Solutions
15-410, Fall 2024, Homework Assignment 1.

1 Tape drives (4 pts.)

Imagine the system is in the state depicted below.

Who	Max	Has	Room
A	3	0	3
B	3	1	2
C	3	0	3
System	3	2	-

*List one request **for a single tape drive** which the system should grant right away, and one request **for a single tape drive** which the system should react to by blocking the process making the request. Briefly justify each of your answers.*

If Process B requests one tape drive, the request should be granted, because the system would move to a different safe state, as depicted below.

Who	Max	Has	Room
A	3	0	3
B	3	2	1
C	3	0	3
System	3	1	-

In particular, after the request were granted there would still be one tape drive remaining, and one process with a maximal future need of one tape drive. In other words, Process B could request its final tape drive, be granted the request, and then finish up and release three tape drives. At that point either Process A or Process C could request and be granted as many as three tape drives, in which case that process could run to completion and return three tape drives for the other process. So granting Process B a second tape drive would allow the safe sequence (B,A,C), so the grant is a safe thing to do.

In the other direction, if Process A requests one tape drive, the system should block Process A before granting the request, because granting the request would move the system to an unsafe state as depicted below.

Who	Max	Has	Room
A	3	1	2
B	3	1	2
C	3	0	3
System	3	1	-

This state is unsafe because the system has only one free tape drive, but all three processes are entitled to request more than one. If the system were in this state, it might well happen that some process would exit without requesting anything further. However, it could also happen that both Process A and Process B would request two tape drives. In that case, the system would have no choice but to block both of them; because each of the processes would be blocked waiting for tape drives held by the other, that would form a deadlock. Formally speaking, in the state depicted above no process can request resources up to its maximum without blocking, so no process can request resources up to its maximum and complete in a timely fashion, thus there is no first item in a safe sequence, thus no safe sequence can exist.

2 "Son of Peterson's Solution" (6 pts.)

...There is a problem with this critical-section protocol. Identify a required property which this protocol does not have and then present a trace which supports your claim.

A violated requirement is bounded waiting. Intuitively, the real Peterson's Solution has the property that, if two threads contend and one loses, the winner will ensure that, even if it rushes around and enters contention again, the loser will win. This is ensured by the "after you" statement, `turn = j`. But this version has `turn = i` instead, so we should intuit that we could get a thread to enter the critical section an unbounded number of times.

Execution Trace

time	Thread 0	Thread 1
0	w[0]=1	
1		w[1]=1
2		turn=1
3	turn=0	
4	while:!(turn==j)	while:yes
5	return /*locked*/	while:yes
6	w[0]=0 /*unlock*/	
7	w[0]=1	while:yes
8	turn=0	while:yes
9	while:!(turn==j)	while:yes
10	return /*locked*/	while:yes

Note that the code being executed by the two threads at step 10 is the same as it was during step 5, and (this is critical!) the state variables are the same too: `w[0]==1`, `w[1]==1`, `turn==0`. Since it is possible to copy/paste lines 6 through 10 as many times as desired, Thread 0 can obtain the lock an unbounded number of times even though Thread 1 wants it too. What makes this not happen for the *real* Peterson's Solution is that then the system state is *not* the same because Thread 0 would set `turn=1`. Each time a winner wins it will arrange to lose next time.

Be careful when writing traces! What you write must describe execution paths that are *possible*, and must also be sufficiently *unambiguous* that it cannot simultaneously describe traces which exhibit a property you are claiming and traces that do not exhibit the property. For example, a trace that shows repeated locking by one thread, but does not ever show *unlocking* by that thread, is both impossible and ambiguous. It is not enough to write something down that reminds you, personally, of what various threads are doing. What you write must convince your reader that you completely understand a particular path through the code.

In terms of difficulty, this homework is easier than typical exam questions on similar topics. However, this homework represents some of the reasoning we expect you to carry out in order to detect and explain race conditions (and/or deadlocks).