CSc 452: Principles of Operating Systems
Spring 23 (Lewis)

Test 1

Thu 12 Feb 2023

Solutions

Name: _____ NetID: _____

Question	Points	Score
Page 1	32	
Page 2	32	
All False	36	
Total:	100	

NetID: _____

1. (a) (8 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) (8 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) (8 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.

(d) (8 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.

2. (a) (8 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.

(b) (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.

(c) (8 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.

(d) (8 points) How can we use "try" locks to prevent deadlock? What must we do if a try lock fails?

Solution: If a trylock fails, you must release all locks and start over - so that it doesn't block while holding a lock.

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

(a) (18 points) This question is a two-parter. In this question, we suppose that there is a variable `x` in global memory, which is initialized to 0. You create two processes, and let both run to completion; each one runs the following C code:

```
void foo() {
    int tmp = x;
    for (int i=0; i<25; i++)
        tmp++;
    x = tmp;
}
```

First False Statement:
There is a race condition between the two processes, because they both modify the `tmp` variable at the same time.

> **Solution:** The `tmp` variables are local variables, and thus stored on the stack. Thus, there can't be a race, because the two threads don't use the same variable.

Second False Statement:
Because we copy the value of `x` into `tmp`, we no longer have a race condition between the threads; thus, the value of `x` is guaranteed to be 50 when both processes have completed.

> **Solution:** We still have a race on the `x` variable. Imagine a scenario where both threads read `x` at roughly the same time; they would run in parallel, and then **both** would write the same value (25) back to `x`.

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

(b) (9 points) In the Dining Philosophers problem, there is a special case: if there are an even number of philosphers, we can reach a special type of deadlock, where half of them are eating, and the others are all stuck.

> **Solution:** This is not deadlock! The philosophers that are eating are not currently blocked; instead, they will eventually finish their meal, and relinquish their utensils. This will unblock their neighbors.

(c) (9 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 block until the child dies.

> **Solution:** It does not block until the child dies; it blocks 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).