Ola! Greetings from Puerto Rico, where the air is warm and salty and the mojitos are cold and sweet. Please answer all questions for a total of 200 points. Keep it clear and concise: answers are graded on content, not style. I expect that you can answer each question within the space provided. If you need to make any assumptions that are not clear from the question, then please state them explicitly. Any kind of pseudocode is fine as long as its meaning is clear. You may assume standard routines like lists, queues, hash tables, etc.

You are allowed one sheet of notes; please do not refer to any other sources of information such as other students, laptops, tablets, phones, books, notes written on your hand, etc. Please sign this page to indicate that you have completed your exam within these rules. Good luck!

This is the solution copy. Answers are in italics. In general, an answer containing just the words in bold would receive full credit.

Name:

Signature:

<table>
<thead>
<tr>
<th>Problem</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>40</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
</tr>
<tr>
<td>3</td>
<td>40</td>
</tr>
<tr>
<td>4</td>
<td>50</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>total</td>
<td>200</td>
</tr>
</tbody>
</table>
1 Scheduling etc.: shorts (40 pts)

We have studied thread management and several policies for selecting the next job or thread to run on a CPU core on a context switch. The policies include FCFS and STCF (also called SJF) scheduling policies, and their preemptive variants: FCFS-P (also called RR) and STCF-P. We also considered a hybrid using adaptive priorities, called Multilevel Feedback Queue (MFQ).

(a) What causes a context switch? List all the causes you can think of.

- thread exit (or process is killed due to a fatal fault)
- thread yield (voluntary)
- thread block/sleep/wait (e.g., due to I/O or synchronization: locks acquire, CV wait, semaphore P)
- thread exhausts its quantum (timer interrupt) → preempt
- a higher-priority thread enters the ready state (for preemptive scheduling) → preempt

Events that do NOT normally switch the current thread (except as above): trap, fault, interrupt.

(b) What is the advantage of STCF relative to FCFS? What is the advantage of FCFS relative to STCF?

STCF: optimal average response time, but must predict the future to know the time-to-completion for each job or thread, and may cause starvation for non-short jobs.

FCFS:
- easy and fast (no need to predict the future, no sorting).
- fair
  + for FCFS, threads receive service in order of their arrival time
  + for FCFS-P, active threads receive service at the same rate

(c) One goal of MFQ is to keep I/O devices utilized. Why is this goal important? Briefly, how does MFQ achieve it?

Maximize concurrency by overlapping (slow) I/O times with CPU work. Using I/O resources efficiently (i.e. not leaving them idle when they could be doing useful work) is important for system throughput and response time.

MFQ increases the CPU priority of threads that do a lot of I/O, so they can get the next I/O operation started sooner. Each time the thread runs, the OS increases the priority of a thread if it blocks without using its full CPU quantum, and decreases the priority if it uses its full CPU quantum.

(d) What is a good choice for a scheduling quantum? Describe the bad things that happen if the quantum is “too long” or “too short”?

Too long: degrades to Run-To-Completion, so no benefit → threads can be stuck behind a long job.

Too short: too much time is spent context switching → high overhead.

Good choice: “it depends”. Maybe 1000X the context switch time? 5-100 milliseconds?
2 Deja Vu (50 points)

This problem asks you to write the core of a program to issue and service disk requests. There are \(N\) requester threads to issue disk requests and one servicer thread to service disk requests. Write procedures for the requesters and servicer.

Each requester issues a finite sequence of requests for disk tracks: you may use any pseudocode for a requester to decide on its sequence. Requesters issue requests by queuing them at the disk scheduler. The scheduler queue can contain at most MAX requests: requesters must wait if the queue is full. Requests are synchronous; each requester can have at most one pending request.

The servicer handles requests only when all living requesters have a pending request, or if the queue is full. The servicer may use any pseudocode to select which request to service next.

This problem is part of Project 1, so no solution is given here.

It is a straightforward bounded buffer producer/consumer problem, with multiple producers and a single consumer, and with the extra twist that the consumer runs only when all living producers are blocked.

Since producers depart when they complete their finite sequence of requests, you must keep track of the number of living producers. That requires some synchronization. A producer may depart while the consumer is waiting for it to produce, so you need another signal there.

In general, the solution requires two waits and two signals on the producer side (requester), and one wait and one broadcast (with optional additional signal) on the consumer side (servicer). Don’t forget to loop before you leap!

