These are sample assignments in my Computer Science doctoral program.

Unit 1 Project

Concurrent and Distributed Systems

Eric W. Wasiolek

Professor: Dr. Bo Sanden

Date: January 18, 2016

Part 1: Describe semaphores, test-and-set, reentrancy, and preemption in your own words.

Semaphores. A semaphore is a variable the typically has two values: open and closed (typically represented by a zero or one bit). This is a binary semaphore; a counting semaphore can have more than two states. The semaphore works as a lock on resources that are shared between threads like a safe object. A thread can acquire a lock via the semaphore if the semaphore is open. It then closes it and the thread has exclusive access to the resource locked by the semaphore until it releases it (opens it). Once the thread has acquired the lock it executes its code (critical section --- a code section that only one thread at a time executes, this is different from reentrant code which multiple threads may execute simultaneously).

Test and Set. Test and set is a hardware instruction that operates when a thread acquires a semaphore (Sanden, 2011). If it is open (zero), test and set sets it to one (closed). This happens automatically. This is what happens underneath when a thread attempt to acquire a lock on a safe object (enabling exclusion synchronization, or mutual exclusion between threads accessing a common safe object). Test and set is sometimes called a lock.

Re-entrancy. Although this was excluded from the assignment, it means code that is shared between threads and can be executed by many threads simultaneously.

Pre-emption. One thread can pre-empt another thread in its access to a resource like a processor if it has higher priority. Every thread has a priority as part of its context. Priority can be assigned with the setPriority() method in Java. In preemptive scheduling a higher priority thread preempts a lower priority thread. A thread keeps a processor until it releases it voluntarily.

Part 2: Describe priority inversion, push-through blocking (or push-through stalling), priority inheritance, and priority ceiling in your own words.

This part was not asked for in this assignment according to the notes at the bottom of the assignment.

Part 3: When distributed systems are discussed later in the course, you will encounter a safe object called Maekawa. (The PowerPoint set for Distributed Mutual Exclusion has an Ada version with a protected object Maekawa.)

Here is a simplified version of Maekawa as a Java class Msynch. The method `acquire ( )` lets one calling thread send 5 messages and wait for 5 replies. For each reply, a communication thread calls `replyReceived ( )`. Ultimately, the first thread calls `release ( )`.

- Describe briefly what happens in Msynch when the first thread calls `acquire ( )` the first time, and each time *another* thread calls `replyReceived ( )`.

My answer to this question is given by the comments embedded in the code below.

```
class Msynch

{

Comment.  Mysynch is a safe object; any class with synchronized
methods is a safe object.  Not shown here is the Thread class
and its instantiated objects which are the threads that call
methods in this safe object.

     int replies;

     int currentState = 1;

     synchronized void acquire ( )

     { // Called by thread wanting access to a critical section



Comment:  When acquire() is called, as with any synchronized
method that is called it acquires a lock on the safe object.

          while (currentState != 1) wait ( );

Comment:  When the condition becomes true and wait() is invoked,
the thread releases its lock and goes into the safe object's
wait set.  When the thread is reactivated, it retests the
condition (as long as while is used rather than if).  This keeps
happening until the condition becomes false.

          replies = 0; currentState = 2;

          //

          // (Here, 5 messages are sent)
```

```
        //

        while (replies < 5) wait ( ); // Await 5 replies
```

Comment: The thread keeps testing the reply counter until it is incremented to five. This is another wait() which causes the thread to release its lock and goes into the safe objects wait set.

```
        currentState = 3;

    }  // end acquire()
```

Comment: When acquire() ends, it releases its lock on the safe object.

```
    synchronized void replyReceived ( )

    { // Called by communication thread when reply is received
```

Comments: A separate thread is activated when a reply is received and the thread requests a lock on the safe object, which it eventually gets and locks the safe object. The thread receives a reply and increments the replies counter.

The thread then calls notifyAll(). notifyAll() releases all threads waiting for a lock on the safe object. The highest priority thread gets a lock on the safe object.

Each time a thread calls replyReceived the actions above are repeated. The replies counter will eventually be five, which will cause the statement "while (replies < 5) wait();" not to execute.

```
        replies++;

        notifyAll ( );

    }  // end replyReceived()



    synchronized void release ( )

    { // Called by a thread releasing the critical section

        currentState = 1;
```

```
            notifyAll ( );

     }  // end release()

}  // end class Msynch
```

- **Find 3 synchronization related errors in Msynch1. (This can be done through line-by-line comparison.) Explain why each error will cause Msynch1 not to work.**

```
class Msynch1

{

     // private variables

     int replies;

     int currentState = 1;



     synchronized void acquire ( )

     { // Called by thread wanting access to a critical section

          while (currentState != 1) yield ( );
```

