CSC 256/CSC 456
Midterm Exam
October 16th, 2012
Total: 60 points

This exam is closed book, closed notes. Please be concise in, but always explain, your answers. An answer without an explanation will not receive credit. While I expect more than just the name of a particular technique if that is the answer to a question, extraneous remarks may count against you. On the flip side, exceptionally good answers may receive extra credit.

There are a total of 60 points (budgeting 1 point per minute will allow you to pace yourself, with a little extra time to spend on questions where you need a little more thought - some questions will take more or less time, but you should use this estimate to keep on track - a good test taking strategy is to do a quick scan and start with the questions you feel most confident about).

While I have tried to make the questions as clear as possible, in the event that you need to make an assumption to proceed, please explicitly write down your assumption/s. In the interest of fairness, the proctor (Brandon Shroyer) has been instructed not to attempt to answer any questions during the exam.

1. [10 points] Short answers:

   (a) [6 points] Which of the following instructions should be privileged and only allowed to execute in kernel mode (explain your answer)?

   - Load a value from a memory address to a general-purpose register.
   - Load a value from a memory address to an entry in the translation look-aside buffer (TLB).
   - Set a new value in the program counter (PC) register.
   - Turn off interrupts.
   - Call a subroutine — pushes the return address onto the stack and jumps to the subroutine.
   - Halt the CPU.

   (b) [2 points] Most modern operating systems use a preemptive, priority-based CPU scheduling algorithm. Explain how preemption is achieved, i.e., what hardware support is required in order to preempt processes?

   (c) [2 points] Translation-lookaside buffers (TLBs) are a hardware cache of page table entries in modern processors used to speed up virtual to physical memory address translation. In the hopefully common case of a hit in the TLB, a memory access proceeds without incurring additional translation overhead. Assuming a single-level page table as discussed in class, how many additional memory accesses would be needed on a miss in the TLB before the programmer-specified memory access can proceed?

2. [25 points] Processes, threads, and scheduling:

   (a) [5 points] An application using threads rather than processes for concurrency has the advantage of lower context switch overheads due to eliminating address space switching (e.g., TLB flushing, updating page table registers). Describe at least two potential issues with the thread model (you may assume the use of kernel-level rather than user-level threads) from the kernel's or user's perspective.

(b) [5 points] Linux kernel versions 2.6 and beyond provide the option to compile the kernel in *preemptible* mode, i.e., a kernel in which a process switch may occur at any point, even when a process is executing in kernel mode.

- Why is this considered an important property?
- What restructuring might be required in the kernel in order to make this possible?

(c) [7 points] Consider a server with two clients running independent multi-threaded databases. Client 1 has paid for 75% of the available CPU resources, while Client 2 has paid for 25%. How would you ensure that each client got their requested (paid for) share of CPU resource?

(d) [8 points] Scheduling: Suppose a processor uses a prioritized round robin scheduling policy. New processes are assigned an initial quantum of length q. Whenever a process uses its entire quantum without blocking, its new quantum is set to twice its current quantum. If a process blocks before its quantum expires, its new quantum is reset to q. For the purposes of this question, assume that every process requires a finite total amount of CPU time. For each of the policies below, is starvation possible? Why or why not? If starvation is possible, what could you do to prevent it? What are the implications of the policy for throughput, fairness, and response time?

i. [4 points] The scheduler gives higher priority to processes that have larger quanta.

ii. [4 points] The scheduler gives higher priority to processes that have smaller quanta.

3. [25 points] Synchronization and deadlocks:

(a) [10 points]

Consider a data type called `bankaccount`, defined by the following:

```
typedef struct {
    unsigned SSN;
    float balance;
    semaphore mutex;
} bankaccount;
```

Suppose we have a set of such accounts. Suppose further that a function FindAccount exists that, when called with an unsigned argument, returns a pointer to the record containing an SSN (the key) equal to the argument, and NULL if no such record exists. You may assume that at most one record (bankaccount) exists for each unique SSN.

The following procedure named `Debit` takes two SSNs, subtracts the specified amount (a float argument) from the first account if the balance is greater than or equal to the requested amount, and adds it to the second.

```
int Debit(unsigned SSN1, unsigned SSN2) {
    bankaccount *f1, *f2;

    f1 = FindAccount(SSN1);
    f2 = FindAccount(SSN2);
    if ((f1 != NULL) and (f2 != NULL)) {
        if (f1->balance >= amount) {
            f1->balance -= amount;
            f2->balance += amount;
            return SUCCESS;
        }
    }
    return FAILURE;
}
```

The above procedure functions correctly in the absence of concurrency. Using the semaphores associated with each record, write a version of `Debit` that works correctly in the presence of concurrent processes making calls to `Debit`. Your solution should admit high concurrency while making the net effect of all the Debit calls the same as some sequence of sequential calls. Make sure to consider and avoid the possibility of deadlock.

(b) [5 points] Monitors allow safe sharing of abstract data types among concurrent processes by ensuring that only one process can execute (any code that manipulates the shared data) within the monitor at any given time. In order to allow a process to wait on a condition from within a monitor, condition variables are implemented as follows: when p_0 signals a condition while p_1 is waiting on it, p_0 continues to execute; p_1 is merely woken up from the corresponding wait and allowed to compete for re-entry to the monitor when p_0 exits the monitor. What implications does this implementation have on the semantics someone using this style of monitor can assume?

(c) [10 points] One form of synchronization that Linux uses for reader-writer synchronization is *seqlocks*, with the goal of prioritizing writes and speeding up reads in the case of no contention for the lock. *seqlocks* have a spinlock and a counter associated with them. Writers must acquire the spinlock prior to writing, and increment the counter once the lock is held. Writers also increment the sequence counter again prior to releasing the spinlock. In other words, the counter (or sequence number) is always *odd* when a writer is inside its critical section, and *even* when there is no writer in the critical section. Readers merely read the counter prior to entering and leaving their critical sections, and repeat/redo the critical section if either the counter value is odd (a writer is in its critical section) or if the current counter value is different from the value read during entry.

A linux kernel develepor decides to simplify the *seqlock* and replaces the counter with a boolean variable, set to true (1) when a writer enters its critical section and reset to false (0) when the writer leaves its critical section. To the developer's surprise, the kernel began to occasionally malfunction, causing the system to crash.

- What do you think the problem is in the simplification that might cause the kernel to malfunction?
- How does the use of a counter fix it and is there any scenario under which even the counter might fail to properly synchronize the readers and writers (i.e., what are the assumptions under which the counter-based implementation works)?