

CSE 120 Winter 2001 Midterm Examination Solutions

1. (10 points) Consider the following proposed solution to the two-process mutual exclusion problem: *This protocol is the Bakery algorithm specialized to two processes, and so it is both safe and live.*

```
cobegin
int try0 = 0, try1 = 0, turn0 = 0, turn1 = 0;
while (1) {
    try0 = 1;
    turn0 = turn1 + 1;
    try0 = 0;
    while (try1) ;
    while (turn1 != 0 && turn1 < turn0) ;
    critical section;
    turn0 = 0;
    non-critical;
}
||
while (1) {
    try1 = 1;
    turn1 = turn0 + 1;
    try1 = 0;
    while (try0) ;
    while (turn0 != 0 && turn0 <= turn1) ;
    critical section;
    turn1 = 0;
    non-critical;
}
coend;
```

- a) Is this a *safe* solution to the two-process mutual exclusion? If so, then give an informal argument as to why it is; otherwise, give an adversary schedule demonstrating that it is not.

It is safe: a process in its critical section has its value of turn nonzero. The possible race would occur if process 0 enters the critical section because $\text{turn0} < \text{turn1}$, and then process 1 enters the critical section because $\text{turn0} = \text{turn1}$. But, in an execution in which $\text{turn0} = \text{turn1}$ occurs only when both processes simultaneously read each other's value of turn. To enter its critical section, process 0 must test $(\text{turn1} < \text{turn0})$ while turn1 is still zero. For this to occur, try1 would be 1, and the loop while (try1) will block process 0.

- b) Is this a *live* solution to the two-process mutual exclusion? If so, then give an informal argument as to why it is; otherwise, give an adversary schedule demonstrating that it is not.

It is live. When a process wishes to enter the critical section its value of turn will be nonzero. If the other process is in the critical section then it will eventually leave. And, if that process attempts to re-enter the critical section it will obtain a value of turn larger

than the process wishing to enter. Its value of `try` will remain zero while it loops waiting its turn to enter.

2. (15 points) Consider a process p in an entry procedure and another process q waiting on a condition variable (or the implicit condition variable for Java-style monitors). With Hoare-style monitors, if p signals q then p immediately relinquishes the monitor lock to q and waits on the *urgent queue*. With Mesa- and Java-style monitors, when p notifies q then q waits to reacquire the monitor lock while p continues to hold the lock.

Another proposed weakening of monitors allows a *signal only as the last statement of an entry procedure*. With this new kind of monitor, if p signals q then p passes the monitor lock to q . Note that p no longer needs the lock because the *signal* is the last statement of the entry procedure; p is leaving the monitor anyway.

- a) In the same manner that I gave a semaphore implementation of Hoare-style monitors in class and that you gave a semaphore implementation of Java-style monitors in the second homework, give a semaphore implementation of this new kind of monitor. You need to describe the `struct` that is allocated for each monitor, give the code that is executed before and after an entry procedure invocation, and the code for *signal* and for *wait*.

For simplicity, assume that the monitor has a single condition variable x . There are a variety of ways to implement this kind of monitor. The difficulty is the conditional release of the monitor lock after invoking an entry procedure. Below, I assume that a process can obtain its identity by calling `PID`.

```
struct monitor {
    semaphore mlock = 1;    // monitor lock
    ProcessID locker = 0;    // id of process holding mlock
    semaphore xblock = 0;    // queue of processes blocked on x
    semaphore xcount = 0;    // number of processes blocked on x
} m;
```

An entry procedure invocation of `Foo(...)` becomes

```
P(m.mlock);
m.locker = PID();
Foo(...);
if (m.locker == PID()) V(m.mlock);
```

`x.wait` becomes

```
m.xcount++;
m.locker = 0;
V(m.mlock);
P(m.xblock);
m.locker = PID();
```

`x.signal` becomes

```
if (m.xcount > 0) {
    m.locker = 0; m.xcount--; V(m.xblock); }
```

- b) How do these new monitors compare with Hoare-style monitors? In particular, are there any problems that can be solved using Hoare-style monitors but not with new kind monitors? Explain.

They have the same power as Hoare-style monitors. One can easily implement semaphores using this new kind of monitors, and we know that Hoare-style monitors can be implemented using semaphores.

3. (10 points) Consider an *Election* object that supports a single method

```
int Vote (int vote); // vote is either 0 or 1
```

The object assumes that n processes can invoke this method for some odd and positive value of n . The method returns the result of the election: once a majority of the processes vote for 0 or 1, all invocations return the majority value.

