Lab 4: 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 Due: Thursday, April 15, 2021, 11:59pm
Final Due: Tuesday, April 27, 2021, 11:59pm
Demo By: Tuesday, May 4, 2021

Contents

1 Introduction and Overview

1.1 Goal

In this lab, you will develop a real-time kernel capable of admission control, task scheduling, isolation, and synchronization. In simpler terms, this means ensuring tasks are schedulable, actually scheduling tasks on the fly, memory protection (per thread), mutexes and avoiding deadlocks. The lab is divided into three main parts:

- Context switching and task management.
- Fixed priority rate-monotonic scheduling.
- Isolation and real-time synchronization.

1.2 Task List

1. Get the starter code from GitHub Classroom (Section 2.1)
2. Implement threading (Section 3)
   (a) Implement context saving.
   (b) Implement context switching.
   (c) Implement thread initialization and creation SVCs.
   (d) Design a thread control block.
   (e) Implement round-robin scheduling.
3. Implement Rate Monotonic Scheduling (Section 4)
   (a) Implement the UB test.
   (b) Implement the scheduler.
   (c) Implement scheduling-related SVCs.
4. Implement thread cancellation (Section 5)
5. Complete written questions on the Priority Ceiling Protocol (PCP) (Section 7).
6. Turn in Checkpoint 1 (Section 8).
7. Implement memory protection (Section 9).
8. Implement mutexes (Section 10).
9. Implement the Priority Ceiling Protocol (Section 11).
10. Submit to GitHub (see Section 13). Make sure your code has met style standards including Doxygen documentation.
11. Demo at office hours.

1.3 Grading

Your kernel should meet the following requirements:

Task	Points
Context swap with PendSV	40
Rate Monotonic Scheduling	50
Written Questions	10
Checkpoint Submission	10
Memory Protection	20
Mutexes	30
Priority Ceiling Protocol	30
Style (including Doxygen and submission protocols)	10
Bonus: User Test	5
TOTAL	200 pts

2 Before You Begin

2.1 Starter Code

Use the link on Canvas to create a GitHub repository for Lab 4. Create a new team if your partner hasn't made one or join your partner's team upon accepting the assignment.

Then fill in the following files using your Lab 3 implementation. Warning! Some of the files might be slightly different. Do this carefully.

```
kernel/src/i2c.c
kernel/src/kernel.c
kernel/src/led_driver.c
kernel/src/svc_handler.c
kernel/src/syscall.c
kernel/src/timer.c
kernel/src/uart.c

kernel/asm/asm_helpers.S
kernel/asm/boot.S

user_common/asm/svc_stubs.S
```

You'll mainly be working on the following files (and may wish to create your own files as well):

```
kernel/src/mpu.c
kernel/src/svc_handler.c
kernel/src/syscall_thread.c

kernel/asm/asm_helpers.S
kernel/asm/boot.S

user_common/asm/svc_stubs.S
```

2.2 Lab 3 System Call Interface

We will be using your Lab 3 system call implementation as a starting point for running multi-threaded programs. We have added threading-related system call prototypes in `user_common/include/349_threads.h`. The matching assembly function stubs are in `user_common/asm/svc_stubs.S`. As you did in Lab 3, map them to the appropriate SVC numbers from `kernel/include/svc_num.h`.

Since your scheduler will need to use SysTick it cannot co-exist with the Lab 3 implementation of `servo_set()` and `servo_enable()`:

- Gracefully degrade these functions by making the kernel implementation always return an error. You could re-implement a servo controller using your real-time kernel, but you do not have to do so in this lab.
- The `systick_c_handler` has moved to `kernel/src/syscall_thread.c` and is no longer in `timer.c`, since it is now performing a different job.

You may wish to fill in `user_common/asm/svc_stubs.S` and `svc_handler.c` for the new SVCs, which are described in the Appendix (Section 15.2). Some of the SVCs in this lab will have five arguments! You will want to check calling conventions to figure out how to access the fifth argument.

In this lab, SVCs have a lower priority than other interrupts, such as SysTick, PendSV (Section 3.1), and memory faults (Section 9) while UART interrupts have a higher priority in order to allow you to see output while executing these areas of code. In order to ensure that `printf` always succeeds in the tests, your `sys_write` should not return until it has placed all the characters in the UART buffer. If the buffer is full, it should poll the buffer until there is space. Your `sys_write` should now look something like this:

```
int sys_write(int file, char *buf, int len) {
    if (file != STDOUT) return -1;
    for (int i = 0; i < len; i++) {
        while (uart_put_byte(buf[i]));
    }
    return len;
}
```

Important note: UART interrupts are able to interrupt `uart_put_byte`. If you are using data structures that assume certain invariants for your kernel buffer, you will need to make sure an interrupt handler does not interrupt while you are modifying the UART kernel buffers. However, unlike in Lab 3, you do not want to blindly disable and enable interrupts, since this function may now be called while interrupts are already disabled, and you don't want this function to enable them. To do this, we record if interrupts are enabled or disabled before we disable interrupts, and then we simply restore the original state after we are done modifying the data structure. Two functions have been provided for you in `kernel/include/arm.h`:

- `int save_interrupt_state_and_disable()`, which returns the interrupt enabled/disabled state before it disables interrupts.
- `void restore_interrupt_state(int state)` which restores the given interrupt state.

3 Threads

In this section, you will be implementing basic multi-threading functionality in the kernel, allowing a user to run multiple threads.

The User Perspective

In Lab 3, user programs were "single threaded". Once the kernel called `enter_user_mode`, the user program ran, and the kernel occasionally performed SVCs and handled UART and timer interrupts, and eventually slept indefinitely if the user program ever returned.

For Lab 4, after the kernel calls `enter_user_mode`, the user mode `main` is able to register threads with the kernel. Once it has registered all the necessary threads it can call `scheduler_start` to begin running and scheduling the registered threads.

Three syscalls are provided to the user, to use in the following manner:

1. Call `thread_init` to set global threading parameters.
2. Call `thread_create` with thread-specific parameters for each thread the user wants to create.
3. Call `scheduler_start` to pass control to the kernel scheduler, which schedules and runs the registered threads.

You will have to implement these three system calls in Section 3.3, Section 3.4, and Section 3.7 respectively.

The Kernel Perspective

Once the kernel is initialized, it begins by branching to the user mode `main` function (hereafter referred to as the **"default" thread**). If the user program requires threading, the user program will invoke the syscalls mentioned above, and the kernel performs the necessary checks and bookkeeping (described later in this section) to run the threads that are being registered. Finally, when the user calls `scheduler_start`, the kernel begins scheduling and running the threads.

