Jackson Paull

JHP2539

28 February 2025

ECE 381L - RTOS

Lab 3 Report

1. **OBJECTIVES**

This lab was focused on the implementation of blocking semaphores, background threads, and a priority scheduler. I focused on an abstract implementation with minimal reliance on assumptions (e.g. AddPeriodicThread only being called once). The main objective shared between these two tasks is an effective and efficient virtualization of CPU resources, minimizing overhead of the scheduler while ensuring mutual exclusion and priority.

1. **HARDWARE DESIGN**

None for this lab

1. **SOFTWARE DESIGN** - *Note: See GitHub, tag ‘lab3-release’ for source code files.*

Jitter Main Program

I slightly modified testmain6.

Each thread selects a ‘JitterTracker’ from the OS at launch and sets the expected period.

During runtime, whenever the thread launches it calls OS\_Jitter and passes the ID of the tracker. Jitter tracking is otherwise identical to Lab 2.

Blocking Semaphores Main Program

I used testmain5 and direct memory inspection to test blocking semaphores.

Testmain5 passed as expected with priority and no priority.

Blocking/Priority RTOS

Priority is maintained through priority queues, except at the scheduler where it is maintained by keeping a separate circular list for all priority levels. (see fig 1)

When a thread is blocked, it is removed from the scheduler and placed into a semaphore list until the semaphore is signaled.

When the scheduler is called, it finds the highest (lowest) priority level with a non-empty list and performs round-robin scheduling on that list.

