Lab 3: Real-Time Kernel

You need to spend time crawling alone through shadows to truly appreciate what it is to stand in the sun.

18-349: Introduction to Embedded Systems

Checkpoint 1 Due: Tuesday March 27, 2018, 3:00pm
Lab 3 Due: Tuesday April 10, 2018, 3:00pm
Lab 3 Demo By: Friday April 13, 2018, 3:00pm

Contents

1 Introduction and Overview

1.1 Goal

In this lab, you will develop a real-time kernel capable of admission control, task scheduling and synchronization. The lab is divided into three main parts: (1) context switching and task management, (2) fixed priority rate-monotonic scheduling with admission control and (3) real-time synchronization using the priority ceiling protocol.

1.2 Task List

1. Get the starter code from GitHub classroom (see Section 2)
2. Implement a context swap with IRQ interrupts (see Section 4)
   (a) Implement context switching in your IRQ handler. It's best to see if you can swap out and back the same thread first. This means that you will save and restore the user running context at each IRQ interrupt. This will test that your context switch logic is working properly.
   (b) Use GDB to put breakpoints at your irq handler and step through your code to make sure the proper context is saved and restored. Check cpsr to verify that your user mode is restored properly.
   (c) Next work on the thread management functionality. Setup the thread data structures and implement `thread_init()` and `thread_create()`
   (d) Create a simple scheduler that chooses between 2 tasks (threads).
   (e) Create a simple user program that has 2 functions each with a printf statement. This could be similar to Test 1.
   (f) Combine your context swap with your scheduler. Your kernel should be able to switch between 2 threads at each IRQ interrupts.
   (g) With GDB, put breakpoints at your irq handler and step through your code to make sure the proper context of the 2 threads is saved and restored. Check cpsr to verify that your user mode is restored properly.
3. Implement Rate Monotonic Scheduling (see Section 5)
   (a) Add the UB test to `scheduler_start()` to check schedulability.
   (b) Implement a rate-monotonic scheduler that picks the highest priority thread to run from the runnable pool.
   (c) Implement `wait_until_next_period()` that put the current running thread to waiting pool and schedule it to run in the next period.
   (d) Write a test user program with has multiple functions each with a printf statement and `wait_until_next_period()`. Create threads that run those functions. Verify that the threads are running in the order of priority you created them with GDB.
   (e) Implement `spin_wait`.
   (f) Implement checks to determine if threads overrun their computation time.
4. Turn in checkpoint 1 (see Section 6)
5. Implement mutexes (see Section 7.1)
   (a) Implement `mutex_init()`, `mutex_lock()`, and `mutex_unlock()`.
   (b) Test your mutex code with Test 5.
6. Implement Priority Ceiling Protocol (see Section 7.2)
7. Demo to TAs
8. Submit to GitHub (see Section 9). Make sure your code has met style standards including Doxygen documentation!

1.3 Grading

Your kernel should meet the following requirements:

Task	Points
Context swap with IRQ interrupts	40
Rate-Monotonic Scheduling: Tests 1-4 and 4-1	50
Submitting Checkpoint 1	10
Real-Time Synchronization: Tests 5-6	30
Nested Mutexes: Tests 7-9	20
PCP Implementation: Tests 10-13	30
Style (including Doxygen documentation, following submission protocols)	20
TOTAL	200 pts

2 Starter Code

Use the magic link to create a repo for lab 3. Either create a new team if your partner hasn't made one, or join your partner's team upon accepting the assignment.

This lab will build from your previous IRQ and SWI implementations. If your IRQ and/or SWI are not working correctly please fix them before working on the context switch and task management. You will need to merge your code from lab 2 into this lab's starter code as needed.

3 System Call Interface

We will now use the system calls you built in Lab 2 as a basis for running multi-threaded programs.

3.1 Differences from Lab 2

We have added threading-related system call prototypes in `newlib/349include/syscall_thread.h`. Matching assembly function stubs have been added to `newlib/349include/swi_stubs.S`. Similar to what you did in Lab 2, we would like you to map them to the appropriate SWI numbers from `349libk/include/swi_num.h`.

Since your scheduler will need to user your timer IRQ it cannot coexist with the Lab 2 implementation of `servo_set()` and `servo_enable()`. We would like you to gracefully degrade these functions by making the kernel implementation always return an error. In the future you could implement servo controller using your real time kernel, but we are not asking you to do that in this lab.