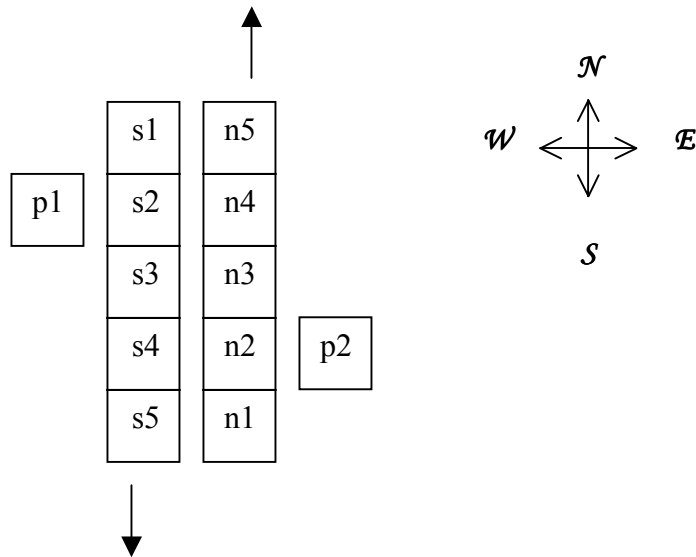
- a) Give a Hoare-style monitor that implements the *Election* object. Be sure to give the monitor invariant and the condition associated with each condition variable you use.

```
monitor Election {
private:
    int votes0 = 0, votes1 = 0;
    // inv: votes0 >= 0 && votes1 >= 0 && votes0 + votes1 <= n
    condition haveMajority;      // votes0 > n/2 || votes1 > n/2
public:
    entry int Vote (int vote) {
        if (vote == 0) votes0++;
        else if (vote == 1) votes1++;
        if (votes0 <= n/2 && votes1 <= n/2) haveMajority.wait;
        haveMajority.signal;
        return (votes1 > n/2);
    }
}
```

- b) Give a Java-style monitor that implements the *Election* object.

```
class Election {
    int votes0 = 0, votes1 = 0;
    public synchronized int Vote (int vote) {
        if (vote == 0) votes0++;
        else if (vote == 1) votes1++;
        if (votes0 <= n/2 && votes1 <= n/2) wait();
        notifyAll();
        return (votes1 > n/2);
    }
}
```

4. (20 points) *Gridlock* can be modeled as deadlock. Consider the following schematic diagram of a piece of a two-lane highway. There is a northbound lane and a southbound lane. The piece shown here is long enough to accommodate five cars each direction. In addition, there is a driveway to the west ($p1$) and a driveway to the east ($p2$).

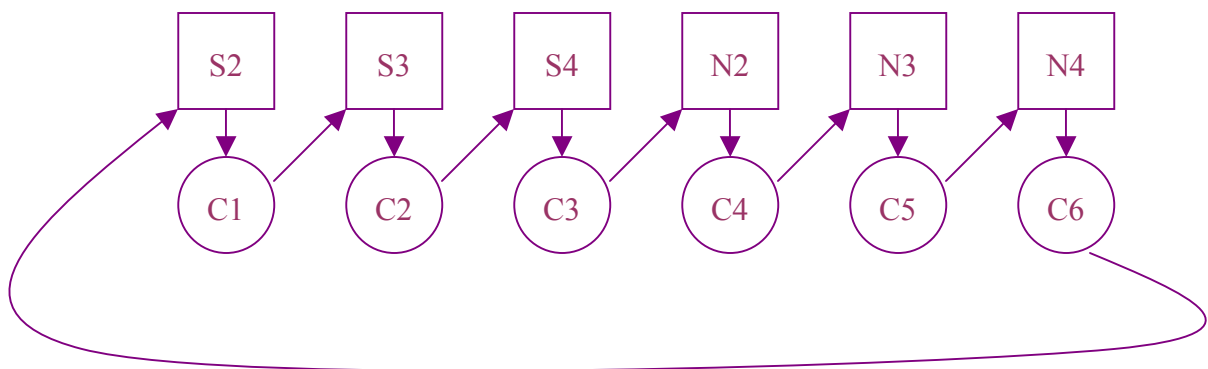


When driving north, a car c may occupy one space in the northbound lane; say, space $n3$. To move to space $n4$, there cannot be any car in that space. And, the car behind c cannot occupy space $n3$ until c first moves to $n4$. Thus, in terms of locking resources, a car is space i attempting to move to space j first obtains the lock on space j , moves to space j , and then releases the lock on space i .

To pull into driveway $p1$ a car must first occupy space $s2$. A car turning right into driveway $p1$ does so from space $s2$; a car turning left into driveway $p1$ does so by first acquiring space $s2$ from space $n4$, and then by acquiring space $p1$ from space $s2$. A similar restriction holds for cars pulling into driveway $p2$ from space $n2$: northbound cars do so via space $n2$, while southbound cars do so via space $s4$ and $n2$.

