April 14 2014 Exam

Q1

1. $\frac{1}{3}$

2.

3. $f_{2,2}$

Q2: client code using the specified operations

```
lock (Server)
    do stuff
release (Server)
```

Client code for lock & release.

```
lock (S) ->
    S ! {self(), lock}
    receive
        ok -> ok
    end.

release (S) ->
    S !                 unlock
    receive
        ok -> ok
    end.
```

Server code

```
loop (Locked) ->
    receive

    {C, lock} when not Locked ->
        C ! ok, loop (true)

    {C, unlock} ->
        C ! ok, loop (false)

end
```

5.a. State Msg) {NewState, RetMsg}

int Logic (X , getState) → { x , X }

int Logic (X , {setState, Y}) → { Y , ok }

5b

Server:rpc(IntServer, {setState, server:rpc(IntServer, setState)+1})

5c race condition

5d have an increment in the intLogic code

intLogic (X, increment) → {X+1, ok}

5e pass a function to intLogic and have it performed atomically by the server

6. The problem is that the generic server (prob 5)

replies to client requests in the order they are placed in the mailbox.

The lock server has to delay replying to a lock op. when the lock is already held until the release is placed in the mailbox

Project questions

Channel Stuff

if you send from different Threads

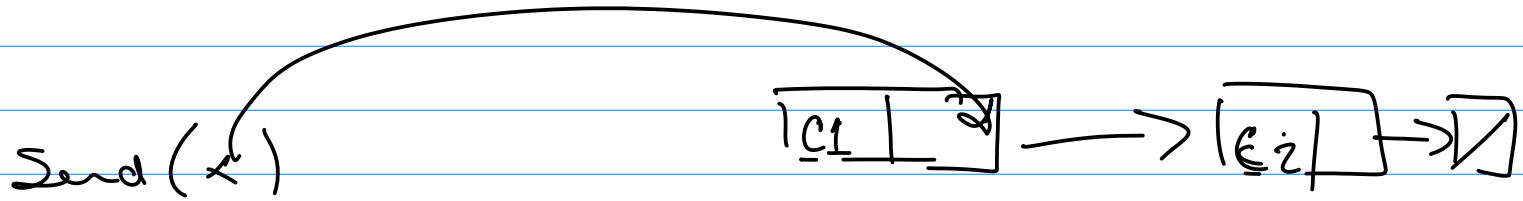
```
send (c1, "a")
send (c1, "b")
send (c1, "c")
send (c1, "d")
```

and then you have (in yet another thread)

```
v ← recv (c1)
```

does v have to be "a"? No.

v1 ← recv(c1) // v2 ← recv(c1)



Transactions — another way of thinking about concurrency.

Transactions were invented to deal with problems of concurrent access to data bases in the early 1970s.

4 things that transactions should do for you:
"The ACID Properties" of transactions

A: Atomicity — either, the whole thing happens or nothing happens.

C: Consistency — every transaction transforms a valid state to a valid state.

different perspectives:

1) if every transaction run alone takes valid states to valid states
Then the DB is always in a valid state.

2) transaction implementers must arrange that if a Xact starts in a valid state it ends in a valid state

I: Isolation – transactions only see (read data from) completed transactions

D: Durability – once a transaction has successfully completed its effects will never disappear from the database.

So, structuring transactions:

```
T = beginTransaction()
    operations such as reading, writing, locking etc passing
      in T to each one so the DB system knows
      which transaction every operation belongs to.
    :
    endTransaction(T, abort)  → leave the DB as if this transaction never existed
                      commit) → make changes permanent
```

Multiple approaches to achieving this:

1) locks and rules about locking

2) optimistic concurrency control - system records xact's reads and writes and decides whether they are "compatible" w/ other concurrent xact's reads/writes.

⇒ res ← endTransaction (T, commit)

↓

res is either committed or aborted