3.2 Implementation Notes

For technical details on the new system call prototypes in `newlib/349include/syscall_thread.h` from the user program prospective refer to the doxygen documentation. To help build intuition on what these system calls will look like in your kernel we have include a few notes below.

`int thread_init(thread_fn idle_fn)`

This call gives the kernel a time to initialize the threading data structures. You may or may not need to do that with your implementation. You must save the `idle_fn()` for use when there are no other threads to run.

`int thread_create(thread_fn fn, uint32_t *stack_start, unsigned int prio, unsigned int C, unsigned int T)`

A core function to the thread library is adding new threads to the scheduler. To make this lab easier (but also a common practice in embedded RTOSs) we have a user initialization `main()` function that can build the thread functions and stack you want to schedule. For a usage example look over the provided test cases.

`stack_start` is the first usable element on a downward growing ARM stack. Note that this is the opposite end of an array as we normally pass in as a pointer. For an example, refer to the stack creation in any of our user test programs.

We recommend that you have simple error checking for the passed in `fn` and `stack_start` arguments to save you debugging time.

`int mutex_init(mutex_t *mutex, unsigned int max_prio)`

Since the user program provides the pointer to the mutex structure memory, we give the kernel a chance to set any default values here. Remember to set your `max_prio` to the highest priority (lowest number) of any task that could use this mutex.

`void mutex_lock(mutex_t *mutex)`

Locking the mutex should not return until the thread has exclusive access to the mutex. Since we're using priority ceiling protocol (emulation mode), we should never find the mutex "locked." Instead, we are elevated to a priority where anybody else asking for the mutex could not possibly run.

`void mutex_unlock(mutex_t *mutex)`

A thread is required to unlock the mutex when it is done with the protected resource. You do not have to worry about rouge threads that capture the mutex forever. Upon unlocking you don't need to call the scheduler immediately to find a higher priority task that was not run while the mutex was held. It is ok to wait for the scheduler to run on the next tick.

`void wait_until_next_period(void)`

When a function is done with the work it needs to do in this period it can call `wait_until_next_period()` to yield to another thread or the idle thread. To reduce race conditions, we recommend that you do not have this syscall perform a context switch. Rather, it should spin in the kernel until the next scheduler call (by IRQ) happens.

`unsigned int get_time(void)`

Since your kernel and scheduler keep their internal time in milliseconds, this implementation should be straightforward. The user tests use this to make your output more readable with timestamps. We may also use this to time user actions later.

`int scheduler_start(void)`

This call tells the kernel that the user initializer program is done setting up tasks and to start running them. The kernel should run the UB admittance test described in the Scheduler section to determine if the given task set is schedulable.

`unsigned int get_priority(void)`

This call returns the current effective priority of the calling thread. As you'll need to track this to implement PCP, this should be a very straightforward implementation.

`void spin_wait(unsigned ms)`

A thread that calls this function should spin wait in kernel space for the given number of milliseconds. This does not include time where other threads are running, nor time when the calling thread is blocked (in the waiting pool). As nearly every test uses this system call, it would be prudent to think about implementation early. Threads in a call to `spin_wait()` are not blocked, they are running without performing any useful work.

4 Context Switching

A context switch is the process of storing and restoring the state of a thread or process so that execution can be resumed at a later time. In this part of the lab, you will implement the `thread_create`, `thread_init`, and `scheduler_start` system calls. In lab 2, user programs were "single threaded". Once the kernel called `enter_user_mode`, the user program ran and hung infinitely if it ever returned. Our new kernel will call `enter_user_mode` and the user mode `main` will then register threads with the kernel and then call `scheduler_start` to begin running the threads you registered. At this point, the user mode `main` function no longer needs to be restored and the threads you registered with the kernel will be the only user mode contexts running.

The user mode `main` should do the following:

1. call `thread_init`.
2. call `thread_create` for the number of threads you want to create.
3. call `scheduler_start`.

To implement threads in our kernel, we need a few things to support these system calls. First, we need to design a TCB (task control block).