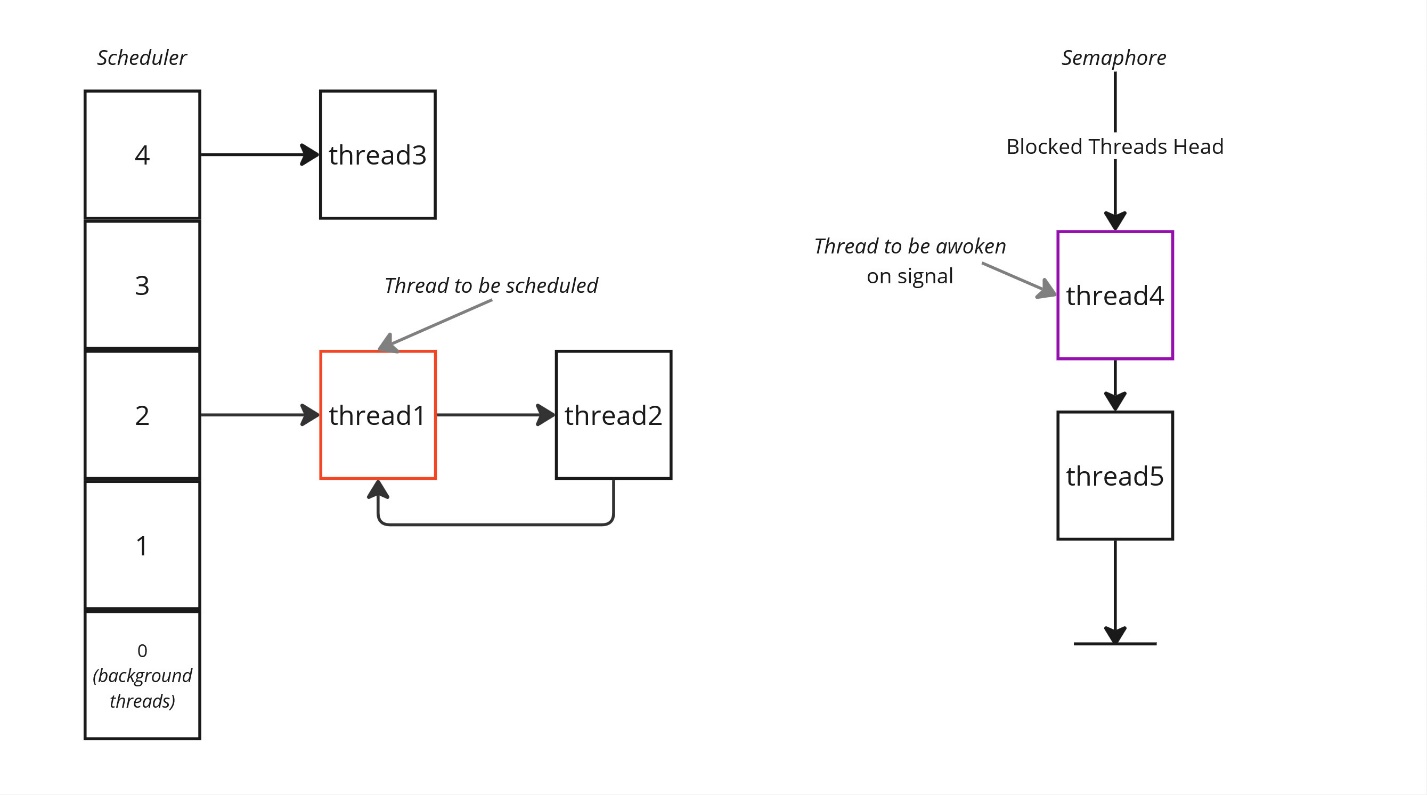


Fig 1. Depiction of priority scheduler and blocking semaphores

1. **MEASUREMENT DATA**

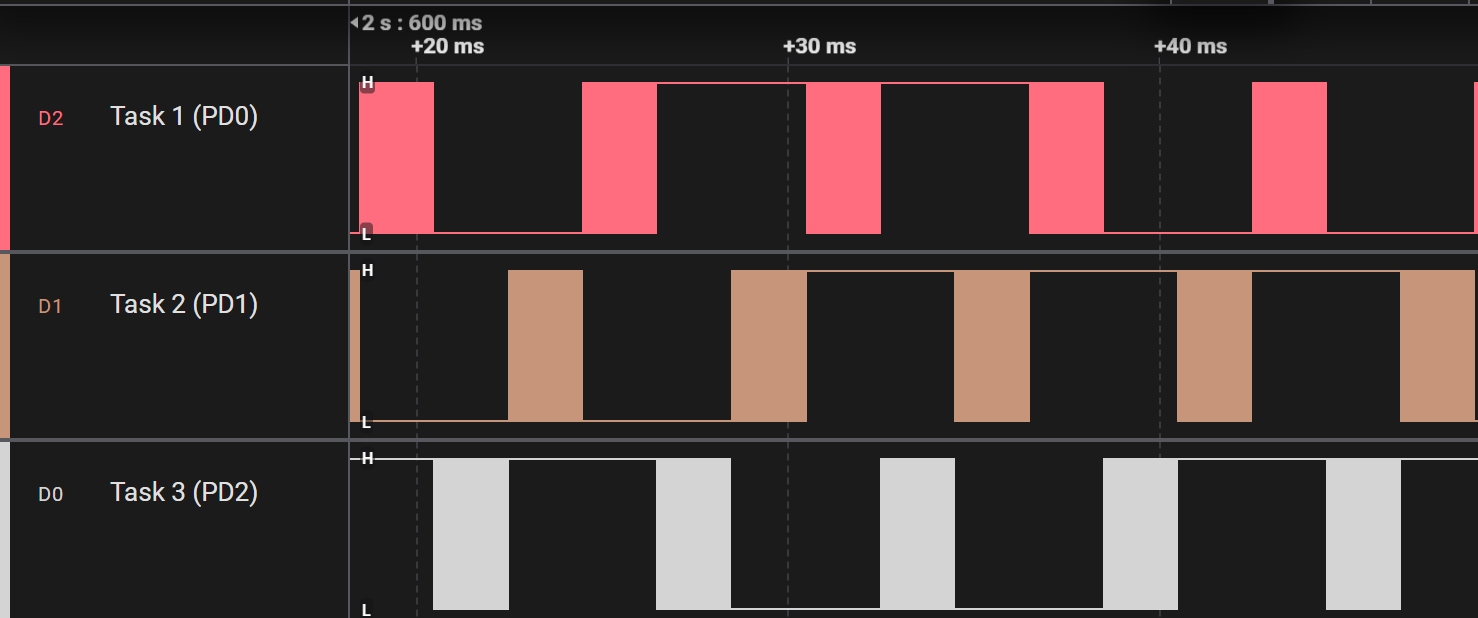


Fig 2. Round Robin scheduling of three threads at equal priority.