Error Number One: Yield() should not be used here. Wait() should be used instead. Yield() causes the thread to give up the processor. This is not what you want and will result in improper synchronization. Also, yield() is not normally called from within a synchronized method.

```
          replies = 0;

          currentState = 2;

          //

          // (Here, 5 messages are sent)

          //
```

```
        if (replies < 5) wait ( ); // Await 5 replies
```

Error Number Two. "If" should not be used here. Instead this should be a while loop. "If" only tests a condition once, whereas while will test the condition repeatedly until the condition becomes false (such as replies equally five). Five messages are sent and five replies are waited for. Therefore you need to wait until replies (the variable that is incremented by the thread calling replyReceived) equals five, and check the condition repeatedly until this is so.

```
        currentState = 3;

    }  // end acquire()



    synchronized void replyReceived ( )

    { // Called by communication thread when reply is received

        replies++;
```

Error Number Three. There needs to be a notifyAll() function here. NotifyAll() unblocks all threads in Msynch1's wait set. If this is not called, other threads will remain blocked.

```
    }  // end replyReceived()

    synchronized void release ( )

    { // Called by a thread releasing the critical section

        currentState = 1;

        notifyAll ( );

    }  // end release()


}  // end Mysynch1 class
```

Unit 2 Project

Concurrent and Distributed Systems

Eric W. Wasiolek

Professor: Bo Sanden

Date: February 1, 2016

Part 1: Compare and contrast the way a protected function in Ada works versus how a nonsynchronized method in a Java class works when the class has other methods that are synchronized.

A protected function in ADA is somewhat like a non-synchronized method in Java in that neither acquires a write lock on the safe object. A function does however acquire a read lock on the object. Functions are read only in that multiple functions can execute on a protected object, reading it at the same time, but not if a protected procedure or entry has a write lock on the object (in which case the function will fail to get a read lock on the object).

In Java, every synchronized method acquires a write lock on the safe object it is accessing (usually done intrinsically by a semaphore). As far as I can tell a non-synchronized method doesn't acquire any sort of lock at all, not even a read lock. So, it doesn't act as an ADA function does. This may mean that a non-synchronized method in Java may allow dirty reads, i.e., reading data that is being changed by another method. This can cause problems. Also, if you want a read lock in Java, it looks like you have to implement it with a synchronized method.

Part 2: Compare and contrast Ada's requeue statement and the corresponding solution in Java.

Part 2 is deleted according to your instructions.

Part 3: In a simple concurrent system, one instance of the task type Worker processes data elements from a queue implemented as the protected object Queue. (Some other task puts the elements there by calling Enqueue.)

- **Describe briefly what Worker does. (This is pretty much spelled out in the comments.)**

The Worker task tries to get an element from the head of the queue. It succeeds if there is an element in the queue. It does this with the "elt: Queue.First_in_line" statement which calls the routine First_in_line from the Queue protected object (safe object) which is managing the queue that tasks try to access. The queue is a FIFO data structure (first in first out). First_in_line is like a typical front() operation which access the item at the front of the queue. Items are pushed into the back of the queue (this would be done by a task calling the Queue.Enqueue protected procedure).

If there is no element in the queue, the Worker tasks delays for a tenth of a second and retries accessing the queue.

When there is an element in the queue, Worker accesses it and then processes it. The code to process the queue element is not shown.

Once the Worker task is done processing the element from the front of the queue it removes it from the head of the queue (this is typically done with a remove() routine, or a pop() routine which both accesses the element at the head of the queue and removes it). This all occurs in a "while true" loop which seems to me, even though I don't know ADA, is like an infinite loop (while(true) in C++ is an infinite loop). This would seem to indicate that the Worker would loop back and try to access, process, and remove another element. The Worker task will later be modified (below) so that it access, processes, and removes only one element.

- **Adapt Queue and Worker so that multiple instances of Worker can process data elements from the queue. Exactly one Worker must process each element. What has to be changed in Queue and in Worker to allow this?**
- **The loop where instances of Worker wait for something in the queue is somewhat crude. How could Queue (and Worker) be changed to do this without such "busy waiting"?**

I am going to indicate my code modifications to both the safe object and the task in one fell swoop, and then explain how I answer both questions.

In replacing the delay statement in the Worker task, which is inefficient, I would use condition synchronization instead, transforming the First_in_line function with a First_in_line entry. The condition would be when the count is greater than zero, i.e., something is in the queue. The modifications to the code would look like this:

1. In the protected object:

a. Replace function First_in_line with entry First_in_line and add the condition when count is greater than zero, i.e., the queue is not empty.

entry First_in_line when count > 0.

b. You would have to add a count variable to the protected object. Count would indicate the number of elements in the queue.

c. You could have to add a function getCount, which gets the count of the number of elements in the queue.

d. In the body of the protected procedure for Enqueue it would have to increase the count by 1 when called, when it inserts an element into the queue.

e. In the body of the protected procedure for Dequeue it would have to decrease the count by 1 when called, i.e., when a task calls it and removes an element from the head of the queue.