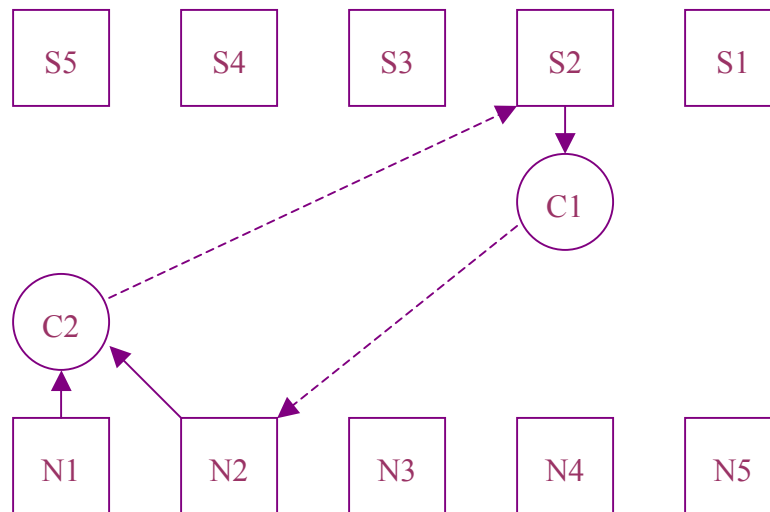
To keep things simple, assume that cars don't enter the road from either of the driveways. And, a car that is in space $n5$ heading north will eventually be able to move thereby free space $n5$. The same is true for a car heading south in space $s5$, a car heading west in space $p1$, and a car heading east in space $p2$.

- a) Draw a resource allocation graph that illustrates a *deadlock state* of this system.



- b) Assume that any car on this piece of highway that hasn't passed the driveway could decide to turn. For example, a car in space $n1$ could decide to turn into driveway $p1$ or driveway $p2$, and a car in space $n3$ could decide to turn into driveway $p1$.

Suppose that the cars use the Banker's algorithm to avoid deadlock. Show a state of the system, using a maximum claims graph, that indicates a request that would be blocked because the resulting state is unsafe. If you wish, draw in only the maximum claims edges needed to show that the state is unsafe.



Consider a car C1 in space S2. It has a claim on N2 because it may turn into P2. Let another car C2 be in N1. It has a claim on S2 because it may turn into P1. When C2 attempts to move into N2, the following maximum claims graph results. It contains a cycle, and so C2 is not granted N2 until C1 moves to S3.

Some of you gave deadlock states and argued they were also unsafe. It's true that a deadlock state is also an unsafe state. But, such a state cannot be attained using the Banker's algorithm, which avoids deadlock states.

(25 points) Some short questions

- a) To use the Banker's algorithm, each process must state its maximum claim for each resource it may attempt to allocate. For example, if a process might need 100 buffers but usually only needs 2, it must still state a priori that it has a maximum claim of 100 buffers.

Deciding appropriate values for maximum claims can be hard. One simple idea is to have a process set its maximum claim equal to the total number of units of that resource. For example, suppose that the buffer pool mentioned above contains 10,000 buffers. We could have our process state a maximum claim of 10,000 rather than 100.

Suppose each process were to follow this policy: for every resource whose allocation is controlled by the Banker's algorithm, each process states a maximum claim equal to the total number of units of that resource. What is the effect of such a policy? Give a description of the resulting executions.

It's terrible. As soon as any process allocates a single buffer, then no other process can allocate any buffers. No concurrent use of the buffers is allowed. When multiple resources are used, a process allocating one unit of any resource will result in all other process' requests for any resource to be held.

- b) Consider an elevator and the people using the elevator as a concurrent system. Give five properties of the system: two safety, two liveness, and one of your own choice. Be sure to identify each property you give as a safety or liveness property.

Safety: No more than 10 people can be in the elevator at any time. The elevator can be at no more than one floor at a time. The elevator door is open only when the elevator is stopped and at a floor.

Liveness: The elevator eventually changes floors. If a passenger pushes the call button then the elevator will eventually stop at the floor. Once the elevator stops at a floor the door will eventually open.

- c) The argument for implementing Java-style monitor semantics rather than Hoare-style monitor semantics is that they have better performance. Give a brief explanation why this is so. Give another brief explanation of a situation in which the Java-style monitor-based implementation would give *worse* execution than the Hoare-style monitors.

With Hoare-style monitors a process that signals must leave the monitor. This can cause one unnecessary context switch if, for example, the process was about to leave the monitor anyway. However, with Java-style monitors a notifyAll can wake up many processes. If only one of these processes can make progress, then the rest of these processes will just call wait() again. This results in many unnecessary context switches.

- d) Consider the pseudocode given in class given for processor multiplexing using the interval timer and for implementing the semaphore P and V operations. Suppose we wished to add a system call `SLEEP x` that would have the calling process to be removed from the ready set of processes for x milliseconds. Describe how the pseudocode given in class could be changed or added to implement `SLEEP`. You don't have to actually give the new or modified pseudocode: a clear explanation will suffice.

Create a private semaphore for each process upon which it will P when it executes SLEEP. Associate a time with this semaphore (by, for example, including both in a struct) indicating when the process should wake up. Have Schedule set the interval timer to the minimum of the quantum and the remaining wakeup times of all the SLEEP semaphore times that are nonzero. When the interval timer fires, V all timer semaphores whose timers have expired.