In order to enforce real-time scheduling, the kernel needs a way to keep track of time, and to interrupt the threads regularly to enforce the schedule. You will be using SysTick interrupts for this purpose. The SysTick interrupts only come into effect when scheduling starts.

A different interrupt, called PendSV, will be used to perform context switching in this lab. A context switch involves storing the state of the currently running thread so that execution can be resumed at a later time, effectively "pausing" the thread, then resuming a different thread from its own stored state.

The kernel has to keep track of the threading parameters the user provides, and keep track of the state of each thread. Thread states change as threads are scheduled and run. You will design a Thread Control Block (TCB) for this purpose in Section 3.6.

3.1 Context Switching and PendSV

Once the scheduler starts and the program begins receiving SysTick interrupts, the kernel has an opportunity, at each interrupt, to check scheduling and run a different thread. This would require switching from one thread's context to another. This is done using the PendSV interrupt as follows:

1. Set PendSV to pending. The processor automatically services the interrupt, calling the handler once the SysTick handler has returned.
2. Upon entry to the PendSV assembly handler, save register values onto the stack to make them accessible to the PendSV C handler.
3. Call the PendSV C handler.
4. In the C handler, run the scheduling algorithm.
5. If the same thread that was interrupted is still scheduled to run, return from the C handler.
6. If it is a different thread:
   (a) Save the current context to current thread's TCB.
   (b) Load the context of the next scheduled thread from its TCB.
   (c) Return from the C handler.
7. Back in the assembly handler, load the values from the stack back into registers. They may or may not have changed depending on whether the same thread was scheduled again.

Generally, we want to avoid assembly as much as possible, but we don't have much of a choice if we want to access registers directly to save or restore context. As such, this is all we do in the assembly handler; everything else is done in the C handler. You do not need to (and may not) use inline assembly.

We will be saving a thread's register values onto its own kernel stack (`msp`), then storing the stack pointer in the TCB as part of the thread's context. This way, you avoid having to copy the register values unnecessarily.

You will need to write at least two functions (but read all the way to Section 3.9 first!):

- `psv_asm_handler`. This is the PendSV assembly handler. It is located in `kernel/asm/boot.S`. You will need to update the vector table, as usual. PendSV is the 14th entry.
- `pendsv_c_handler`. This is the PendSV C handler which is called by the assembly handler, and is located in `kernel/src/syscall_thread.c`. You may change the type of this function if you wish. For now it has been set to receive a pointer as an argument and return a pointer as the result, as you will minimally need to pass the stack pointer as an argument and return a stack pointer as a result. You may change the function prototype if you wish; remember to change the header in `kernel/include/syscall_thread.h`.

3.2 Thread Context

In order to successfully perform a context swap, a thread should not realize it was interrupted when it is swapped out and swapped back in. You need to make sure the whole context is saved and restored and that nothing is missed.

A thread's context entirely determines its state. The context includes all register values, including banked registers, and also includes the exception state, whether it is in **user mode** (also known as thread mode or unprivileged mode) or in **kernel mode** (also known as handler mode or privileged mode), and using `msp` or `psp`.

Because of the way interrupt priority has been set up for you in this lab, PendSV will only interrupt a thread in user mode or while servicing an SVC. This is because SVCs have been set to a lower priority than UART, SysTick, and PendSV, which have been set to the same priority. Therefore, instead of having to discern between the various

interrupt handling states when the thread is in kernel mode, you only need to discern if a thread is in kernel mode servicing an SVC or in user mode.

With regard to the thread's context, note the following:

- `get_svc_status` in `src/kernel/arm.c` has been provided for you to check if the thread interrupted by the PendSV is servicing an SVC or not.
- Some of the thread's register values are already automatically saved when PendSV interrupt is serviced. You do not need to save these registers again, since they will be automatically restored.
- The value in `lr` upon entry to the handler (not the stacked `lr`!) will depend on the thread's mode (handler or thread) and the stack being used (`msp` or `psp`). As long as you ensure the return code for a thread is saved, then restored when the PendSV is launching that thread again, the thread will resume with the correct mode and stack.
- `set_svc_status` in `src/kernel/arm.c` has been provided for you, to set the SVC active status. When set, this simply tells the processor that there is a SVC that has not been completely handled and allows the processor to do a subsequent interrupt return from the SVC. If you do not set this correctly, you will receive **usage faults** (trying to handle an SVC with an SVC already in progress, or return from an SVC with no SVC in progress).

3.3 Threading Initialization

The first SVC that is called by the user for scheduling is:

```
int thread_init( uint32_t max_threads, uint32_t stack_size, ...  )
```

The `thread_init` function should initialize the kernel data structures required by the scheduler. These are global kernel data structures not specific to a particular thread, such as the tick counter, the waiting and runnable pools, the arguments given to the `thread_init` function, and so on. These will be more useful when you need to implement the scheduler, so you may wish to read on and continue filling it in as you go.

For now, the only relevant arguments to `thread_init` are the first two. The rest of the arguments to `thread_init` will be useful later.[1]

- `max_threads` is the maximum number of threads the user will create, supplied by the user at run-time.
- `stack_size` is the size of a thread's user and kernel stacks. The size of the stack that is supplied in the function parameter **is in units of "words", not bytes!** A word is four bytes. A parameter value of 256 means a stack size of 1KB. The user can ask for any stack size, but you may wish to round it up to the closest power of two (with a minimum of 1KB). This is not absolutely necessary now, but will be when you are implementing memory protection. There is a function `mm_log2ceil_size` in `mpu.c` that takes an integer n and returns $\lceil \log_2 n \rceil$.

To make things easier for you, we are fixing the location of the thread stacks. You should make sure the stack size supplied by the user will fit in these regions, given the expected number of threads:

- All threads' user-space stacks should be placed between `__thread_u_stacks_low` and `__thread_u_stacks_top` (32KB of space).
- All threads' kernel-space stacks should be placed between `__thread_k_stacks_low` and `__thread_k_stacks_top` (32KB of space).

Note: the default thread's user and kernel stacks are already allocated by the time `thread_init` is called, so you do not need to allocate them here.[2]

[1] You may refer to Section 4.5 for `idle_fn`, Section 9 for `memory_protection`, and Section 10 for `max_mutexes`.

[2] In fact, the call to `thread_init` is using the default thread's kernel stack. If you check the linker template, the default thread stacks descend from `__psp_stack_top` and `__msp_stack_top`.