2. In the Worker task

You don't need the delay statement anymore in the Worker task, since First_in_line which it calls is an entry instead of a function. When Worker calls it the task waits (does not block other tasks) until the barrier condition is true. Then it reads and processes the element and exits.

The Worker task code would look like this:

begin

elt := Queue.First_in_line

--- will now execute only when there is an element in the queue, as the First_in_line

function has been transformed into an entry with a condition that the queue is not empty.

The task will now wait, not blocking other tasks, until the barrier condition is true.

Process(elt)

--- The code to process the element is not shown

Queue.Dequeue

--- Removes the element that it accessed from the front of the queue so now the next task

will access the next element.

end

Notice that the if ... else statement has been removed as the delay statement is no longer needed. Also notice the while loop has been removed, to enable the other part of this assignment, that the Worker task access, processes, and removes only one element and then exits. It does not loop and access more elements. "While true" is an infinite loop. If one leaves that in the task code the task will endlessly access elements from the queue which is not what is asked for in this assignment. By removing the while true loop the condition you set forth that each Worker task accesses exactly only one element is the case. When the next task calls the protected object queue, it will do the same and access, process, and remove only one element.

In case I misunderstood the statement "exactly one Worker must process each element," which I took to mean each Worker accesses, processes, and removes one element, but perhaps you didn't mean this, each Worker can process more than one element if you stick the above inside a loop. In any case, an infinite loop like "while(true)" should probably not be used, and instead a loop where each Worker processes say ten elements may be made by sticking the above inside a loop statement like (repeat 10, or for(i = 0, i < 10; ++i), or i = 10, while(i > 0), --i< or whatever the syntax is in ADA.

The idea with each task processing one element is that you could get many tasks working on the queue simultaneously running on different processors to increase concurrency and speed. While one task is processing an element, another task may be accessing the next element.

3. Other Notes

a. To enable multiple tasks to execute, the statement typically used would be:

task type Worker

This creates a task type and allows multiple tasks to be created dynamically.

b. You would have to have some Worker tasks add elements to the queue. Something like this:

```
task type Worker2

task body Worker2 is

    elt:  Q_Element

    begin

        getElement()

        --- some sort of routine to get an element, code not shown here

        Queue.Enqueue(elt : in Q_Element)

        --- task calls protected procedure Enqueue to place the element in the queue

    end

end Worker2
```

This Worker2 task would just put one element in the back of the queue. If you wanted it to put multiple elements on the queue you could use a while loop based upon some condition (if your queue has a maximum length, it would have to be sure the queue is not full before placing the element).