4.1 TCBs and Kernel Stacks

For every task that the kernel is instructed to run, the kernel maintains an in-kernel data structure that describes the task's current state, its general execution behavior and scheduling policy information. This data structure is called the task control block (TCB). Your system should only support a fixed number of tasks **(32 total)**. This means that all the TCBs needed can be statically allocated in the kernel and filled in via your system calls to `thread_create`. During the life-time of a task, it can be in a number of states. All tasks start out as *runnable*. The kernel will only launch tasks in the runnable pool. When a *runnable* task is launched, it changes to the *running* state. Once a *running* task finishes (or depletes its CPU budget), its state is changed from *running* to *waiting* while it waits for its next period. A task that is in the *waiting* state cannot be scheduled to run. For your kernel, we assume that no task exits and that all tasks execute periodicly forever. Please remember that the scheduler will not (and is not supposed to) launch any task that is not in the *runnable* state. Your implementation can either support a maximum of 32 user-threads and an idle thread, or 31 user-threads and an idle thread.

Please look ahead to the Task CPU Usage Visualization section at the end of this document as you will see that you eventually need to store information about task runtimes. This might help inform the design of your task control block structure.

The TCB should maintain all important information that the kernel needs to resume a thread's context. This should at least include **but is not limited to**:

1. The stack pointer for the task.
2. The program counter for the task.
3. The C (computation time) for the task.
4. The T (period) for the task.
5. The priority for the task.
6. The task's status (runnable, waiting, running).

Don't worry about using C and T yet, we will discuss those in the scheduler section. Inside the `thread_init` function, you should initialize your kernel data structures required by the scheduler. To emulate the states mentioned previously, we need to maintain a pool of threads in the runnable and waiting pool.

4.2 Runnable pool

We assume that the scheduler should create at most one task for every priority level on the system. Since we will be implementing a rate monotonic scheduler, there is one priority level for every task, and no more than one task with same priority is ever in the same runnable pool. We have a maximum of 32 tasks in our system. This is typically implemented as a bitmask using a 32 bit integer with a bit to represent whether a task of a given priority is present in the system. We simply need to find a non-zero bit to indicate that the task is in the pool. To find the highest priority task quickly, we find the first non-empty bit. **Please note that priority 0 is the highest priority and 31 is the lowest**.

4.3 Waiting Pool

Once a task begins waiting for its next period to run, you can remove the task from runnable pool and put the task into the waiting pool. Since the waiting pool and runnable pool have a 1 to 1 mapping, the implementation of this pool is the same as the runnable pool.

After we set up our thread pools in `thread_init`, the next system call is `thread_create`. This system call should simply just allocate a new thread and fill out the TCB data structure you create. Note that there is no heap for your kernel so you cannot call `malloc`. Instead, you should statically define all the TCBs you will need (32 total) so that the space for them is reserved when the kernel is compiled. The user space application statically defines their own stacks for their tasks and passes the kernel a pointer to the top of their stack in `thread_create`. See the example user mode applications we have supplied for you to see how this is done. Your kernel should only need initialize a TCB structure in the `thread_create` system call and then place the task in the runnable pool.

4.4 Context Swap

At this point, you should have a TCB structure defined and the system calls for `thread_create` and `thread_init` filled out. You should also have a runnable and waiting pool which are manipulated by the previously mentioned system calls. Now we will work on the meat of checkpoint 1: context switching. To context switch, we must go from one context to another. This will be done from the IRQ handler. The IRQ handler will execute at 1 millisecond intervals. In the previous lab, the IRQ handler was used for the servo system calls. These system calls are no longer needed. Now our IRQ handler will call our context switch routine after clearing a pending timer interrupt. To do this, `kernel_main` should also initialize the timer to run at the granularity specified previously before entering user mode. Additionally, your assembly routine for entering user mode should **not** enable IRQs. This is done via the `scheduler_start` system call. The rationale for this is because having IRQs going off while we are creating threads could allow for nasty race conditions with our scheduler. Since the IRQ handler can essentially go off at any time, we may be in the middle of a `thread_create` and our scheduler could try and run a thread who may not have been fully initialized! Instead of having you implement thread-safe system calls, you will disable IRQs (and thus disable your context switcher and scheduler) until the user program calls `scheduler_start`.

The context switch routine is usually written in assembly and handles switching from one context to another. This entire process is done in a few (high level) steps:

1. Save the current context such that we can eventually resume the task just interrupted.
2. Call the scheduler to pick the next task to run.
3. Resume the context of the task selected by the scheduler.