3.4 Thread Creation

The second SVC called by the user for scheduling is:

```
int thread_create( void *fn, uint32_t prio, uint32_t C, uint32_t T, void *vargp )
```

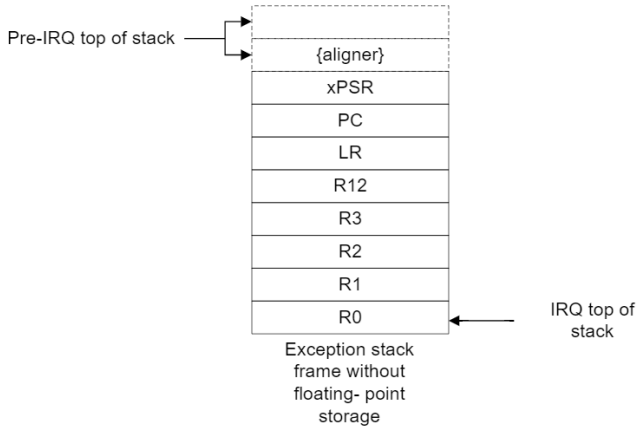
In order to tell the scheduler what threads it needs to schedule, the user calls `thread_create`. This function should fill out the statically defined TCB data structure for one of the threads with the arguments the user has provided, set up a user stack and kernel stack, and make the thread runnable.

You must carefully consider how to fill out the TCB for a new thread. Values in registers must be arranged so that when a thread is swapped in for the first time in the PendSV handler and the PendSV handler returns, it is as if the thread function `fn` has just been called with `vargp` as its only argument. Consider the following:

- Not all the registers matter on the first time a thread runs. Which ones matter? Where should you put their values? Be extra careful with the registers that are automatically restored.
- The STM32 automatically restores `r0-r3`, `r12`, `lr`, `pc`, and `xPSR` from the relevant stack when an interrupt returns (see Figure 1). This will be `psp` if returning to user mode and `msp` if returning to kernel mode. For a brand new thread, it should return to user mode. It is your job to set the `msp`, `psp` as well as the correct return code.
- **Be careful with setting stack pointers!** Remember that an ARM stack grows downwards, so a stack pointer for an empty stack is at the opposite end of an array from the way we normally refer to an array with a pointer.
- There is a `thread_kill` system call which should be run if a thread returns (and in some other situations which are only relevant beginning at Section 5).

EXC_RETURN	Description
0xFFFFFFF1	Return to Handler mode. Exception return gets state from the main stack. Execution uses MSP after return.
0xFFFFFFF9	Return to Thread mode. Exception Return get state from the main stack. Execution uses MSP after return.
0xFFFFFFFD	Return to Thread mode. Exception return gets state from the process stack. Execution uses PSP after return.
All other values	Reserved.

(a) Return Codes



(b) Interrupt Stack Frame

Figure 1: Exceptions

3.5 Thread Priority

For this lab, you may assume that the user creates at most one thread for every static priority level on the system. Two threads will not be assigned the same static priority. The user program determines the priority level of the thread and passes this during `thread_create`. If you wish, you can return an error if the user tries to assign the same static priority to multiple threads, but this will not happen in the user tests.

Because of this constraint, threads can be distinguished from each other by their static priority level. You can utilize this fact when constructing your TCB, and also the runnable and waiting pools.

In this lab, we will use the convention that a smaller number is a higher priority. As such, **priority 0 is the highest priority!** A thread with priority 0 will run before a thread with priority 1.

There is a SVC called `get_priority`, which should return the priority of the currently running thread. For now, this is the same as the static priority of the thread.

3.6 Thread Control Blocks

The kernel maintains a data structure for every thread it is instructed to run. This data structure describes the thread's current state, its general execution behavior and scheduling information, and it is called a thread control block (TCB).

A TCB should maintain all the information that the kernel needs about a thread in order to schedule it at the right frequency, pause it and continue running it at a later time. You will be designing your own TCB. It should include (but is definitely not limited to) the following:

- Thread context (see Section 3.1)
- C and T for the thread (see Section 4.2)t
- Priority
- The thread's status (runnable, waiting, etc.)

You will have to continue modifying your TCB as you implement the rest of the kernel.

Your system only needs to support a maximum of 16 threads, including the idle thread and the default thread. This means that all the necessary TCBs can be statically allocated in the kernel, and later filled in via your system calls to `thread_create`.

3.7 Start Scheduling

```
int scheduler_start( uint32_t freq )
```

The threads that have been registered only start running when the SVC `scheduler_start` is called. The frequency (in Hz) of SysTick will be supplied by the user when `scheduler_start` is called, and you should start the timer with that frequency. Every time SysTick fires, it should check your scheduler to determine which thread runs next, then perform the context swap (if necessary).

In `scheduler_start`, once you have done your bookkeeping, you can pend a PendSV to kickstart the scheduler. Because of the way the interrupt priorities have been configured, the PendSV will be serviced immediately, halting the default thread in the middle of `scheduler_start`.

You might be wondering what to do with the default thread's context: save it in its TCB in case the scheduler ever runs out of threads. Once the scheduler starts, the default thread should not run again until all other threads (other than the idle thread) have returned or been killed (see Section 5). Once there are no more threads to run, the scheduler should terminate and resume the default thread so that the system can either initialize new threads or exit cleanly. (Note: this functionality will not be tested until Section 5, but you might as well work it out now.)

3.8 Round Robin Scheduler

Once you've implemented a context swap, we can implement a simple scheduler to test it. For now, make a round-robin scheduler, which just runs all the threads in order of static priority. (In PendSV, find the next runnable thread and select it as the thread to run, and loop back to the first thread when there are no more.)

Note: don't worry if your round-robin scheduler doesn't work particularly well, or it doesn't account for edge cases, as long as context switching is working. Your round-robin scheduler will not be tested; it is mainly for you to test context switching.

3.9 Debugging and Testing

Congratulations! You have just made a simple round-robin scheduler, a bare-bones kernel that will support multiple threads but does not yet provide real-time guarantees. To test your implementation, you can use the provided `test_0_0`, `test_0_1`, and `test_0_2` by running:

```
make flash USER_PROJ=test_0_X
```

`0_0` creates one thread, which should test your context saving and restoring. `0_1` creates two threads which you should be able to switch between. `0_2` creates many threads.

Here are some useful debugging tips:

- Running only one thread, use GDB to put breakpoints within your PendSV handler and single step through your code to make sure a single thread's context is properly saved and restored.
- Examine your TCB to make sure the threads' context are properly saved and restored respectively when switching threads.
- You can use the `breakpoint()` function in `kernel/include/arm.h` to set breakpoints in your code.
- You can use functions in `kernel/include/debug.h` to set breakpoints in your code, and it allows your to turn the macros on and off with the DEBUG compiler switch. Adding DEBUG=0 turns them off. We will not penalize your style for macros which can be removed at compile time in this way.

4 Rate Monotonic Scheduling

Now that we have successful context switching and a round-robin scheduler, we can generalize this implementation and implement our real-time scheduler.