Unit 3 Project: The Garage Door State Machine

Eric W. Wasiolek

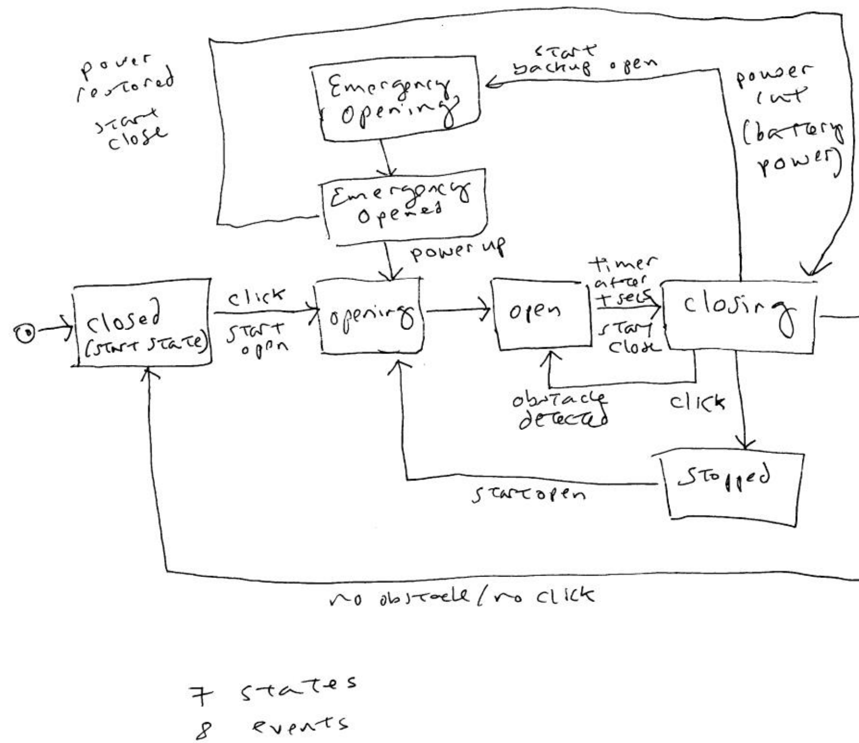
Concurrent and Distributed Systems

Professor: Dr. Bo Sanden

Date: February 15, 2016

PART I: STATE DIAGRAM

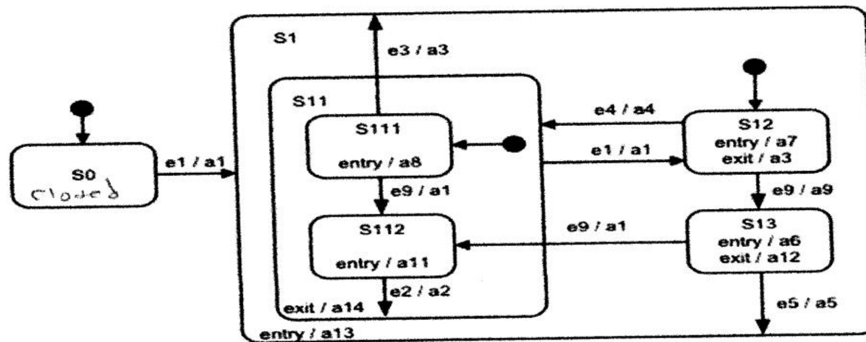
Here was my original state diagram for the garage door opener, based upon the textual description. In my diagram there are seven states, as I added the Stopped state between Closing and Opening. I also have eight events. I think this describes the garage door opener accurately in terms of its stated operation. However, to be consistent with the state diagram provided in this assignment, I will remove Stopped and go directly from Closing to Opening.



This diagram, as far as I'm concerned, exactly describes the states and their interrelationships through events. I know this is not in exact state diagram format. But it correctly describes the events and the states. The diagram starts in the Closed state. With a click event the door starts opening and enters the Opening state. It then proceeds to Open. After t seconds, it starts to

close, enters the Closed state. If there is an obstacle it goes back to the Open State. If there is a click event while closing, it will go back to the Opening state. If there is no obstacle and no click Closing proceeds to the Closed state. This is the basic operation of the garage door. Then if there is a power cut, the Closing state will go to Emergency Opening, which will then proceed to Emergency Opened. If the power is restored at this point, Emergency Opening will go to Closing. If the system is powered up due to battery, Emergency Opened will go to Opening. This is how you described the operation and what I put in the diagram.

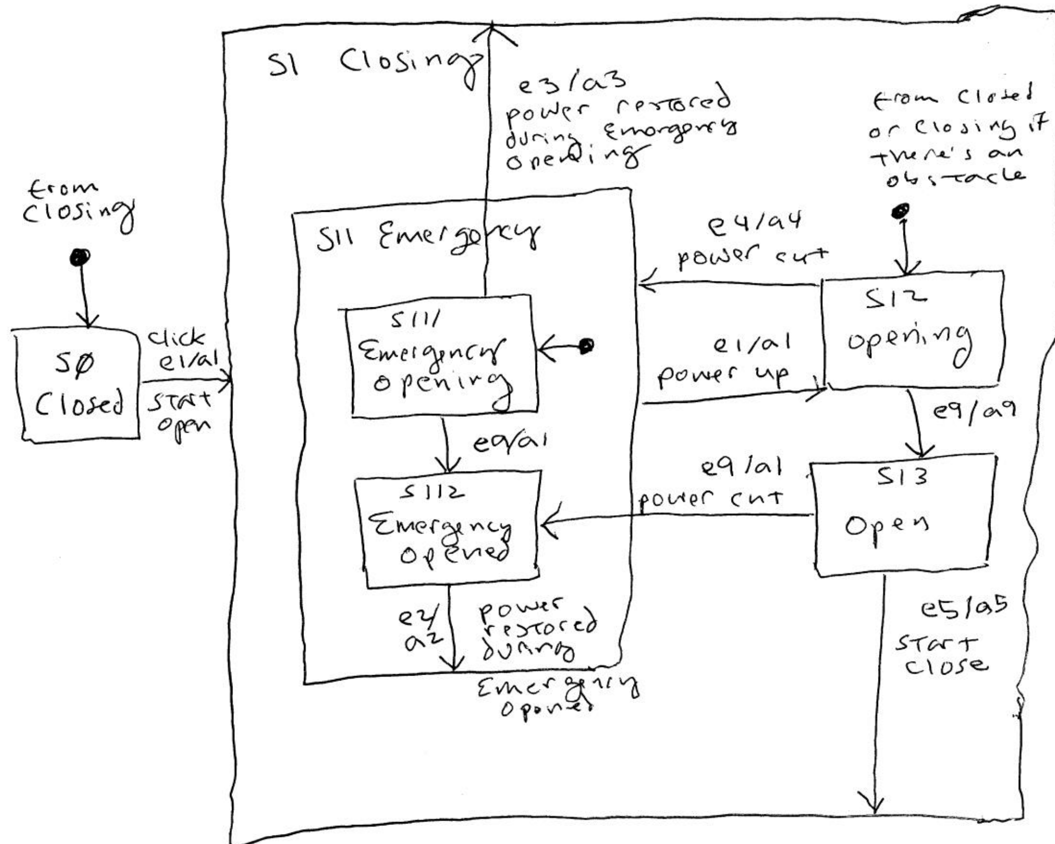
The next question is how to represent the state diagram below with these states and events:



series. Event #1 occurs in state S0 and causes a transition to a different state. Event #2 occurs in that new state and so on. States can be visited more than once.