The first and last steps should be done in assembly and the scheduler can be written in C. You will need to modify your `irq_asm_handler` such that it saves context and then calls your `irq_c_handler` with this context. The C handler should then clear the pending interrupt and then call the scheduler routine in C. The scheduler, for this checkpoint, should just find the next runnable task in the runnable pool and select it as the task to run. Your scheduler will then return to the `irq_c_handler` which should return to the `irq_asm_handler` with a pointer the new context to resume. The actual switch of context should be done as you exit the `irq_asm_handler`.

1. Recieve an IRQ
2. Enter IRQ handler (in IRQ mode)
3. Save IRQ SPSR
4. Save user mode registers
5. Save SVC mode registers
6. Return to IRQ mode, save running task's context, call scheduler
7. Scheduler returns next task to run
8. Restore SVC mode registers
9. Restore User mode banked registers
10. Restore IRQ SPSR
11. Return from IRQ interrupt

Your IRQ handler needs to support context switching from artbitrary threads - threads could be in user mode or supervisor mode. As illustrated by the steps above, it is easier to follow the same steps, regardless of what mode the thread was in when the IRQ fired. Since the ARM architecture banks registers, we need to switch modes to save registers banked for specific modes. System mode has the same set of banked registers as user mode, but with a higher privledge level.

Once you've implemented a context swap, implement a dummy scheduler that finds the next runnable task in the runnable pool. Once the scheduler picks the next task, it should place the current running task in the runnable pool and make it *runnable* while also setting the task selected to run as *running.*

Finally you will need to implement the `scheduler_start` system call. This system call should simply just enable IRQs interrupts. Then once the first IRQ fires, the user `main` context will be saved on it s stack but never resumed since it was never registered with the kernel using a `thread_create` system call.

4.5 Basic Context Swap

Huzzah! you have just made a simple round robin scheduler! To test your implementation, run the released `Test_0` from the course staff. This test creates 2 threads and expects you to successfully run them both by switching between each of them at millisecond intervals.

5 Rate Monotonic Scheduling

Once we have context switching and a dummy scheduler for 2 tasks, we can generalize this implementation and implement our real time scheduler. Before the kernel decides to schedule a task set, it attempts to ascertain the schedulability of the given task set. There are numerous techniques that one can use to verify schedulability. In this lab, we are going to use the UB admissibility rule to ensure that the given task set is schedulable. The UB test is discussed in detail in the lecture slides. We will include the basic condition here for completeness.

A set of n independent, periodic tasks scheduled rate-monotonically, where task τ_i has periodicity T_i and worst case execution time of C_i, will always meet its deadlines, for all task phasings, if

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq U(n) = n(\sqrt[n]{2} - 1)$$

The task scheduler checks schedulability for the given task set in `scheduler_start`. This system call should take all the registered tasks with the kernel and make sure they are schedulable according to the previously mentioned test.

If it is not schedulable, you should return an error code. To test the scheduability for a set of n tasks, you will need to generate the upper bound for the n tasks according to the previously mentioned formula. For our kernel, $n = 32$. You will need to make a table of recalculated upper bounds from U($n = 1$) to U($n = 32$) and hard code this table into your kernel for use when checking schedulability. You are not required to do the exact response-time test.

5.1 Enforcement

Once you add this to your scheduler, we now need to enforce the C and T values specified. Previously, we just used round robin scheduling and switched between tasks at every IRQ. Now we will use Rate-Monotonic Scheduling (RMS) and enforce the policy within our kernel. At every IRQ, interrupt our RMS scheduler will need to check for the following things:

1. Has the current *running* task gone beyond its computation time? If so, this task should be put into the waiting pool and a new wake up time should be set for this task.
2. Has the current *running* task changed to be *waiting* via a call to `wait_until_next_period` (more on this later)? If so, move this task to the waiting pool and a new wake up time should be set for this task.
3. Is there a task in the runnable pool that has a strictly higher priority than the current running task? If so, make this higher priority task the new *running* task and take the old *running* task and add it to the runnable pool.
4. Are there any tasks in the waiting pool whose period has ended? If so, move them from the waiting pool to the runnable pool and set their next period time stamp.
5. If there is no runnable task and the current *running* task has been moved to the waiting pool, then run the idle task (more on this later).
6. If nothing else, then just return to the context of the task we interrupted and update the computation time spent on this task.

