OS161 Concurrent Math

[Roll - 61][Roll - 09][Roll - 11]

April 18, 2017

1. What happens to a thread when it exits (i.e., calls thread_exit())? What about when it sleeps?

   Answer - When a thread exists, it ensures the stack isnt mangled, removes its virtual memory space and destroys it, decrements the counter of whatever vnode it may be pointing at. Puts itself into a zombie state. S_ZOMB, and preps itself to panic if it ever runs again before it dies. When its sleep, it makes sure its not in an interrupt handler, yields control to the next thread, enters the S_SLEEP state, and only starts taking control again when wakeup() is called on its address.

2. What function(s) handle(s) a context switch?

   Answer - There are two functions that handles a context switch. They are - mi_switch : high level, machine independent context switch function. Md_switch : machine independent code that actually does the context switch.

3. How many thread states are there? What are they?

   Answer - There are four thread states. They are - S_RUN,S_READY,S_SLEEP,S_ZOMBIE. They define whether the thread is running or ready or sleeping or became a zombie.

4. What does it mean to turn interrupts off? How is this accomplished? Why is it important to turn off interrupts in the thread subsystem code?

   Answer - If Interrupts are turned off, then even if an interrupt is signaled the interrupt handlers is not called until interrupts are turned back on. Interrupts are turned off using splhigh(set priority level high) function and turned on using spl0(set priority level zero). The priority level can also be set to their intermediate level using splx() function. Turning off interrupts for thread operation is necessary to ensure that these operations complete successfully and completely and arent broken during mid-operation. For example things can get pretty badly if the scheduler interrupt us in the middle of a context switch and try to start executing a thread that wasnt finish setting up its stack.

5. What happens when a thread wakes up another thread? How does a sleeping thread get to run again?

   Answer - It removes the sleeping thread from the queue and calls make runnable on the thread which currently adds it to the end of run queue. The thread gets to run again when mi_switch is called and the thread is returned by the scheduler.

6. What function is responsible for choosing the next thread to run?

   Answer - Struct thread * scheduler(void)

7. How does that function pick the next thread?

   Answer - It uses round robin queue that schedules each thread in the queue in equal time slice without priorities.

8. What role does the hardware timer play in scheduling? What hardware independent function is called on a timer interrupt?

   Answer - The interrupt handler for the hardware timer calls hardclock . The method hardclock finishes by calling thread_yield every time it is run forcing a context switch.

9. What is a wait channel? Describe how wchan_sleep() and wchan_wakeone() are used to implement semaphores.

   Answer - Wait channels are a low level synchronization primitive. They are often implemented as a part of thread scheduler. A wchan is an object(sometimes just a memory address) that a thread can sleep on. It sleeps until another thread wakes it. wchan_sleep () put the current thread to sleep and wchan_wakone() wakes one thread sleeping on the wchan. The first function suspends the current thread until the semaphore count is greater than zero. wchan_wakeone() can particularly wakes up one thread depending on the semaphore.

10. Why does the lock API in OS/161 provide lock_do_i_hold(), but not lock_get_holder()?

    Answer - lock_get_holder could introduce the case where the holder of the lock change when we want to use the return value. lock_do_i_hold() will perform the verification job in a more atomic fashion.

11. Here are code samples for two threads that use binary semaphores. Give a sequence of execution and context switches in which these two threads can deadlock.

    Answer - in me() function when it takes p(mutex) and then if the scheduler switches and then if you() function takes p(data) , the the deadlock occurs, because both of the functions cant work from second line.

12. Propose a change to one or both of them that makes deadlock impossible. What general principle do the original threads violate that causes them to deadlock?

Answer - Actually here deadlock can occur. Because if this case , suppose laurel() takes acquire of file1, lock acquire(file2) then hardy() will wait as it cant acquire file1, then laurel release file1, releases mutex and suddenly hardy() acquire file1 and wait for acquiring file2 because laurel() still holds it , then if laurel() again wants to acquire file1 it waits. And deadlock begins, because laurel() never gets file1() it belongs to hardy() and hardy() cant get file2 because it was not released by laurel().

Solution 1: void laurel() lock_acquire(mutex); /* do something */

lock_acquire(file1); /* write to file 1 */

lock_acquire(file2); /* write to file 2 */

lock_release(file1); lock_release(file2); /*****First change ****/ lock_release(mutex);

/* do something */

lock_acquire(file1);

lock_acquire(file2); /*second change*/ /* read from file 1 */ /* write to file 2 */

lock_release(file2); lock_release(file1);

void hardy() /* do stuff */ lock_acquire(file1); /* read from file 1 */

lock_acquire(file2); /* write to file 2 */

lock_release(file1); lock_release(file2);

lock_acquire(mutex); /* do something */ lock_acquire(file1); /* write to file 1 */ lock_release(file1); lock_release(mutex);

Solution2:

void laurel() lock_acquire(mutex); /**First change*/ /* do something */

lock_acquire(file1); /* write to file 1 */

lock_acquire(file2); /* write to file 2 */

lock_release(file1); /**Second change**/

/* do something */

lock_acquire(file1);

/* read from file 1 */ /* write to file 2 */

lock_release(file2); lock_release(file1); lock_release(mutex); /***third Change*/

void hardy() /* do stuff */ lock_acquire(mutex); /**First change*/ lock_acquire(file1); /* read from file 1 */

lock_acquire(file2); /* write to file 2 */

lock_release(file1); lock_release(file2);

lock_acquire(mutex); /* do something */ lock_acquire(file1); /* write to file 1 */ lock_release(file1); lock_release(mutex);

13. The thread subsystem in OS/161 uses a linked list of threads to manage some of its state (kern/thread/threadlist.c). This structure is not synchronised. Why not? Under what circumstances should you use a synchronised linked list?

    Answer - The runqueue used by the thread scheduler subsystem is only accessed by a single scheduler thread, so does not need any synchronization primitives. We should use a synchronized queue structure for any queue that multiple threads could access simultaneously.