Please submit your assignment.

The question in this diagram is how to label the states and events (and actions). I have done so below to the best of my ability:



This is the best I could do interpreting your unlabeled diagram. I have given the states and events and actions labels. I don't know if this is right, but it's as good as I can do. S0 must be Closed, as it is the starting state and the garage door starts in the state Closed. e1 would then appear to be click with the action start open, which connects to S12, the Opening state. e9/a9 would seem to just go from the Opening state to the Open state. e5/a5 would appear to go from the Open state to the Closing state, as indicated in my original diagram, Open only goes to Closing. Since there are only 5 boxes and 6 states, the only way I could get Closing in the diagram is to consider S1 Closing. I do not know if that is right. There is an arrow to S0, Closed, which I am supposing is coming from S1, Closing. Closing goes to Closed if there is no obstacle and no click. If there is an obstacle, Closing goes to Opening. It's not clear that that is modeled here. I tried to put it in the arrow pointing to S12 Opening. Clearly this interpretation has problems, and it is not completely right, but I am hoping it is not completely wrong.

In the case of a power cut while in Opening or Open, these states go to the superstate S11, which I take to be the Emergency state. Specifically, if the power cut occurs while in the Opening state, e4/a4 goes to the Emergency Opening state. If the power cut occurs during the Open state, e9/a1 goes to the Emergency Opened state. If there is a power up event, Emergency goes back to

Opening. If there is a power restore event Emergency Opening (S111) or Emergency Opened (S112) go to a Closing state (S1). Again, I don't know if any of this is right, but it is the best I could do.

PART II: EVENT SERIES

There was one thing that confused me when I saw the event series. The events were not contiguous, i.e., they didn't form a pathway through the state diagram. Consider the event series, e1, e4, e3. This forms a pathway from Opening to Emergency Opening to Closing. That's fine. But the next event, e9 is a transition from Opening to Open, or Emergency Opening to Emergency Opened, or Open to Emergency Opened (e9 occurs three times in the diagram). None of these are contiguous with Closing, the last state that e3 ended in. So this event series skips around, which doesn't make any sense to me.

Again, the best I can do is this:

e1/a1: is start open, and moves the Closed state to the Opening state.

e4/a4: is a power cut and moves the Opening state to the Emergency Opening state

e3/a3: is power restored and moves the Emergency Opening state to the Closing state.

e9/a9 or e9/a1: occurs three places in the state diagram, so it is either a transition from the Opening state to the Open state, a transition from the Open state to the Emergency Opened state, or a transition from the Emergency Opening state to the Emergency Opened state. If we consider #4 and #5 together, the two e9 events would be transitioning from the Opening state to the Open state and transitioning from the Open state to the Emergency Opened state.

#4, 5, and 6: e9, e9, e2. This would be the series transition from the Opening state to the Open state, a power cut that transitions from the Open state to the Emergency Opened state, and a power restore which transitions from an Emergency Opened state to a Closing state.

e1/a1: Number 7 would indicate a start open again moving from a Closed state to an Opening state.

On this last one I may be wrong. It may also be, for #4, 5, 6, and 7, or e9, e9, e2, e1: Transitioning from the Opening state to the Open state (e9), a power cut that transitions from the Open state to the Emergency Opened state (e9), a power restore transitioning to the boundary of the Emergency state, and e1 transitioning back to the Opening state with a power up.

Unit 4 Project

Deadlock and Vector Clocks

Concurrent and Distributed Systems

Eric W. Wasiolek

Professor: Dr. Bo Sanden

Date: February 29, 2016

C&DS Unit 4 Project: Wait Chains and Deadlock

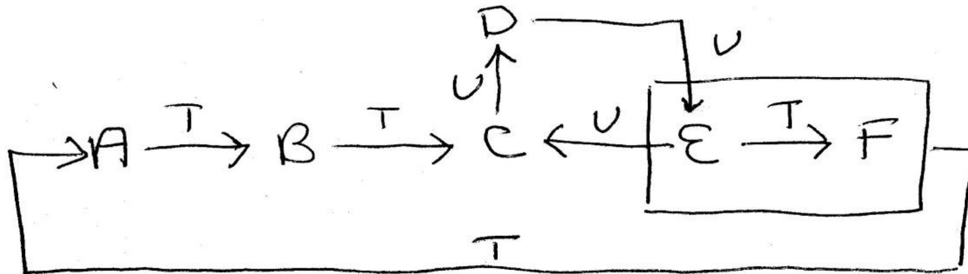
I'm not sure I did Part 1 correctly, but I did the best I could.

