EECS 343: Homework 3

Threads and Synchronization - Solution

Fall 2014

Important Dates

Out: November 7, 2014.

Due: November 14, 2014, 11:59PM CDT.

Submitting your assignment: Please use the course submission site. There is a link to it from the class site. **Submit only ASCII text files.**

Problems

1. In class we listed the register set as a per-thread rather than per-process item. Why? After all, the machine has only one set of registers.

   Answer: *When a thread is stopped, it has values in the registered. They must be saved, just as when the process is stopped the registers must be saved. Multiprogramming threads is no different that multiprogramming processes, so each thread needs its own register save area.*

2. If a multithreaded process forks, a problem occurs if the child gets copies of all the parent's threads. Suppose that one of the original threads was waiting For keyboard input. Now two threads are waiting for keyboard input, one in each process. Does this problem ever occur in single-threaded processes?

   Answer: *No. If a single-threaded process is blocked on the keyboard, it cannot fork.*

3. Can two threads in the same process synchronize using a kernel semaphore if the threads are implemented by the kernel? What if they are implemented in user space? Assume that no threads in any other processes have access to the semaphores.

   Answer: *With kernel threads, a thread can block on a semaphore and the kernel can run some other thread in the same process. Consequently, there is no problem using semaphores. With user-level threads, when one thread blocks on a semaphore, the kernel thinks the entire process is blocked and does not run it ever again. Consequently, the process fails.*

4. Show how counting semaphores can be implemented using only binary semaphores and ordinary machine instructions.

Answer: *Associated with each counting semaphore are two binary semaphores, M, used for mutual exclusion, and B, used for blocking. Also associated with each counting semaphore is a counter that holds the number of ups minus the number of downs, and a list of processes blocked on that semaphore. To implement down, a process first gains exclusive access to the semaphores, counter, and list by doing a down on M. It then decrements the counter. If it is zero or more, it just does an up on M and exits. If M is negative, the process is put on the list of blocked processes. Then an up is done on M and a down is done on B to block the process. To implement up, first M is downed to get mutual exclusion, and then the counter is incremented. If it is more than zero, no one was blocked, so all that needs to be done is to up M. If, however, the counter is now negative or zero, some process must be removed from the list. Finally, an up is done on B and M in that order.*

5. Consider a system in which threads are implemented entirely in user space, with the run-time system getting a clock interrupt once a second. Suppose that a clock interrupt occurs while some thread is executing in the run-time system. What problem might occur? Can you suggest a way to solve it?

Answer: *It could happen that the runtime system is precisely at the point of blocking or unblocking a thread, and is busy manipulating the scheduling queues. This would be a very inopportune moment for the clock interrupt handler to begin inspecting those queues to see if it was time to do thread switching, since they might be in an inconsistent state. One solution is to set a flag when the run-time system is entered. The clock handler would see this and set its own flag, then return. When the runtime system finished, it would check the clock flag, see that a clock interrupt occurred, and now run the clock handler.*