4.1 Schedulability Check

Before the kernel decides to schedule a set of threads, it attempts to ascertain the schedulability of the 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:

> A set of n independent, periodic tasks scheduled rate-monotonically, where task t_i has periodicity T_i and worst case execution time C_i, will always meet its deadlines (for any task phasings) if
>
> $$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq U(n) = n(2^{1/n} - 1)$$

The UB test is discussed in detail in the lecture slides. You are not required to do the exact response-time test.

The task scheduler checks schedulability for the given task set every time a new thread is created (in `thread_create`). Each time a thread is created, the kernel should make sure the resulting set is schedulable according to the previously mentioned test. If it is not schedulable, you should return an error. To test scheduability for a set of n tasks, you will need to generate the UB test value for the n tasks according to the previously mentioned formula. For your kernel, $n \leq 14$. Unless you want to write complicated non-integral exponentiation functions, the UB test values have to be hardcoded. This has been done for you in `syscall_thread.c`.

Your implementation **does not** need to ensure that threads with the shortest period have the highest priority, since we defer priority assignment to the user. You can assume that the user has already taken care of static priority assignment when passing priority values in `thread_create()`.

4.2 Thread Scheduling

Threads in a real-time system are generally periodic. Every so often, they will run for a little while, complete their task, and then wait for the next cycle. The kernel's job is to make sure they get to run as often as they need to. If not, **very bad things** could happen!

When threads are registered, two of the required parameters for a thread are its period (called T) and its allotted computation time in a single period (called C). In this lab, we assume a thread's deadline D is equal to its period T.

During the lifetime of a thread, it can be in a number of states.

- A **running** thread is currently running. Clearly, there should only be one running thread.
- A **runnable** thread is ready to be run. There could be many runnable threads. This is when scheduling policy is important to determine which of the runnable threads to run.
- A **waiting** thread is not ready to be run. It could be because the thread exhausted all its allotted C, or it has simply finished its task and is waiting for the next period.

Threads generally start out **runnable**. Once a thread is launched, it is in the **running** state. Once a **running** thread finishes (or depletes its C budget), its state is changed to **waiting** while it waits for its next period. The kernel will not schedule a thread in the **waiting** state, but it can change the state of the thread from **waiting** to **runnable**, if necessary.

You may wish to create a "runnable pool" and a "waiting pool", which indicate which threads are in which states. These will be helpful for debugging as you can simply check the pools to see that the correct thread is running.

The kernel should only launch runnable threads!

4.3 Enforcement

Previously, the kernel just used round robin scheduling and switched between threads at every SysTick interrupt. Now, with Rate-Monotonic Scheduling (RMS), the kernel needs to enforce priority and the C and T values specified. In the PendSV C handler, the RMS scheduler should switch to the highest priority runnable thread (which could be the currently running thread, in which case there is no switch). Note that if the running thread is pre-empted by another thread, the running thread should become runnable.

At every SysTick interrupt, the RMS scheduler will need to do the following:

- Update the computation time for the currently running thread.
- Has the currently running thread reach its computation time limit for the current period? If so, change its state to **waiting**.
- Has the wake up time of any **waiting** threads been reached? If so, make them **runnable**. This might be a good time to update their next wake up time!
- Run the scheduler by pending a PendSV.

4.4 System Time

```
get_time( void )
```

The scheduler needs to keep track of a **system time**. This is the number of SysTicks since the scheduler was started. In the SysTick handler, add code to update a global 32-bit counter every interrupt. Don't worry about overflow[3]. This system time becomes the global clock.

Then, the TCB needs to accept time-stamps to keep track of when the next period of each thread starts. It also needs to track the current computation time elapsed (in the current period) for each thread. In our scheduler, we will use this global system timer updated by the SysTick handler to obtain time-stamps and then use these time-stamps to figure out which threads should be scheduled.

You might be wondering about edge cases relating to threads running for a little bit of a tick and then calling `wait_until_next_period`, which runs a different thread for the remaining part of a tick. Threads could also run for less than a tick before calling `wait_until_next_period`, especially if we reduce the scheduler frequency. To address these edge cases, you should charge the computation time only to the thread that is interrupted by the SysTick.

You should implement the SVC `get_time` so the user can obtain the above-mentioned system time.

4.5 Idle Thread

```
thread_init( ..., void *idle_fn, ...  )
```

Now that we are actually enforcing computation times and periods, there may come a time when there are no runnable threads. This is where an idle thread comes in. The user will create such a thread and initialize it in `thread_init`. Then, the kernel will run this thread whenever the runnable pool is empty.

Occasionally the idle thread is used in an RTOS to run auxiliary functions that are not time-critical. Therefore, the kernel should also save the context of the idle thread and restore it as necessary so these functions can run properly. You must allow the system calls that can apply to the idle thread to work properly with it. `wait_until_next_period` should do nothing for the idle thread.

Sometimes, the user might not have an idle thread. The user will indicate this by passing `NULL` as the idle function. In this case, we will assign a default idle function for the kernel to run. This default function should simply sleep and

[3]However, this is an interesting problem and you should consider what would happen if it did.

wait for the next interrupt. and you can use the `wfi` assembly instruction or the `wait_for_interrupt` C function in `user_common/include/349_lib.h` which also calls that instruction.

It is probably simplest to write the default idle function in assembly. You can choose to do this either in kernel mode or user mode. You can place the function in `kernel/asm/asm_helpers.S` or in `user_common/asm/svc_stubs.S` respectively.

4.6 Sleeping Early

`wait_until_next_period( void )`

Because the user wants to ensure threads always meet their deadline, they provide more than enough computation time and threads will often finish their work early . When a thread is done with the work it needs to do in its current period it can call `wait_until_next_period` to tell the scheduler, and the scheduler can forfeit its remaining computation time C for the current period and run a different thread.

Not all threads will call `wait_until_next_period`. Some threads will (legally) rely on the scheduler to put them to sleep when their computation time has run out. Some threads will (legally) call this for some periods and run out of computation time for other periods.

When a thread calls `wait_until_next_period`, you should set the thread to *waiting* then immediately pend a PendSV to run the scheduler. This will swap the thread out immediately, without waiting for the next SysTick. If you are concerned about computation time edge cases, see Section 4.4 and Section 4.7.

4.7 Simulation of Work

`thread_time( void )`

Because we are running dummy test programs, they don't always have useful work to do. However, we want them to pretend that they have some work that takes some time to complete. In real programs, this might be reading a sensor, checking the network, reporting status, or something else.

In order to pretend they are working, the user programs call the function `spin_wait` and `spin_until`, which has already been defined and implemented for you in `user_common/src/349_lib.c`.