Part 1: There are 6 resources, A through F, each with the operations Acquire and Release. Upon return from X. Acquire the calling thread has exclusive access to resource X. By calling X. Release, the caller releases the exclusive access. There are two thread types, T and U, with the following logic:

T:		U:	
	A.Acquire;		E.Acquire;
	B.Acquire;		C.Acquire;
	A.Release;		E.Release;
	C.Acquire;		D.Acquire;
	B.Release;		C.Release;
	C.Release;		E.Acquire;
	E.Acquire;		E.Release;
	F.Acquire;		D.Release;
	A.Acquire;		
	A.Release;		
	F.Release;		
	E.Release;		

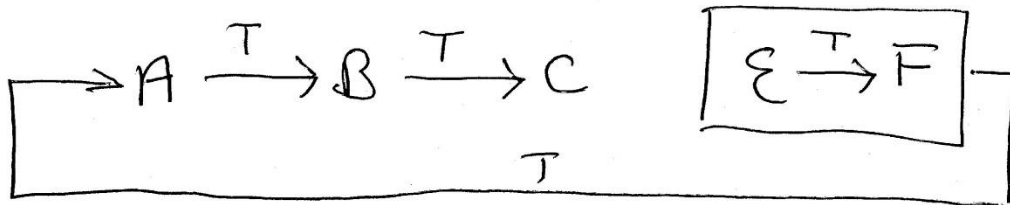
Draw a wait-chain diagram.

Part 1 Wait Chain Diagram



Then consider the following scenarios:

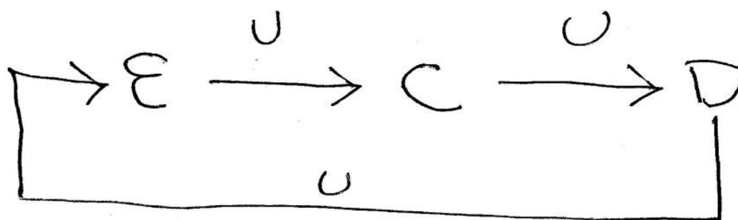
- **There are only instances of T. How many T instances does it take to create deadlock?**



T's don't produce a circular wait chain because C is not held when E is acquired. So there would be no deadlock no matter how many T's there were.

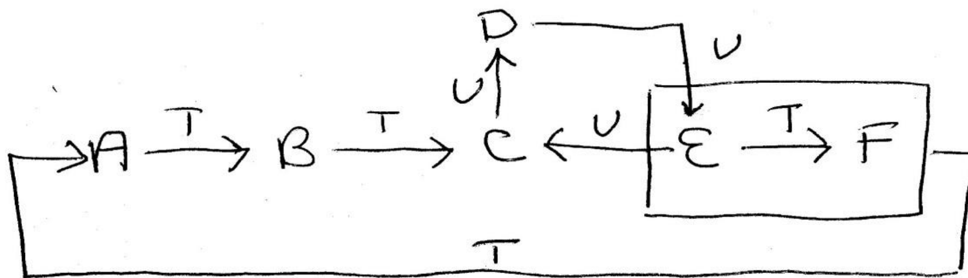
- **There are only instances of U. How many U instances does it take to create deadlock?**

2



In the scenario with only U's as instances a circular wait chain is created. So you would have deadlock with three U's.

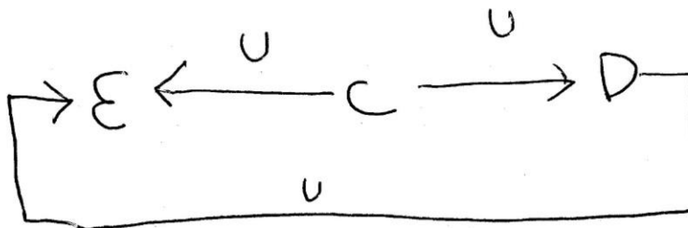
- **There are many instances of T and some instances of U. How many Us does it take to create a deadlock? How many Ts need to participate in the circular wait?**



This scenario would seem to be similar to the original wait chain. There is a circularity created with 4 T's and 2 U's. It would seem here just 2 U's create deadlock. Also from C to D to E, there are 3 U's and this is a circular loop that may create deadlock as well.