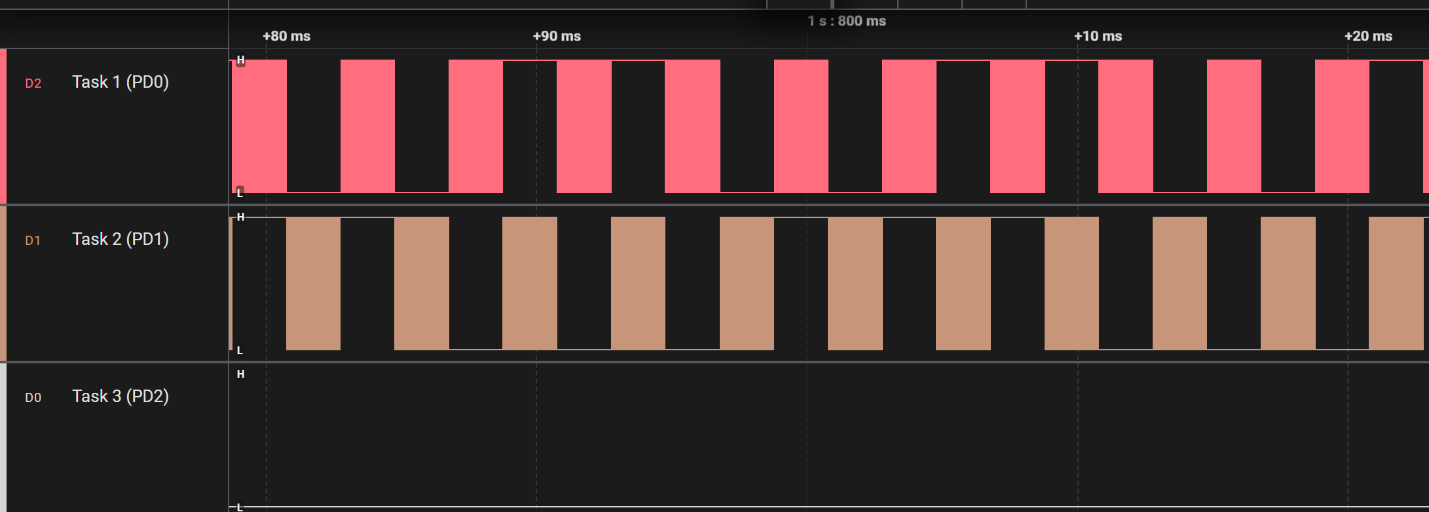


Fig 3. Round Robin scheduling of 3 threads with one thread at low priority.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | Blocking Priority | | | Blocking RR | | | SpinLock RR | | | SpinLock Cooperative | | |
| FIFO Size | **Tslice (ms)** | **Data Lost** | **Jitter (us)** | **CPU Util** | **Data Lost** | **Jitter (us)** | **CPU Util** | **Data Lost** | **Jitter (us)** | **CPU Util** | **Data Lost** | **Jitter (us)** | **CPU Util** |
| 4 | 2 | 1788 | 16 | 33.08 | 1788 | 16 | 33.08 | 2671 | 25.3 | 100 | 0 | 28.0 | 98.92 |
| 32 | 2 | 0 | 16 | 40.3 | 0 | 16 | 40.3 | 559 | 25,3 | 100 | 0 | 28.1 | 98.92 |
| 32 | 1 | 0 | 14.5 | 41.65 | 0 | 14.5 | 41.65 | 348 | 24.9 | 99.99 | 0 | 26.3 | 98.97 |
| 32 | 10 | 0 | 15.9 | 40.29 | 0 | 15.9 | 40.29 | 545 | 25.3 | 100 | 0 | 27.8 | 98.93 |

Table 1. System Profiling

Note: Blocking RR/Priority are equivalent in my implementation for realmain, as periodic threads are triggered in the order they are added (which is also their priority order).