You need to implement the SVC `thread_time`, which is called by the `spin_X` functions. It should simply return the number of ticks the current thread has used for computation since the scheduler started. **This is the total over all periods, not just the current period. This is not the same as C or the remaining quota in the current period!**. Once again, to address edge cases, you only need to increase this value for a thread when it is interrupted by SysTick.

4.8 Testing

At this point you should be able to run Tests `0_X` to `3_X` correctly. For more details on what the tests do, see Section 15.1 in the Appendix. Here are some tips to help you debug:

- Look over the tests before you run them to know what the expected output should be. Most of the tests should have their expected output in a comment at the top of the file.
- Look through the code and manually draw out the thread timelines, then at each context swap check that the yielding thread and scheduled thread match your timeline at the given tick.
- If it doesn't match, check that each thread is in the correct state at every tick.

5 Thread Cancellation

Sometimes, in a RTOS, if a thread finishes its task and is no longer needed, it can be de-scheduled (removed from both the runnable and waiting pools). In doing so, we can free up resources which can then be used by another thread.

5.1 Terminating Threads

Occasionally, threads might do something they are not supposed to, which includes bad memory accesses, attempting to acquire mutexes they should not, and so on. When something like this happens, we want to be able to terminate the thread without crashing the entire system.

You should implement the system call `thread_kill`, which permanently deschedules the thread it is called from. The scheduler should continue with the other threads. If there are no more threads, the scheduler should stop SysTick and switch back to the default thread to continue execution (`scheduler_start` finally returns!). Note that the idle thread and the main thread are not allowed to use this SVC. If the idle thread calls it, you should run the default idle thread instead, and if the main thread calls it, you should abort the program and call `sys_exit`.

5.2 Late Thread Creation/Restart

Clearly, a system running with fewer threads than it is supposed to is not ideal. We will therefore let the user call `thread_create` after `scheduler_start`, to create an entirely new thread[4] that takes the place of any dead or uninitialized threads.

If you need to, modify `thread_create` so it will still work after the scheduler has started. The standard conditions must still hold for `thread_create` to succeed: the total number of threads must still not exceed the maximum threads given during `thread_init`, the resulting threads must still be schedulable under the UB test, the thread must still have a unique static priority level, and so on. If these conditions are violated, return an error from `thread_create` just as you would have before the call to `scheduler_start`.

5.3 Testing

After you implement thread revival, you should be able to run tests `4_X` correctly, as well as `grade_RMS`.

[4] You might be wondering why we don't just revive the old thread. This is because it is possible that a system is, by design, switching between two task sets at a larger time-scale, and simply wants to change the task set. In this case the threads would have returned normally from their functions rather than being terminated for bad behavior.

6 Priority Ceiling Protocol

Later in this lab, you will be implementing mutexes and the Immediate Priority Ceiling Protocol (IPCP). Because this scheduling algorithm takes a while to get familiar with, we want you to complete two written scheduling questions before the checkpoint to get you started thinking about it. You will only need to implement mutexes and IPCP after the checkpoint for the final submission.

6.1 About IPCP

The Immediate Priority Ceiling Protocol is a synchronization protocol for shared resources to avoid unbounded priority inversion and mutual deadlock by temporarily elevating the priorities of tasks in certain situations.

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.**

There are two situations in which a thread τ_a, with priority P_a, cannot acquire a resource:

- because the resource is held by another thread τ_b; or
- because another thread τ_b holds a resource with a priority ceiling greater than or equal to P_a.[5]

A thread that locks a resource immediately takes on the new inherited priority.

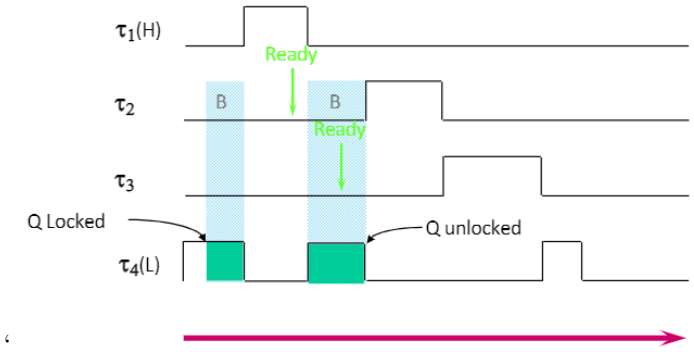


Figure 2: IPCP Example

An example from the lecture slides is shown in Figure 2. When t_4 attempts to lock Q, t_4's priority is raised to 2. t_1 can interrupt its execution, but when t_2 and t_3 are scheduled to run they are blocked by t_4. Once t_4 has released Q, its priority will be lowered to the original value and t_2 will start running. Note in this example that Q has a priority ceiling of 2 because t_2 will lock it at some point. You may refer to the lecture notes for more information on PCP.

[5]This second case actually covers the first case, so there is only really one case.

7 Written Problems

Complete the following two written problems using IPCP. You may complete this question digitally or on paper. Upload a picture (.png or .jpg) or a scan (.pdf) of your solution in the base directory of your repository. Please name this file `written.<png/jpg/pdf>`.

7.1 Question 1

Assume we want to schedule the following task set {C,T} with RMS where all the tasks are independent. Assume all tasks finish within their allotted C.

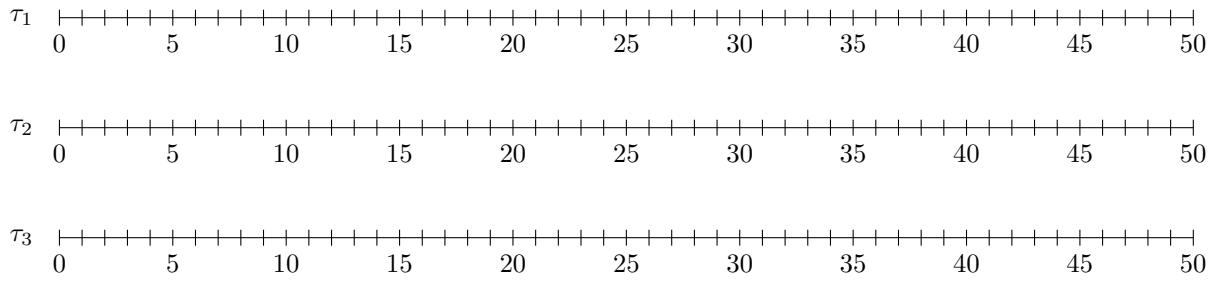
$\tau_1 = (4, 11)$
$\tau_2 = (5, 16)$
$\tau_3 = (3, 31)$

Prove that the above task set is schedulable.