- **Assuming that each U thread needs exclusive access to C+D together at one point, C+E at another point, and D+E at a third point. Give an example of an order rule that allows this but prevents deadlock. (U may have to be restructured to comply with the rule.)**

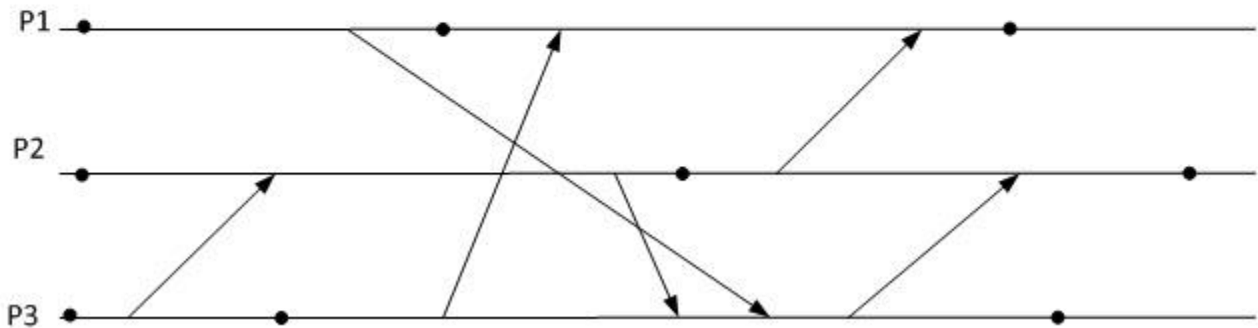
(4)



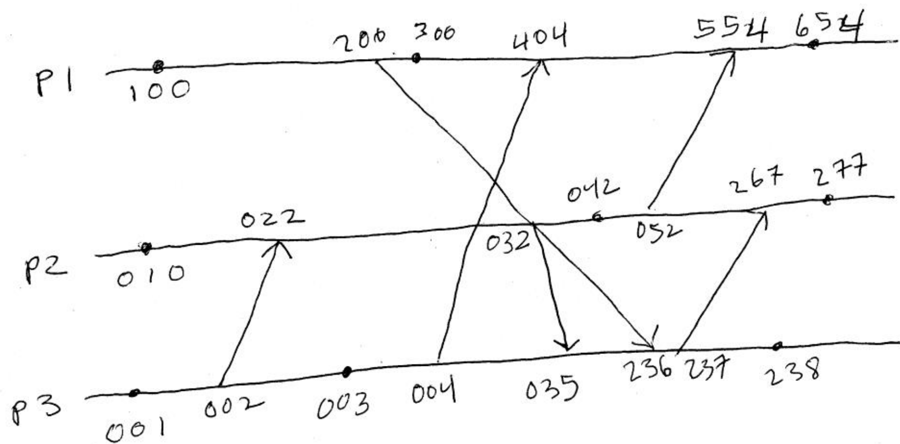
1. To avoid deadlock, you could limit the number of entities so they can't populate the wait chain, like have only two U's.

2. An order rule of C < D < E, i.e., C can only acquire a resource greater than itself, would not work here, because both C also acquires D and C also acquires E would be okay. So you have to have a different type of order rule. You could say an entity can only acquire a resource one greater than the one it currently has, so C could acquire D but C couldn't acquire E. This might work.

Part 2: What is each node's (process's) vector clock value at the end? P1's clock starts at (1, 0, 0). P2's clock starts at (0, 1, 0). P3's clock starts at (0, 0, 1).



The vector clock diagram would look like this:



The vectors at the end would be (5,6,4) for P1, (2,7,7) for P2, and (2,3,8) for P3.

Part 3 was deleted according to your instructions.

Part 3: The happens-before relation is a partial order. So is the order rule you used as one approach for deadlock prevention. What does it mean that the order is only partial? Do not only quote a formal definition. Use an example to illustrate if you wish.

References

Sanden, C. (2011). Design of Multithreaded Software: The Entity-Life Modeling Approach. Wiley, Singapore.

Unit 5 Project

Maekawa Algorithm and Lamport Clocks

Concurrent and Distributed Systems

Eric W. Wasiolek

Professor: Dr. Bo Sanden

Date: March 12, 2016

Part 1

In a two-phase, total-order multicast system, messages are exchanged in the following sequence between nodes S, G1, and G2.

G1 and G2 are the members of the group.

(a1 is an acknowledgment of m1; a2 is an acknowledgment of m2)

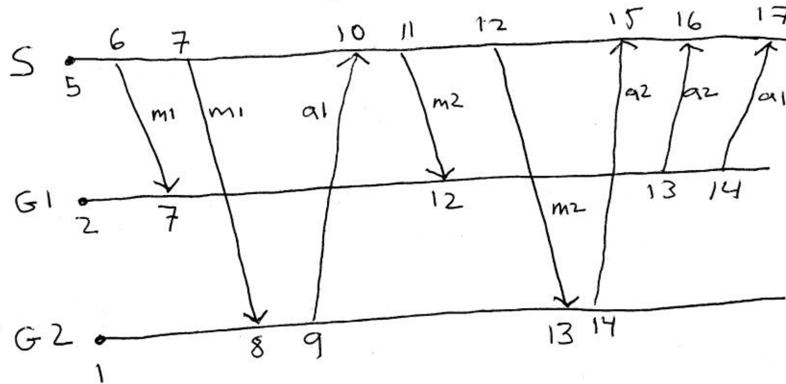
- **S sends m1 to G1**
- **S sends m1 to G2**
- **G2 sends a1 to S**
- **S sends m2 to G1**
- **S sends m2 to G2**
- **G2 sends a2 to S**
- **G1 sends a2 to S**
- **G1 sends a1 to S**

Let the Lamport clocks at each node start at: S: 5, G1: 2, G2: 1

In which order and at what times are m1 and m2 delivered?

