no deadlock
- if the system is overloaded, lock timeouts will happen more often and long transactions will be penalised
- it is hard to select a suitable length for a timeout

Resolution of the deadlock in Figure 16.19

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*

Drawbacks of locking

- Lock maintenance costs an overhead.
- The use of locks can result in deadlock.
- To avoid cascading aborts, locks cannot be released until the end of the transaction, which may reduce significantly the potential of concurrency.

Assignment#3 (Chapter 16)

- 16.2
- 16.3
- 16.17