To make the this scheduler work, we need to keep track of system time. In your IRQ C handler, you need to add code to update a global 32-bit counter every 1 millisecond. Don't worry about overflow (However, this is an interesting problem and you should consider what would happen if it did). This system timer is the global clock. We then need to modify our TCB to accept time stamps for a given task. We need to keep track of the next period time stamp for a given task and the current computation time stamp. In our scheduler, we will use this global system timer updated by our IRQ handler to time stamp tasks and then use these time stamps to figure out the schedulability of a given task.

Now that we are actually enforcing computation times and periods, there may come a time when there are no runnable tasks. It is at this point when we need an idle task. You should create such a task and initialize it in `thread_init`. Then run this task whenever the runnable pool is empty.

Lastly, we need to implement 3 more system calls: `wait_until_next_period`, `get_time`, and `spin_wait()`. `get_time` should simply return the current system time in milliseconds. `wait_until_next_period` should change the current process state to *waiting* and then hang until another IRQ comes to deschedule the task. `spin_wait()` should make the calling thread spin (in a loop) for the given number of milliseconds, not including time where other threads were running.

6 Checkpoint

At this point you should be able to run Tests 0 to 4 correctly. Test 4-1 tests creating a large number of threads, this should work as well. To submit the checkpoint, use the tag `lab3-checkpoint`

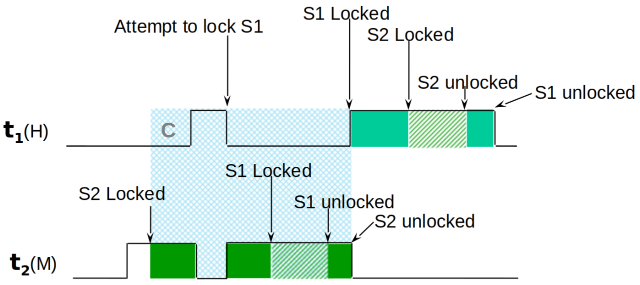


Figure 1: PCP Example From Lecture

7 Mutex and Priority Ceilings

In this section, we will add support for Mutexes and Priority Ceiling Protocol. Mutexes allow multiple program threads to share the same resources through acquiring and releasing of locks only allowing one thread to use the resource at a given time t. Priority Inversion is a problematic scenario in scheduling in which a high priority task is indirectly preempted by a medium priority task effectively inverting the relative priorities of these tasks. In this lab we will implement Priority Ceiling Protocol, which is a synchronization protocol for shared resources to avoid priority inversion and mutual deadlock by temporarily elevating the priorities of tasks in certain situations.

7.1 Mutexes

The three Mutex Functions that will implement are `mutex_init`, `mutex_lock`, and `mutex_unlock`. Look at the function definitions defined in *syscall_thread.h*. These functions all take a *mutex_t* pointer as a parameter. You get to define this structure yourself in `kernel/include/mutex.h`. User mode programs will be compiled against this header so the can user this header as well for the definition of the mutex structure.

Your implementation must support nested mutexes, and they will be tested with tests 7-9. We strongly suggest you think about designing your kernel with nested mutexes in mind from the beginning! Your kernel does not need to support more than 32 mutexes.

7.2 Supporting PCP (Priority Ceiling Protocol)

The priority ceiling of a shared resource is defined to be the priority of the highest-priority task that can ever access that resource. A given thread can only acquire a resource if its priority is strictly higher than the priority ceilings of all resources held by other threads. When a thread cannot acquire a resource because the resource is held by another thread or becuse another thread holds a resource with a priority ceiling greater than or equal to the priority of the thread attempting to acqurie the resource, that thread is blocked and gives its priority to the thread blocking it. When the blocking thread releases its resources, it will return to its original priority. An example from the lecture slides is shown in Figure 1. When t1 attempts to lock S1, t2's priority is raised to 1. Once t2 has released both of its mutexes, its priority will be lowered to the original value and t1 will start running.

Please refer to the lecture notes for more details on the Priority Ceiling Protocol. The threads are now changing priority as they block other threads returning to their old priority upon releasing. This will probably mean modifying your TCB to accommodate 2 priorities. One will represent the current priority and the other will represent the original priority. Upon Releasing control of the last mutex, we need to reset the threads priority to what it was prior to acquiring mutexes. This task can then continue executing even if there is a higher priority task in the runnable pool. Your RMS scheduler then treats the task like any other at the next IRQ interrupt. Like many obstacles, there are many well designed approaches, and many hacked-together solutions. We recommend you take some time to think about what exactly what you need to add to your code to implement PCP before you being coding.