Unfortunately, we can't always run our tasks in isolation. We need to share resources between each of them. We protect this data from corruption with the following scheme:

- τ_1 uses mutex m_1 for the **last two ticks** of its computation time.
- τ_3 uses mutex m_1 for all three ticks of its computation time.

Please draw the expected scheduling using IPCP of this task set with the mutexes on the timelines below, until $t = 40$. Assume that locks and unlocks can be done instantaneously (0 ticks). Indicate in which ticks each task is holding a mutex by shading that tick in. Does any task miss its deadline?



7.2 Question 2

Once again, assume we want to schedule the following task set {C,T} with RMS where all the tasks are independent. Assume all tasks finish within their allotted C.

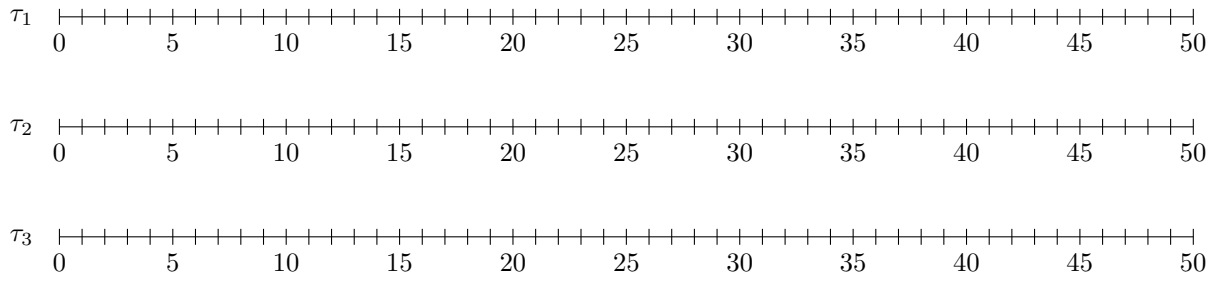
$\tau_1 = (3, 7)$
$\tau_2 = (1, 9)$
$\tau_3 = (6, 26)$

Prove that the above task set is schedulable.

Once again, consider now that the threads share resources as follows:

- τ_1 uses mutex m_1 for the **last two ticks** of its computation time.
- τ_2 uses mutex m_2 for all of its computation time (lock at the beginning of the tick, unlock at the end of the tick).
- τ_3 uses mutex m_1 for the **first four ticks** of its computation time, and mutex m_2 for the **last four ticks** of its computation time. (There will be two ticks when both mutexes are locked.)

Please draw the expected scheduling using IPCP of this task set with the mutexes on the timelines below until $t = 40$. Assume that locks and unlocks can be done instantaneously (0 ticks). Indicate in which ticks each task is holding a mutex by shading that tick in. Does any task miss its deadline?



8 Checkpoint

Congratulations, you have completed the checkpoint. Instructions on submitting to GitHub are in Section 13. Remember to submit the written question!

9 Memory Protection

The STM32 has a memory protection unit (MPU, on page 193 of the M4 programming manual). In this section you will be using it to protect kernel memory when in user mode, and to isolate a thread's stack from other threads.

The MPU provides functionality that allows you to declare a base address and region size and modify the permissions on this region. This MPU gives you 8 such regions. The main settings to note are the CTRL register, RNR register, RBAR register, and the fields AP, XN, and ENABLE in the RASR register. Once protection is on, if a user thread attempts to access a region it does not have permissions to access, a memory protection fault will be generated. **You must enable memory management faults in the system control block, or you will get hard faults!** See page 235 of the M4 programming manual.

When the system is in user mode:

- User-mode code and read-only data should be read-only.
- The user heap, data section, bss, and the default thread's stack should be fully accessible (read/write).
- If the memory protection mode when `thread_init` was called was `KERNEL_ONLY`, threads should have full access to all user stacks. Otherwise (if the mode was `PER_THREAD`) each thread should only have access to its own stack (and the default thread's).
- The rest of the memory should be protected using the background region.

When the system is in kernel mode, memory should be fully accessible (since the kernel manages the MPU anyway).

You should modify your `kernel_main` to protect the kernel at all times when the program is in user mode, even when threads are not running. Giving the user access to the non-thread regions listed in Section 9.1 will require 6 of the 8 available region numbers. You should use the 2 remaining region numbers for your thread stacks, and change the region as you switch between the threads.

We have already provided two helper functions for you that handle interfacing with the MPU for the memory region definitions. You should still read the manual to figure out what these functions do in detail. The functions we provide and their arguments are described in detail in `kernel/src/mpu.c`:

- `mm_region_enable`: Defines a memory region on the MPU. This function enables user access to the memory region if background protection is on.
- `mm_region_disable`: Removes a memory region from the MPU. This disables user access if background protection is on.

9.1 Memory Region Sizes

You may assume the following sizes for these memory regions:

- User code: 16KB
- User read-only data: 2KB
- User data: 1KB
- User BSS: 1KB
- User heap: 4KB
- Default thread user stack: 2KB

Refer to the linker script for the appropriate linker symbols. Because memory protection regions must be powers of two and base addresses must be aligned to powers of two, you may have to add padding if the stack size provided by `thread_init` is not a power of two. Some auxiliary functions have been provided in `mpu.c` to help to do this. Remember to update `thread_init` as necessary!

9.2 Memory Faults

We provide an incomplete memory fault handler for you in `kernel/src/mpu.c`, which should help with debugging. You will have to provide an assembly handler in `kernel/asm/boot.S`. The handler is the 4th entry in the interrupt vector table and is labelled as MM Fault. The assembly handler only needs to move `psp` into `r0` and then call `mm_c_handler`.

You should fill in the condition for stack overflow in `mm_c_handler`, as this is checked in some of the tests. We have no choice but to abort the system if stack overflow happens, because any adjacent memory to the stack would be clobbered when the processor automatically saves the context to the now-overflowing stack. You should call `sys_exit` in this case.

For all other errors, we can simply kill the offending thread. You should decide what the best way to do this is, and fill out the rest of `mm_c_handler`.

9.3 Debugging and Testing

You should now be able to run tests `5_X`.

Here are some tips to help you debug:

- If you changed your `thread_kill` code, make sure to run the earlier tests again to verify your updates.
- Ensure your stack locations are correct and valid, especially for the idle and default thread. Remember you don't need to separately allocate a stack for the default thread!
- If the region enabling and disabling functions provided to you return -1, that means there is an error with the way you are using these functions. Don't simply ignore the return value!
- If you're getting faults immediately upon enabling the MPU, make sure you enable kernel mode access to the background region at the same time. Otherwise, the kernel will be unable to run!

