Duke University, Department of Computer Science
Qualifying exam: operating systems (CPS 510, Spring 2020)

This exam has four parts. Some of the questions ask you to summarize certain OS concepts and so invite long answers. Just give the best answer that you can in the space and time available. "Answers are graded on content, not style." A few key words or pictures are often sufficient to show that you are familiar with the concepts.

Part 1. Scheduling

Consider a server that receives a continuous sequence of client requests. Each request runs as a task in a single thread. This part asks you to consider the scheduling policy that controls how the server CPU is shared among the set of tasks that are pending at any given time.

For this part you may assume that tasks are CPU-bound and that each task's CPU service demand (its time to run) is known in advance. To keep it simple we assume a single processor with a single core slot: no CPU parallelism.

(a) Give an example schedule to show that a Shortest Task First (sometimes called Shortest Job First or Shortest Remaining Processing Time) scheduling policy can reduce average response time relative to a FIFO (first in first out) policy.

(b) Give an example schedule to show that a FIFO policy can reduce average response time relative to Round Robin (RR). RR is FIFO with task preemption after a fixed time quantum.

Part 1. Scheduling, continued

(c) Given a finite sequence of tasks, does Shortest Task First always have lower average response time than FIFO or Round Robin? Explain (sketch a proof) or provide a counterexample.

(d) Given a finite sequence of tasks, does FIFO always have lower average response time than Round Robin? Explain (sketch a proof) or offer a counterexample.

Part 1. Scheduling, continued

(e) Suppose your team leader asks you to implement a preemptive Shortest Task First policy in your server. Is this a good idea? What practical difficulties might you encounter (other than predicting task times correctly)?

(f) What advantage does Round Robin offer relative to Shortest Task First and/or FIFO? Propose a metric by which it is superior to these alternatives. ”Superior” means “generally better and never worse”.

Part 2. Thread blocking

Threads often block (sleep or wait) while executing in the operating system kernel. More precisely, when a thread is executing kernel code on a processor in kernel mode, the code may block the thread by calling an internal kernel procedure—let's call it **sleep()**. This part asks how and why blocking occurs in your favorite operating system.

(a) List three examples of events that might cause a thread to switch to kernel mode.

(b) What does **sleep** do? For example, what does the core do next after its current thread sleeps?

Part 2. Thread blocking, continued

(c) List eight examples of common reasons for kernel code to block by calling **sleep**. For each example, list the operation that called **sleep** and an object it was operating on (e.g., join on a child thread) and an event that might cause the thread to wake up again (e.g., child thread completes).

(d) Which examples from (c) do you think are the most common/frequent? Surely the answer depends on the system and workload. Choose three of the examples and identify a system factor and a workload factor that influence their relative frequency.

Part 3. Concurrency Control

A **monitor** is a classic abstraction for concurrency control using the four operations on the right. Monitors appear in modern programming languages including Java and C#, and as linked mutexes and condition variables in Modula-2 and pthreads. The pseudo-code on the right is a flawed implementation of monitors using **semaphores**. (P is down, and V is up.) This question asks you to summarize what is wrong with the code.

Write a list of **four distinct correctness properties** that this solution violates even when used correctly. Outline a fix for each one. Feel free to mark the code.

```
semaphore mutex(0);        /* init 0 */
semaphore condition(1);    /* init 1 */

acquire() {
    mutex.p();
}

release() {
    mutex.v();
}

wait() {
    condition.p();
}

signal() {
    condition.v();
}
```

Part 4. Block I/O

Suppose a program P reads and writes large files stored on spinning hard disks. P reads and writes its files sequentially in their entirety. Processes running P often read files created during earlier runs of P. These files are much larger than system memory. Oddly, P reads and writes one byte at a time, e.g., by issuing read and write system calls for size one byte.

Ideally, the operating system enables processes running P to read and write at "spindle speed"---the maximum bandwidth that the disk system can transfer data when there are no seek delays. How can it do this? To keep it simple you may assume a single locally attached disk.

To enable I/O at spindle speed in scenarios like this, operating systems use various mechanisms (techniques or design choices) that operate on units of fixed-size **blocks** of file data. Identify four such mechanisms. Briefly summarize each mechanism and explain why it is needed, i.e., why is I/O too slow without that mechanism?