8 Scheduling Tests

To help you exercise your kernel, we have provided several benchmark user programs. You can find the source code for each program under `code/lab3_test<n>`. Before running a test, you should look through the code and carefully think about what the expected behavior is. Note that the provided sample output in the handout may differ slightly from your output in terms of the timing of each task, because of differences in the JTAG debugging frequency on different machines. The order of tasks will, in most cases, remain the same (with some minor differences in the time values); however, you should still read through the code for each test and thoroughly convince yourself of the proper output of the test.

Test 0: Context Switching
Test 0 creates two identical threads which each print their name and spin-wait for a few milliseconds. This simple test is meant to verify that your kernel properly handles thread creation and context switching. You should be able to run this test without implementing any sort of scheduling or admission control.

Test 1: Admission Control
Test 1 calls `thread_create` with a set of 3 un-schedulable tasks. Your kernel should identify this via a basic UB test and `scheduler_start` should return an error.

Test 2: Rate-monotonic scheduling (no preemption)
Test 2 creates a set of 3 harmonic tasks intended to sanity test your kernel's rate monotonic scheduling. The execution times and periods for the tasks are designed such that no task will preempt another.

```
t = 1 --- Task: 1 Count: 0
t = 8 --- Task: 2 Count: 0
t = 25 --- Task: 3 Count: 0
t = 1000 --- Task: 1 Count: 1
t = 2000 --- Task: 1 Count: 2
t = 2007 --- Task: 2 Count: 1
t = 3000 --- Task: 1 Count: 3
t = 3007 --- Task: 3 Count: 1
t = 4000 --- Task: 1 Count: 4
t = 4007 --- Task: 2 Count: 2
t = 5000 --- Task: 1 Count: 5
t = 6000 --- Task: 1 Count: 6
t = 6007 --- Task: 2 Count: 3
t = 6024 --- Task: 3 Count: 2
t = 7000 --- Task: 1 Count: 7
t = 8000 --- Task: 1 Count: 8
t = 8007 --- Task: 2 Count: 4
t = 9000 --- Task: 1 Count: 9
t = 9007 --- Task: 3 Count: 3
t = 10000 --- Task: 1 Count: 10
t = 10007 --- Task: 2 Count: 5
t = 11000 --- Task: 1 Count: 11
t = 12000 --- Task: 1 Count: 12
t = 12007 --- Task: 2 Count: 6
t = 12024 --- Task: 3 Count: 4
t = 13000 --- Task: 1 Count: 13
t = 14000 --- Task: 1 Count: 14
t = 14007 --- Task: 2 Count: 7
t = 15000 --- Task: 1 Count: 15
t = 15007 --- Task: 3 Count: 5
```

Test 3: Rate-monotonic scheduling (with preemption)

Test 3 again tests your kernel's rate-monotonic scheduling. However this time, the execution times and periods are designed so that higher priority tasks will preempt lower priority ones.

```
t = 2 --- Task: 1 Count: 0
t = 19 --- Task: 2 Count: 0
t = 316 --- Task: 3 Count: 0
t = 410 --- Task: 1 Count: 1
t = 800 --- Task: 2 Count: 1
t = 820 --- Task: 1 Count: 2
t = 1200 --- Task: 3 Count: 1
t = 1230 --- Task: 1 Count: 3
t = 1600 --- Task: 2 Count: 2
t = 1640 --- Task: 1 Count: 4
t = 2050 --- Task: 1 Count: 5
t = 2400 --- Task: 2 Count: 3
t = 2460 --- Task: 1 Count: 6
t = 2714 --- Task: 3 Count: 2
t = 2870 --- Task: 1 Count: 7
t = 3200 --- Task: 2 Count: 4
t = 3280 --- Task: 1 Count: 8
t = 3600 --- Task: 3 Count: 3
t = 3690 --- Task: 1 Count: 9
t = 4000 --- Task: 2 Count: 5
t = 4100 --- Task: 1 Count: 10
t = 4510 --- Task: 1 Count: 11
t = 4800 --- Task: 2 Count: 6
t = 4920 --- Task: 1 Count: 12
t = 5114 --- Task: 3 Count: 4
t = 5330 --- Task: 1 Count: 13
t = 5600 --- Task: 2 Count: 7
t = 5740 --- Task: 1 Count: 14
t = 6000 --- Task: 3 Count: 5
t = 6150 --- Task: 1 Count: 15
```