Please submit the entire sequence of messages, including their times.

Total order multicast System
using Lamport clocks



msg	Suggested Time	Delivery Time
m1	7	8
m2	12	13

Delivery order: m1 is sent before m2

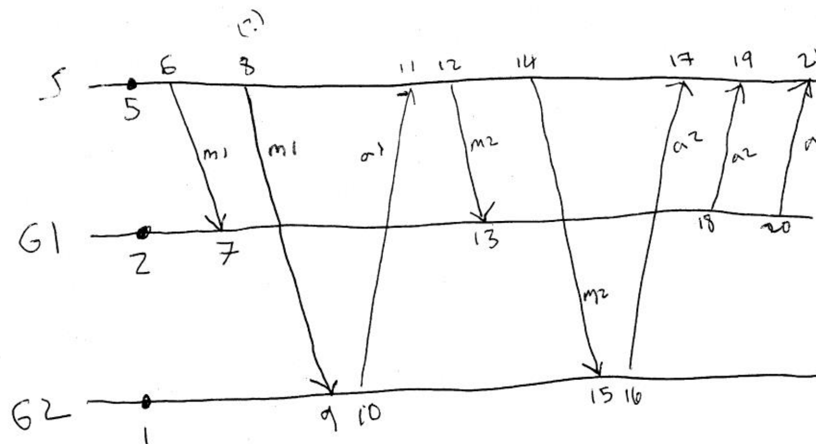
This is scenario one and the correct one according to what you said. Each clock at each node is independent and doesn't know what the clock is at another node unless it receives a message.

Included in the diagram are the Lamport clock times for each node.

It ends up that message 1 is delivered before message 2 as it has an earlier delivery time.

I've included the second scenario which you said would possibly be okay too, just for your edification and delight.

Alternate Lamport clocks
Scenario you said might be okay too.



msg	Suggested Time	Delivery Time
m1	7	9
m2	13	15

Delivery order: m1 is sent before m2

Here the times are a little different, but constantly increasing per node as is required for a Lamport clock. Here the messages are delivered in the same order, m1 then m2, with delivery times 9 and 15 respectively.

Part 2

Make a state diagram of Maekawa's voting algorithm on the slide "Maekawa's algorithm" in the "Distributed mutual exclusion" slide set. This can be done in various ways. One option is the following 5 states:

1. **Released and not voted**
2. **Released and voted**
3. **Wanted and not voted**
4. **Wanted and voted**
5. **Held and voted (for self)**

Events are:

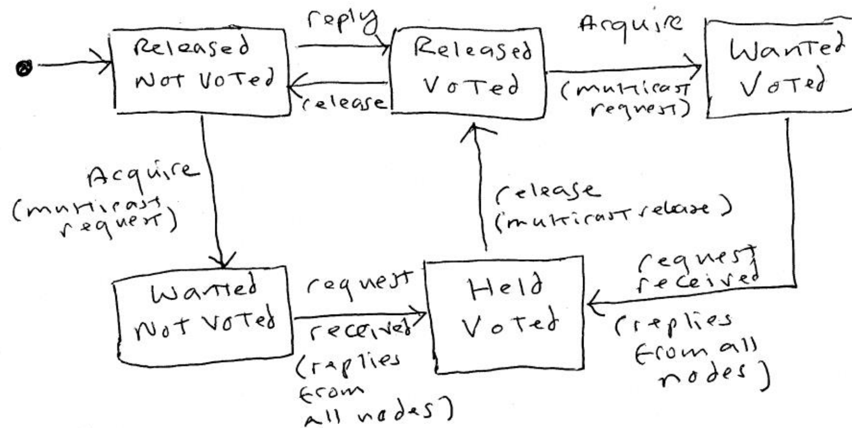
***request_received*, etc., for messages arriving from other nodes**

***acquire* for when the local node wants the lock**

***release* for when the local node gives up the lock.**

It may be quite difficult to get this absolutely correct, so partial credit will be given.

Here is the state machine diagram for Maekawa's algorithm.



There are 5 states: Release Not Voted, Released Voted, Wanted Voted, Wanted Not Voted, and Held Voted. There are 4 events: acquire, request received, release, and reply. In parenthesis I give more information about the event.