1. **ANALYSIS AND DISCUSSION**
2. *How would your implementation of* OS\_AddPeriodicThread *be different if there were 10 background threads?*
   1. My implementation wouldn’t change (outside of increasing a macro in a header file). The timer interrupt repeatedly executes as a one-shot timer with a period based on which task needs to execute soonest, allowing for multiple tasks to be scheduled simultaneously with priority, and maintaining priority for switch tasks and periodic tasks. (i.e. a switch task with a priority of 2 will always execute after a periodic task with priority 1 regardless of the priority of interrupts in the NVIC)
3. *How would your implementation of blocking semaphores be different if there were 100 foreground threads?*
   1. My implementation wouldn’t change. Assuming all the threads and their stacks fit in memory, my implementation can operate on an arbitrary number of threads. When a thread is blocked, it consumes exactly the same amount of memory as when it is scheduled, the only difference is whether the scheduler or the blocking resource is holding the TCB entry.
4. *How would your implementation of the priority scheduler be different if there were 100 foreground threads?*
   1. My implementation wouldn’t change. Assuming all the threads and their stacks fit in memory, my implementation can operate on an arbitrary number of threads (with a fixed resolution of priority levels) as it is based on linked list pointers in the TCB structure.
5. *What happens to your OS if all the threads are blocked? If your OS would crash, describe exactly what the OS does? What happens to your OS if all the threads are sleeping? If your OS would crash, describe exactly what the OS does? If you answered crash to either or both, explain how you might change your OS to prevent the crash.*
   1. All threads being blocked is equivalent to all threads sleeping for my RTOS. In both of these cases, there are no threads added to the scheduler. The scheduler would therefore be unable to select a thread to schedule and would select a backup OS thread to run, which in this case is an infinite loop but could in principle be changed to some crash handling code (for instance if no threads are sleeping and all threads are blocked, logs could be dumped) or be changed to a thread which is actively scheduled with other threads and never returns (such as the interpreter).
6. *What happens to your OS if one of the foreground threads returns? E.g., what if you added this foreground*

**void BadThread(void){ int i;**

**for(i=0; i<100; i++){};**

**return;**

**}**

*What should your OS have done in this case? Do not implement it, rather, with one sentence, say what the OS should have done?*

* 1. My OS would not crash. When setting up the stack, foreground threads have the LR set to point to OS\_KILL, meaning that they will automatically die when they return. Background threads have a special version of OS kill which does not return the TCB to the inactive pool so that it is permanently allocated for the background task even when it is not running.

1. *What are the advantages of spinlock semaphores over blocking semaphores? What are the advantages of blocking semaphores over spinlock?*
   1. Spinlock semaphores are easy to implement. Blocking semaphores allow for priority and improved CPU utilization when properly linked with the scheduler, at the cost of a more complex system which could be harder to debug.
2. *Consider the case where thread T1 interrupts thread T2, and we are attempting to verify that the system has no critical sections. Assume that T2 has* **n** *interruptible locations (between instructions). The location where T2 gets interrupted is random and each location is equally likely to be interrupted (in other words, interrupt locations are uniformly distributed). Let* **k** *be the number of times T1 interrupts T2. What is the probability that one specific location in T2 was never interrupted? Bonus question (extra credit): what is the probability that all locations were interrupted at least once?*
   * 1. This result follows from the inclusion-exclusion principle.
     2. Further note that , hence the expression in the sum