Test 4: Enforcement

Test 4 verifies that you've implemented enforcement correctly. It creates a task which intentionally violates its reserved execution time. Your kernel should detect this and continue to schedule the other two tasks properly.

```
t = 1 --- Task: 1 Count: 0
t = 18 --- Task: 2 Count: 0
t = 319 --- Task: 3 Count: 0
t = 410 --- Task: 1 Count: 1
t = 820 --- Task: 1 Count: 2
t = 1018 --- Task: 2 Count: 1
t = 1200 --- Task: 3 Count: 1
t = 1230 --- Task: 1 Count: 3
t = 1640 --- Task: 1 Count: 4
t = 2050 --- Task: 1 Count: 5
t = 2460 --- Task: 1 Count: 6
t = 2518 --- Task: 2 Count: 2
t = 2718 --- Task: 3 Count: 2
t = 2870 --- Task: 1 Count: 7
t = 3280 --- Task: 1 Count: 8
t = 3600 --- Task: 3 Count: 3
t = 3690 --- Task: 1 Count: 9
t = 4001 --- Task: 2 Count: 3
t = 4100 --- Task: 1 Count: 10
t = 4510 --- Task: 1 Count: 11
t = 4920 --- Task: 1 Count: 12
t = 5019 --- Task: 2 Count: 4
t = 5118 --- Task: 3 Count: 4
t = 5330 --- Task: 1 Count: 13
t = 5740 --- Task: 1 Count: 14
t = 6000 --- Task: 3 Count: 5
t = 6150 --- Task: 1 Count: 15
t = 6502 --- Task: 2 Count: 5
t = 6560 --- Task: 1 Count: 16
t = 6970 --- Task: 1 Count: 17
```

Test 4-1: Many Threads

Test 4-1 simply spawns a large number of threads and switches between all of them on the same period. This tests that your kernel can handle storing the context of many threads.

Test 5: Mutexes

Test 5 verifies that you've implemented mutexes correctly. It creates 3 tasks which each lock and unlock a global mutex. However, one task spin-waits for a long period of time, thereby blocking the other two tasks. If mutexes are properly implemented, the other two tasks should not run while the third task is holding the lock.

Test 6: Priority Inversion

Test 6 tests your PCP implementation. The task set is designed to exhibit unbounded priority inversion, unless avoided through PCP.

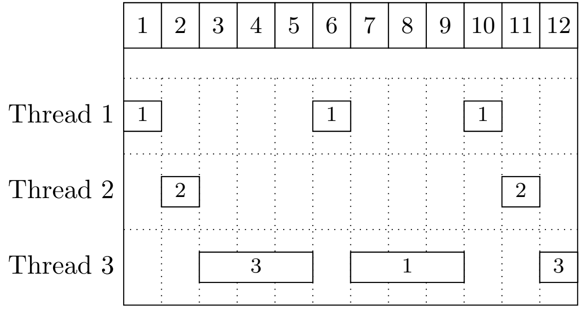
Tests 7-9: Nested Mutexes

Tests 7-9 test your implementation of nested mutexes. The expected results are listed in the header comment in the test's `main.c` file.

Tests 10-13: PCP

Tests 10-13 test your implementation of the priority ceiling protocol. Scheduling graphs for the tests, as well as descriptions of thread interleavings, are given below. You will need to determine whether your output matches these graphs or not.

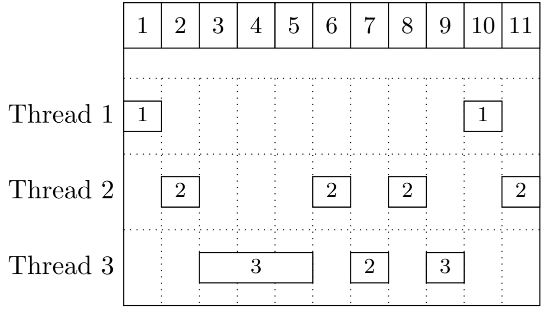
Test 10



```
Mutex 1 Ceiling = 1, Mutex 2 ceiling = 2;

T1@1  : mutex_lock(&m1)
T1@2  : mutex_unlock(&m1)
T2@2  : mutex_lock(&m1)
T2@3  : mutex_unlock(&m2)
T3@4  : mutex_lock(&m2)
T3@5  : mutex_lock(&m1)
T1@6  : Tries mutex_lock(&m1), fails (T3 holds lock), T3->prio=1
T3@8  : mutex_unlock(&m1) - T3 priority stays at 1, since it still holds m2
T3@9  : mutex_unlock(&m2) - T3 priorty returns to 3, no mutexes held
T1@10 : mutex_lock(&m1)
T1@11 : mutex_unlock(&m1)
T2@12 : mutex_lock(&m2)
T2@13 : mutex_unlock(&m2)
```

