Do all of questions 1 through 4.

1. This question concerns a system employing a single (single-core) processor running a Unix-like operating system, in which interrupts are handled on the current thread’s kernel stack. We have a user-level process in which two threads are running. One (the producer) is reading characters from standard input and depositing these characters into a buffer. The other thread (the consumer) is taking characters from the buffer (in the same order as the characters were put into the buffer) and writing them to standard output. Synchronization on the buffer is done with semaphores, using the standard producer/consumer pattern. Note that if a thread is waiting on a P (or wait) operation, it blocks, i.e., sleeps. Assume that standard input comes from the keyboard, and that it is set up so that each time a character is typed in, an interrupt is generated. The interrupt handler puts each incoming character into an input buffer in the OS kernel (different, of course, from the user-space buffer the two threads are using). Something similar may be happening for standard output, but we’re not concerned about it for this problem.

The OS kernel’s input buffer is of finite size. The interrupt handler puts characters into it. Characters are removed when the producer thread issues read system calls. In the current implementation, when the OS kernel’s input buffer is full, the interrupt handler must discard all typed-in characters.

A suggested improvement is to use producer-consumer synchronization on the OS kernel’s input buffer: the interrupt handler waits, using a P (wait) operation on a semaphore if the buffer is full. Explain why this might lead to deadlock.

Recall that semaphores are non-negative integers. The P (wait) operation on a semaphore S is defined as:

```
when (S > 0) [
  S = S-1;
]
```

The “when(exp) [ ...]” notation means that at some point when exp evaluates to true, the sequence of statements inside the square brackets is executed atomically. If exp is false, then the thread waits by blocking.

The V (post) operation on a semaphore S is defined as:

```
[S = S+1;]
```

The square brackets mean that the statement inside them is executed atomically. The V operation wakes up blocked threads waiting for the semaphore to become positive.

2. Your task is to implement pthread_cond_wait and pthread_cond_signal. You have the following code snippets that are known to work on a uniprocessor system that has a non-preemptible kernel:

```c
void pthread_cond_wait(pthread_cond_t *cv, pthread_mutex_t *m) {
    pthread_mutex_unlock(m);
    enqueue(cv->wait_queue, CurrentThread);
    thread_switch();
    pthread_mutex_lock(m);
}

void pthread_cond_signal(pthread_cond_t *cv) {
    thread_t *next = dequeue(cv->wait_queue);
    if (next != NULL)
        MakeRunnable(next); // puts next on the run queue
}
```

a. Show how to modify `pthread_cond_wait` so that it works on a uniprocessor that has a preemptible kernel. In particular, assume that clock interrupts occur at some frequency. The clock interrupt handler examines the current thread (if any) and, if its time slice has expired and there is another thread on the run queue
(we may assume all threads are at the same priority), it moves the current thread to the end of the run
queue and arranges so the first thread on the run queue becomes the current thread. If your solution
requires additional machinery, feel free to invent it as long as you describe what it does and it is
something that is commonly found in OS kernels. For example, if you need to mask certain interrupts,
you might use a routine “mask_x_interrupts()”, where x is the name of the interrupt you want to mask. In
a case such as this, if it’s necessary for such interrupts to be unmasked, you must indicate where this will
happen. You may assume that enqueue and dequeue are thread-safe and that thread_switch works as
discussed in the course. You need not show modifications to pthread_cond_signal.

b. We’d now like pthread_cond_wait to work on a multiprocessor system that has a preemptible kernel.
Show how the code could be modified to support this. Again, you may invent procedures as necessary,
and modify data structures as necessary, as long as it is clear what you are doing and your inventions
make sense. Be sure to supply useful comments! If you feel your solution to part a works here, simply say
so (with some explanation of why) rather than repeating the code.

3. An operating system has a simple round-robin scheduler used in conjunction with time slicing: when a thread’s
time slice is over, it goes to the end of the run queue and the next thread runs. The run queue is implemented as a
singly linked list of threads, with pointers to the first and last threads in the queue. Assume for parts a and b that
we have a uniprocessor system.

a. The system has a mix of long-running compute threads that rarely block and interactive threads that spend
most of their time blocked, waiting for keyboard input, then have very brief bursts of using the processor.
Assuming we want the system to have good interactive response, explain what is wrong with the
scheduler.

b. How might the scheduler be improved to provide good interactive response? (Hint: a simple improvement
is sufficient.)

c. We add three more processors to our system and add the appropriate synchronization (spin locks) to our
scheduler data structures. But this may result in a performance issue due to the multiple processors.
Describe this performance issue.

d. Describe what might be done to alleviate this performance problem, yet still have reasonable parallelism
(by which we mean that, in general, threads will not be in run queues waiting for processors while at the
same time there are idle processors).

4. We have an operating system whose only file system is S5FS. Suppose we decide to employ a RAID level 4 or
RAID level 5 disk system. Can S5FS take advantage of all the features of these systems? Explain. Which would
be better: level 4 or level 5? Assume that the block size of S5FS is equal to the disk-sector size. Recall that in
RAID level 4 one disk is dedicated as the parity disk, while in RAID level 5 parity information is spread across all
disks. (Hint: recall that among the features of RAID that are potentially available are: larger effective disk (n disks
provide more usable space than one), per-request speed-up for a single (single-threaded) user, and system-wide
speed-up for a collection of users.)

If you do all of the following correctly, you’ll get an A regardless of how well you do on the first four problems. If
you miss any of the following, your grade will be based solely on how well you do on the first four problems.

5. The original versions of Unix were numbered as “editions” — sixth-edition Unix, seventh-edition Unix, etc.
However, a separate sequence of Unix versions was commercialized by Bell Labs, given the names “System n”,
for various not necessarily integer values of n.

a. What was the value of n for the first version offered for sale to the public?

b. What was the value of n for the second such version?

c. What followed that?

d. What was the last such value?


a. What was it called?
b. The computer systems research group at UC Berkeley, starting with this variant, produced an improved version of Unix. These were given the names nBSD, for various not necessarily integer values of n. What was the first such value?

c. What was the last such value?

7. Sun Microsystems produced a version of Unix called Solaris.
   a. What was the number assigned to the first version they released of Solaris?
   b. What was the number assigned to the version following 2.6?

8. Microsoft produced a new OS in 1993 called Windows NT.
   a. What was its first version number?
   b. What did they call the version following version 4?
   c. What did they call the version following version 8.1?

9. Before this course was known as 1670 it was 167. Before that, it was known as 169 (i.e., there was just a full-credit 169 and no 167). What was the course known as before it was called 169?

10. What was CS 15 called before it was CS 15?