10 Mutexes

In order to enable sharing of resources, your scheduler will use mutexes. Mutexes allow multiple program threads to share the same resources without stepping all over each other and causing race conditions. This is done through the acquiring and releasing of mutexes. Only one thread is allowed to acquire a mutex at a given time, and it may only use the shared resource associated with a mutex when it has acquired the mutex.

The three mutex functions that you will implement are `mutex_init`, `mutex_lock`, and `mutex_unlock`. You may take a look at the function definitions defined in `kernel/src/syscall_mutex.h`. These functions all use the `kmutex_t` type, which is a struct. You should decide what fields to include in this struct.

The mutex struct is opaque to the user, who will see the typedef in `user_common/include/349_thread.h` where `mutex_t*` is simply a void pointer. While this pointer would potentially point at the mutex's memory location in the kernel, the user will not be able to access the values in the `kmutex_t` struct if you have implemented memory protection correctly.

10.1 Mutex Initialization

```
mutex_t *mutex_init( uint32_t max_prio )
```

The user program requests a mutex by calling `mutex_init` and providing the priority ceiling (`max_prio`) of the mutex. The kernel returns a handle to the mutex. The kernel does not need to verify the correctness of the priority ceiling in `mutex_init` (to do so would be very difficult). However, you should check its validity when mutexes are locked.

10.2 Acquiring and Releasing Mutexes

When a thread attempts to acquire a mutex, it will not proceed until it has acquired the mutex. This may not happen immediately. For example, another thread could have acquired the mutex earlier and have yet to release it. In your implementation, `mutex_lock` should not return (from the thread's perspective) until the mutex has been acquired, since the user thread will assume that the mutex has been acquired when the function returns.[6] This means there is a possibility of pending a PendSV in `mutex_lock` and `mutex_unlock`, where you switch to the thread holding the mutex (only for now; subsequently you might be blocked in other ways when you implement PCP). Remember to save the old context!

In `mutex_lock`, a check should be performed by the kernel to ensure a thread is not locking a mutex with insufficient priority ceiling. This is to make up for the fact that the kernel does not check if the priority ceiling is valid during mutex initialization. You should terminate the thread if it tries to do this, and print a warning.

A thread unlocks a mutex when it has finished using the associated shared resource. In your implementation, you do not have to worry about badly-behaved threads that capture the mutex forever. Your scheduler may assume mutexes will not be locked when a thread calls `wait_until_next_period`. In fact, you kernel should print a warning if mutexes are held when a thread finishes its computation time (either misses its deadline or calls `wait_until_next_period`). To make things simpler, the idle thread is not allowed to lock mutexes.

Your implementation must support nested mutexes. For example, a thread should be able to lock mutex A, lock mutex B, release mutex A, then release mutex B. 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.** This should allow you to use bitvectors to keep track of them.

You can test mutexes with tests 6_X and 7_X. See Section 15.1 in the Appendix for more information about the tests.

[6] Since we're using the Priority Ceiling Protocol, a thread should never find the mutex "locked". Instead, the thread that has the mutex is immediately elevated to a priority where that thread asking for the mutex would not run.

11 PCP Implementation

When implementing IPCP, the threads are now changing priorities as they obtain and release resources. This may (depending on your design choices) require modifying your TCB to accommodate two priorities. One will represent the thread's inherited (dynamic) priority and the other will represent the thread's original (static) priority. You will need to update `get_priority` to return the thread's inherited priority when it is different from its static priority.

For mutexes and priority inversion in general, it is possible that a single thread locking a single resource may be blocking multiple threads, or that multiple locked resources separately block a single thread, or that multiple locked resources block multiple threads. However, this might not be the case for the version of IPCP we have defined for you. It is worth thinking through and deciding how best to implement IPCP, perhaps enumerating all possible situations where IPCP is invoked. The guarantee that threads will not hold mutexes across multiple periods should simplify your implementation significantly.

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 IPCP before you being coding. Re-read the IPCP notes above (Section 6.1) and re-visit the lecture slides on PCP.

It is possible to avoid full inherited priority re-computations when mutexes are locked and unlocked, but we will not require you to do so.

You may test your IPCP implementation with tests 8_X, 9_X, and grade_PCP. See Section 15.1 in the Appendix for more information about the tests.

12 Bonus

12.1 User Test (5 points)

Write a threaded user program! It should pass if a certain aspect of context swapping, RMS, memory protection or PCP is implemented correctly, and fail if it is not correct. You must describe precisely what aspect of the kernel implementation the test checks for in the comments at the top of the file and the expected output.

Place your test in a folder `user_proj/test_bonus/src`. It should run using `make flash USER_PROJ=test_bonus`.

13 Submission and Demos

13.1 Github

If you have any comments or suggestions, please feel free to let us know in `README.txt`.

1. To submit your checkpoint, create an issue on GitHub titled `Lab4-Checkpoint`. In the comments section include your and your partner's name, Andrew ID, and the commit-hash you want to submit.
2. To submit the completed lab, create an issue on GitHub titled `Lab4-Submission`. In the comments section include your and your partner's name, Andrew ID, and the commit-hash you want to submit.
3. If you completed the bonus, please submit an additional issue, titled `Lab4-SubmissionBonus`, with the bonus commit hash. Finally, be sure to submit the link to your repository to Canvas.

When you are done, it should look like Figure 3. Pictures of cute animals are not needed but are recommended.

13.2 Demo

You do not need to demo the checkpoint.