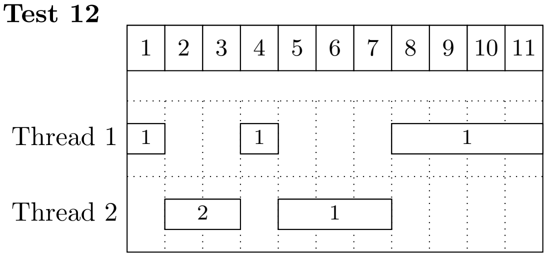
Test 11



```
Mutex 1 Ceiling = 0, Mutex 2 ceiling = 1;

T1@1  : mutex_lock(&m1)
T1@2  : mutex_unlock(&m1)
T2@2  : mutex_lock(&m1)
T2@3  : mutex_unlock(&m2)
T3@4  : mutex_lock(&m1)
T2@6  : mutex_lock(&m2), fails (T3 holds lock w/ higher ceiling), T3->prio=2
T3@7  : mutex_unlock(&m1) - T3 priority returns to 3
T2@8  : mutex_lock(&m2)
T2@9  : mutex_unlock(&m2)
T1@10 : mutex_lock(&m1)
T1@10 : mutex_unlock(&m1)
T2@11 : mutex_lock(&m2)
```

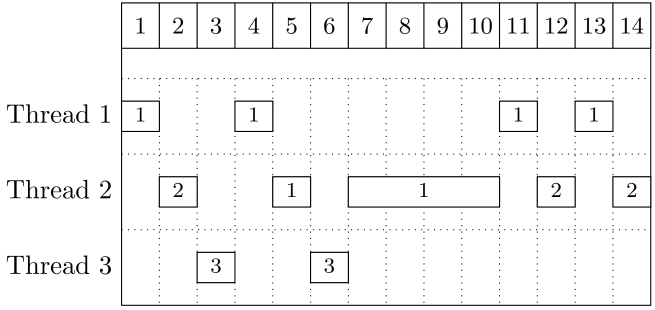
Test 12



```
Mutex 1 Ceiling = 1, Mutex 2 ceiling = 1;

T1@1  : wait_until_next_period()
T2@2  : mutex_lock(&m2)
T1@4  : mutex_lock(&m1), fails since T1->prio <= m2->ceiling, T2->prio=1
T2@5  : mutex_lock(&m1)
T2@6  : mutex_unlock(&m1)
T2@7  : mutex_unlock(&m2) - T2 priority returns to 2
T1@8  : mutex_lock(&m1)
T1@9  : mutex_lock(&m2)
T1@10 : mutex_unlock(&m2)
T1@11 : mutex_unlock(&m1)
```

Test 13



```
Mutex 1 Ceiling = 0, Mutex 2 ceiling = 1;

T1@1  : wait_until_next_period()
T2@2  : mutex_lock(&m1)
T3@3  : wait_until_next_period()
T1@4  : mutex_lock(&m1), fails since T2 holds lock, T2->prio=1
T2@5  : wait_until_next_period()
T3@6  : mutex_lock(&m2), fails (T3->prio <= m1->ceiling)
T2@8  : mutex_lock(&m2)
T2@9  : mutex_unlock(&m2)
T2@10 : mutex_unlock(&m1), T2 priority returns to 2
T1@11 : mutex_lock(&m1)
T3@12 : mutex_lock(&m2), fails since T3->prio <= m1->ceiling
T1@13 : mutex_unlock(&m1)
T2@14 : exit(2)
```

9 Submission and Demos

Make sure to incrementally push versions of your code to avoid losing work. Also, make sure to submit a link to your Github Repository on Canvas for both partners.

1. To submit the checkpoint, use the tag `lab3-checkpoint`. We are looking for a kernel that prints ”IRQ!” every second. Doxygen style will not be graded for the checkpoint.
2. To submit the final, use the tag `lab3-submit`. You must demo all the tests.

You will be penalized for not correctly tagging your submissions.