ECE 3140 / CS 3420
Matthew Daniel (mrd89)
Xinyi Yang (xy98)
Lab #4
2019-March-27

Objective

The main purpose of this lab is to give us a better understanding of how an operating system locks and unlocks programs for the critical section. To do this, we used the same program queue and switching framework from Lab 3 but with these critical sections that require locking of the processor.

Main Program Report

Lock

To incorporate lock.c into our design, we need to initialize some lock with theD variables "state "and "blocked_processqueue". These are set to be 0 (not blocked) and empty for the queue. Within the locking program we initiated a few programs; one of which is the enqueue function. When the program is called, it checks if the queue is empty. If this is the case, then the queue is only the current process proc. Else, the program adds the proc to the end of the blocked queue and finally adds a null slot for the upcoming program in the queue. Naturally the program needs to have a dequeue function which removes the program from the queue. First it checks for a NULL queue, then moves the next process onto the queue and returns the process.

For the actual locking function, the first step is to disable interrupts which are on the PIT channel 0. The purpose of this is so two programs do not try to lock at the same exact time which would result in both running in critical section or having other issues. Next the lock checks if the lock is already taken by any other program. If this is the case, the current process is moved to the blocked process queue. Then the current process struct is updated to tell the processor that it was blocked then calls the process_blocked() function. If the lock was unused, all the program needs to do is set the state to 1 (blocked) and enables interrupts again.

In order to unlock a lock, the program again disables interrupts to prevent process switching. It needs to check if there is another processor in the blocked queue, but if the queue is NULL, then it sets the lock to 0, enables interrupts, and exits. If there are processes in the queue, then the unlock program sets the lock to 0, removes itself from the block queue, and uses the push_tail_process function to add the process to the process_queue. Finally to exit it enables interrupts.

Process

We built from the process solution posted on Piazza to ensure the program would not have any issues from lab 3. The change made was in the process_select function inside the" if (cursp)" statement to check if the current process was blocked. If the process was not blocked, the program can call the push_tail_process like normal. This implementation was fairly straight forward after the other programs were created.

Code Testing

The first test case given is Lab4_l0 which has one process "p1". This process runs 11 times through a critical section that toggles the red and blue LED once per critical section. The main section creates two instances of p1 via process_create and then starts the program. If the program is successful it will run 22 red and blue flashes and then a green LED on. We were able to get the code to successfully complete!

The second given test case uses the idea of readers and writers who need the critical section. This is a much more in depth check using two instances of lock (w and r) . The main section creates a writer, three readers, and another writer in that order. When the writer runs successfully, it will make the red LED blink once and when the reader runs successfully, it will call p1 which toggles the blue led 5 times. When we tested our code, the board output 1 red flash, 15 blue flashes that were not always the same delay, 1 red flash, and finally a solid green indicating all the programs finished successfully!

For the additional test cases, we created two processes: p1 and p2. The functionality of p1 is to loop three times of a critical section which locks a, toggles the red LED, and then unlocks and exits. P2 uses lock b to run its critical section and toggles the blue LED. The purpose of this test is to check how the program can handle running two locks at the same time. Essentially there are critical sections that are independent of each other and are allowed to run at the same time. The main section runs an instance of p2, three of p1, and another of p2 in that order; if all finishes successfully, the program ends with a green light. This test case passed with the red and blue flashing simultaneously some of the time but keeping each of the red or blue flashes separate respectfully.

For our final test, we created three processes p1, p2, and p3 which toggle the red LED once, blue LED 6 times, and the red LED 3 times respectfully. Each uses the "a" lock when calling their led toggle. This test is to see how the processor handles when there are more than two distinct processes seeking the lock. The main function calls p1,p2, two instances of p1, and p3. As expected, the program runs correctly with no issues with the LED order and ends with the green light indicating there were no major issues with our design!

Discussion

This lab was able to incorporate our knowledge of locks, interrupts, and the reader and writer analogy from class into a physical program. Having to write an actual locking program gives us a much deeper understanding of how a real operating system works and issues that could arise when trying to design a new system. There are so many methods of implementing these lock features, but with the help of the given programs and lab manual, we were happy with how the program and test cases were programmed.

Work Distribution

Similar to Lab 3, we met Saturday to go over the basic design of the code so we were on the same page. Much of this time was making sure we understood the idea behind the locks for out specific design. The lock.h file was very helpful by giving us a better idea of exactly how it was set up. Once we understood the purpose and an initial approach for the lab we began coding.

We both decided to write our own code for this lab separately due to difficulties in finding good times to meet since we both had other exams in addition to this due. Luckily we started early which helped both of us tremendously. After out initial meetup on Saturday, we went to work and met back Wednesday afternoon to discuss our code. We had a similar idea in our code, but we both agreed to go with Xinyi's code moving forward. During this time we discussed further testing methods that could cause major issues with any locking design. These ideas were similar to what we discussed where we needed to work through the "worst possible case" with our processes. Finally we met up Thursday to test the board with the two given cases and the tests we came up with during the day!

For collaboration as laid out above, we met in the early stage after we both had a chance to read the files and manual. From there we made a basic game plan on how to complete the lab, and then worked individually for a few days. We kept in communication via text on progress updates and any struggles we had completing a task. To share the code, we used email instead of CMS to keep all the uVision files exactly the same.