CSc 452: Principles of Operating Systems
Fall 24 (Lewis)

Test 1

Thu 10 Oct 2024

Solutions

Name: _____ NetID: _____

Question	Points	Score
Page 1	20	
Page 2	20	
Page 3	20	
Page 4	20	
All False	20	
Total:	100	

NetID: _____

1. (a) (10 points) What is a "context switch?"

Solution: We remove the current running thread from the CPU, and store its state into memeory, and then restore another thread from memory, restoring its state into the CPU.

(b) (6 points) If we didn't have a timer interrupt, what major problem would we face in our operating systems?

Solution: User threads might run forever, and never relinquish the CPU.

(c) (4 points) What is a non-blocking operation, on a file, mailbox, or other communication mechanism?

Solution: We perform a normal operation (sending, receiving, or whatever) - but if it normally would block, then it instead returns an error code.

2. (a) (6 points) When you open, read, write, or delete a file, you must perform a syscall into the kernel. Explain why this is - why can't you just access the file yourself, from inside user mode?

Solution: We don't want a rogue user to corrupt the system state, or snoop on data that they are not entitled to access. Therefore, all shared system state needs to be managed by kernel code.

(b) (6 points) Explain the difference between a mutex, a critical section, and a lock.

Solution: A **critical section** is a piece of code that is dangerous - if it races with other critical sections, errors could result.

A **mutex** (mutual exclusion) is one way to protect a critical section - we choose to make it impossible for two CSes to run at the same time.

A **lock** is one possible mechanism for enforcing a mutex.

(c) (8 points) What is an atomic instruction? What new capability does it provide to our programs, more than ordinary load, store, and arithmetic instructions?

Solution: With ordinary load/store, it's possible for other things to happen between the load and store operations, thus leading to race conditions. But with atomic instructions, everything happens at once, with no possibility of other instructions getting between.

3. (a) (10 points) What does the word "spin" in spinlock refer to? What does it tell you about how the lock works?

Solution: The code tries to grab the lock, over and over, forever.

(b) (5 points) In the context of concurrency control systems such as locking, what is the difference between "starvation" and "deadlock?"

Solution: Starvation: We hope that we will make progress in the future, even though we are not making progress now (or things are very slow).
Deadlock: It is impossible to make progress.

(c) (5 points) In class, we discuseed the (very surpising!) fact that the dispatcher must hold interrupts disabled even while it performs the context switch. Explain what sort of problem might arise if we restored interrupts before the switch.

Solution: If we enable interrupts before the context switch, then we might have an interrupt, and call the dispatcher again, which performs its own context switch - meaning that when we finally run again, there's a good chance that the context switch we intended to perform is no longer valid.

4. (a) (8 points) Suppose that you have two concurrent processes, each incrementing the same variable, many times - but they don't use locks or atomic instructions to prevent races:

```
for (int i=0; i<10*1000*1000; i++)
    x++;
```

Now, suppose that a third process is reading the value of `x` over and over, just watching to see how it might change. Explain how it might be possible for `x` to actually go **down** in value (perhaps by a lot), from time to time. (Ignore the fact that `x` might overflow, that doesn't count.)

> **Solution:** Once process can read `x`, and then be context-switched out for an arbitrary amount of time; while it is gone, `x` might be incremented many times.
>
> When the process returns, it increments the value in the register and then saves that value back to `x` - perhaps causing it to go down by a large amount.

(b) (8 points) In our project, we disabled interrupts in our functions, in order to solve race conditions. Explain why this was "as good as" grabbing a lock - provided that we are only running on a single CPU.

> **Solution:** If we disable interrupts, then the only way that the dispatcher can run is if we explicitly choose to call it - thus, no other process (or interrupt handler) can run. Thus, there is no concurrency.

(c) (4 points) Why would disabling interrupts be insufficient, as a replacement for a lock, if we had more than one CPU?

> **Solution:** Because there are two (or more) processes running, one per CPU. So there's concurrency no matter what interrupts occur!

5. Each of the statements below is **False.** Explain why.

(a) (5 points) Because user mode code cannot disable interrupts, the most critical calculations must be done quickly, in order to prevent race conditions with other proceses.

> **Solution:** We must always assume that any operation might take an arbitrary amount of time, because the process could be context-switched out. So speed doesn't solve races!

(b) (5 points) If a process doesn't access any devices (such as reading from the disk), we can accurately predict how fast it will run, because it won't be interrupted.

> **Solution:** Lots of things can change the speed of a process, such as cache state. Plus, an interrupt can still occur, even if this process is not using the device in question!

(c) (5 points) In our Phase 1 scheduler, we had absolute priorities - meaning that the highest priority process always ran. If one process created a child that was higher priority, the parent would not get any time on the CPU until the child dies.

> **Solution:** It does not fail to run "until the child dies;" it fails to run until all higher-priority processes die **or block.** So if the child blocks for any reason, the parent might run again (unless, of course, there were other, also-high-priority processes that wanted to run).

(d) (5 points) When a process calls `quit()`, the kernel will record its exit status and then context-switch to its parent.

> **Solution:** The kernel will context-switch to **some** process, but it doesn't have to be the parent of the dying process.