In general, the solutions are cleaner when each pass through the main loop produces or consumes exactly one request. Various solutions show up in which a thread may make multiple passes through the loop before it succeeds in producing or consuming a request. These solutions annoy the grader, and they tend to degenerate into hairballs. Hairballs take longer to write, and they require additional debugging time that is better spent sleeping or doing almost anything else.

On the exam, you can save yourself time by writing pseudocode. You don’t need to declare all your data structures or use real C++ STL interfaces or whatever and so on. Just say “enqueue(request)”: I’ll know what you mean. Sometimes it seems like students are afraid that I’m going to take off points for a missing semicolon or something. Sheesh. I just want to see you get the synchronization right. Shorter is usually better.
3 Back to the Future (40 pts)

This question asks you to describe some internals of a virtual memory (VM) system, such as your Project 2 due next week. You should answer with specific reference to the various data structures, internal states, and MMU interfaces that the VM system manages.

(a) What happens on a context switch from one process to another? (e.g., vm_switch)

Change the page table base register in the MMU to point to the page table of the process we are switching in to.

(b) What happens when a process exits? (e.g., vm_destroy)

Free page frames held by this process (e.g., by walking the page table and adding page records to a free list). Adjust clock structures to remove tracking of pages held by this process (e.g., by removing page records from clock queues, and/or adjusting the clock hand). Delete the page table for the exiting process. Free the blocks allocated for this process on backing store.

(c) What happens if there is a fault (e.g., vm_fault) on a page that is already resident?

The purpose of such a fault is to notify the VM system that a page marked for sampling is being referenced, and/or that a clean page is being modified. Set the reference bit for the page, and enable read access in the PTE. If the access is a store, then set the dirty bit, and enable write access in the PTE.

(d) What happens if there is a fault (e.g., vm_fault) on an invalid address, e.g., a page that is outside of the arena?

This is an error. The VM system detects this case when it validates that the faulted address is not lower than the base of the arena, and not beyond the last byte of the arena space allocated with vm_extend. If the faulted address is out of range, then the vm_fault fails and the process is killed.
4 Gated Community with Semaphores (50 pts)

This question asks you to implement a gate for the Queen’s Castle. Wandering minstrels and other threads arrive at the gates of the castle throughout the day. If the gate is open they may enter the castle, but if the gate is closed the travelers must wait for the Castle Guard to open the gate. Write three procedures: arrive(), openGate(), and closeGate().

Arriving threads wait in arrive until they can enter the castle. The Guard thread may use openGate and closeGate to open and close the gate at any time.

Semaphore gate;
Initialize gate to 0 (closed) or 1 (open).

Arrive():
    gate.P();  // wait for the gate to be open
    gate.V();  // leave state of gate unchanged
    // traveler goes on its merry way

OpenGate:
    gate.V()

CloseGate:
    gate.P().

Note that due to the genius and elegance of semaphores, no additional mutual exclusion is required.

A full-credit pseudocode solution requires only four characters:

P
V
V
P
5 Something extra (20 pts)

For the thread manager (Project 1) you allowed a thread to call `signal` on a condition variable without holding the corresponding mutex lock. This behavior was regarded as “suspect” but was not necessarily an error. Signaling without the lock can improve performance on multicore systems (multiprocessors). Why? But sometimes this behavior is incorrect. Give one example where the lock must be held when `signal` is called for the program to behave correctly.

A signal or broadcast may cause waiting threads to start running immediately on other cores. If the signaler holds the mutex, these threads must block on the mutex. If the mutex is free, then at least one of them will acquire it without blocking again. The reduced context switches (savings of two context switches per signal: one to block on the mutex, one to wakeup) can be significant. This issue is discussed in detail in the Birrell paper.

When is it incorrect to signal without holding the mutex? In general, the mutex helps to avoid races between the signal and the wait. For example, in some cases, if the signal occurs before the wait, the signal is lost and the waiter waits forever. This is called the “missed wakeup problem”. Condition variables have built-in mutexes in part to avoid this problem.

An example discussed in class is ping pong in which two threads alternate strictly in this loop:

```
Lock()
Do until tired {
    // yadda yadda
    Signal();
    Wait();
}
Unlock();
```

We can transform it to this if we don’t mind some extra useless lock/unlock activity:

```
Do until tired {
    Lock();
    Signal();
    // yadda yadda
    Wait();
    Unlock();
}
```

But this could lead to a missed wakeup and resulting deadlock on any iteration:

```
Do until tired {
    Signal();
    Lock();
    // yadda yadda
    Wait();
    Unlock();
}
```