## Problem 1

a)

LOCK

V=V-10

UNLOCK

b)

We need to disable the interrupt so that when changing the state of the lock, the process will never get descheduled by the scheduler. We then enable the lock after we have finished changing the state of the lock to allow for parrellalism

c)

When one process has already acquired the lock and the second process is trying to acquire the lock it can never succeed. The reason for that is after it jumps to the start following by the execution of the BNZ LOCK instruction. It will not be descheduled since the interrupt has already been disabled and doesn’t get re-enabled anywhere in this execution loop. So the process that is holding the lock can never release the lock.

Another potential problem is that the lock in our definition doesn’t have the concept of the owner. So the process that acquire the lock is not necessarily the process that releases the lock.

LOCK: STI

LOCK: CLI // disable the interrupt

TEST L

BNZ LOCK\_START

MOVE #1 L

STI

d)

Initialization:

Initialize a semaphore with value 1 called red\_lock

Initialize a semaphore with value 1 called green\_lock

enter:

if is green car:

while(true):

down(red\_lock)

down(green\_lock)

if (red\_count == 0):

green\_count++

up(green\_lock)

up(red\_lock)

break

up(green\_lock)

up(red\_lock)

if is red car:

while(true):

down(red\_lock)

down(green\_lock)

if (green\_count == 0):

red\_count++

up(green\_lock)

up(red\_lock)

break

up(green\_lock)

up(red\_lock)

exit:

if is green car:

down(green\_lock)

green\_count--

up(green\_lock)

if is red car:

down(red\_lock)

red\_count--

up(red\_lock)

## Problem 2

I don’t think this is covered any more.

## Problem 3

## Problem 4

a)

Assume we have n bits in total in our address representation. The for our logical address the first (m - n) bits represent the offset in the page table. We look up the page table by the corresponding offset, we find out the page address, then we concatenate the the (m - n) bits page address with n bits page offset.

b)

0.8 \* (120 + 30) + 0.2 \* (120 + 120) = 168 s

0.98 \* (120 + 30) + 0.02 \* (120 + 120) = 151.8 s

Two level paging:

0.8 \* (120 + 30) + 0.2 \* (120 + 120 + 120) = 192 s

0.98 \* (120 + 30) + 0.02 \* (120 + 120 + 120) = 154.2 s

The time it takes to translate the address will be longer.

c)

The advantage of the inverted page table is that it reduces the size of the page table quite siginificantly in modern compuer. However the downside for that is more time spent in actually look up the page table.

d)

if the page size is 64 bytes, then we can only have 4 frames, since 4 \* 64 = 256 bytes

Below we show the page that is currently in the physical memory, and page 0 contains the address 0 to 63, page 1 contains address 64 to 127 and so on

10 11 132 202 73 399 240 318 319 564 595 473

0 0 0 0 0 2 2 3 3 1 1 6

2 2 2 3 3 1 1 6 6 4

3 3 1 1 6 6 4 4 8

1 6 6 4 4 8 8 7

If the page size is 128 bytes, then we have 2 frames in total.

10 11 132 202 73 399 240 318 319 564 595 473

0 0 0 1 1 2 2 2 2 3 3 3

1 2 2 3 3 3 3 4 4 4

As we can see using using larger page size means less eviction and thus less overhead. However larger page size often mean more internal fragmentation. Also by using larger page size we can reduce the size of the page table.