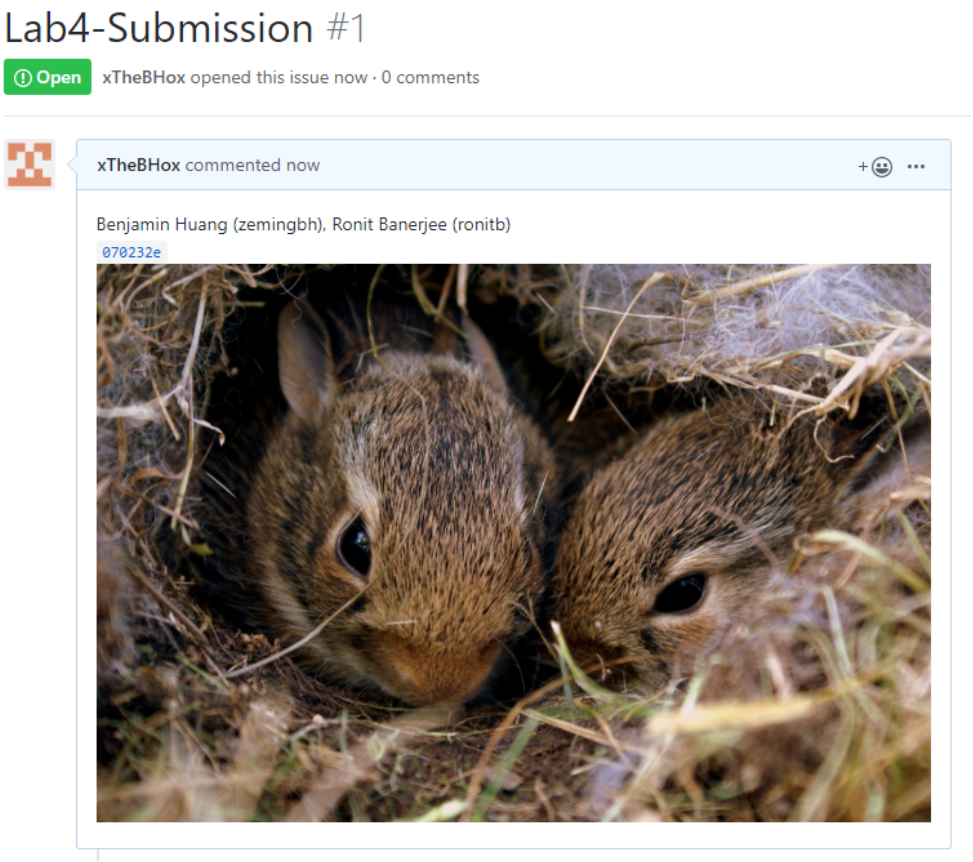


Figure 3: Submission

To demo the final submission, bring your board to any of the TA office hours before the demo deadline and demonstrate the `grade_X` tests. If they do not run correctly, we will grant partial credit by running various `test_X_X`.

14 Style

14.1 Doxygen

Doxygen is a framework that allows you to automatically generate documentation from comments and markup tags inserted directly into the source. This style of embedding documentation in source is often used in industry. We will be using `doxygen` for code documentation for this course moving forward. A tutorial is available here if unfamiliar with it:`http://www.doxygen.nl/manual/docblocks.html`. Steps for setup and use of Doxygen are below:

```
Installation in the VM:
$ sudo apt-get install doxygen

Generating documentation:
$ make doc
```

The handout code contains a default configuration file called `doxygen.conf`. While most of the tags provided in the file already have the correct value, you are responsible for making sure that the `INPUT` tag specifies the correct source files (for this lab and future labs). **Specifically, you will need to have zero doxygen warnings for `kernel`.**

If the above command runs successfully, then you should have a `/doxygen_docs` directory with an `index.html` file. View it locally in a browser to see the documentation created from the code you wrote. When running this command, a file called `doxygen.warn` should have been created in the directory you ran `make doc`. Open this file to inspect any warnings. If there are any documentation warnings in the file about code you have written, then fix them. The TAs will check this file and take off style points if there are *any* warnings in this file.

14.2 Good Documentation

Good code should be mostly self-documenting: your variable names and function calls should generally make it clear what you are doing. Comments should not describe what the code does, but why; what the code does should be self-evident. (Assume the reader knows C better than you do when you consider what is self-evident.)

There are several parts of your code that do generally deserve comments:

- File header: Each file should contain a comment describing the purpose of the file and how it fits in to the larger project. This is also a good place to put your name and email address.
- Function header: Each function should be prefaced with a comment describing the purpose of the function (in a sentence or two), the function's arguments and return value, any error cases that are relevant to the caller, any pertinent side effects, and any assumptions that the function makes.
- Large blocks of code: If a block of code is particularly long, a comment at the top can help the reader know what to expect as they're reading it, and let them skip it if it's not relevant.
- Tricky bits of code: If there's no way to make a bit of code self-evident, then it is acceptable to describe what it does with a comment. In particular, pointer arithmetic is something that often deserves a clarifying comment.
- Assembly: Assembly code is especially challenging to read – it should be thoroughly commented to show its purpose, however commenting every instruction with what the instruction does is excessive.

14.3 Good Use of Whitespace

Proper use of whitespace can greatly increase the readability of code. Every time you open a block of code (a function, "if" statement, "for" or "while" loop, etc.), you should indent one additional level.

You are free to use your own indent style, but you must be consistent: if you use four spaces as an indent in some places, you should not use a tab elsewhere. (If you would like help configuring your editor to indent consistently, please feel free to ask the course staff.)

14.4 Good Variable Names

Variable names should be descriptive of the value stored in them. Local variables whose purpose is self-evident (e.g. loop counters or array indices) can be single letters. Parameters can be one (well-chosen) word. Global variables should probably be two or more words.

Multiple-word variables should be formatted consistently, both within and across variables. For example, "hashtable_array_size" or "hashtableArraySize" are both okay, but "hashtable_arraySize" is not. And if you were to use "hashtable_array_size" in one place, using "hashtableArray" somewhere else would not be okay.

14.5 Magic Numbers

Magic numbers are numbers in your code that have more meaning than simply their own values. For example, if you are reading data into a buffer by doing "fgets(stdin, buf, 256)", 256 is a "magic number" because it represents the length of your buffer. On the other hand, if you were counting by even numbers by doing "for (int i = 0; i ¡ MAX; i += 2)", 2 is not a magic number, because it simply means that you are counting by 2s.

You should use #define to clarify the meaning of magic numbers. In the above example, doing "#define BUFLEN 256" and then using the "BUFLEN" constant in both the declaration of "buf" and the call to "fgets".

This is especially important when putting constants in memory-mapped registers.

14.6 No "Dead Code"

"Dead code" is code that is not run when your program runs, either under normal or exceptional circumstances. These include "printf" statements you used for debugging purposes but since commented. Your submission should have no "dead code" in it.

14.7 Modularity of Code

You should strive to make your code modular. On a low level, this means that you should not needlessly repeat blocks of code if they can be extracted out into a function, and that long functions that perform several tasks should be split into sub-functions when practical. On a high level, this means that code that performs different functions should be separated into different modules; for example, if your code requires a hashtable, the code to manipulate the hashtable should be separate from the code that uses the hashtable, and should be accessed only through a few well-chosen functions.

14.8 Consistency

This style guide purposefully leaves many choices up to you (for example, where the curly braces go, whether one-line "if" statements need braces, how far to indent each level). It is important that, whatever choices you make, you remain consistent about them. Nothing is more distracting to someone reading your code than random style changes.

15 Appendix

15.1 Scheduling Tests

To help you test your kernel, we have provided a large number of user programs. These programs can be run in the same way as your user program from Lab 3. You can find the source code for each program under `user_proj/test_<category_num>` The descriptions that follow tell you the expected output for each of the tests. If something is unclear, you may wish to look through the code and carefully think about what the code should do and where it is not executing correctly.

Test 0_0: Context Saving (1 thread)
This test runs a single thread. Without a scheduler, this test will allow you to test context saving. With a scheduler, this thread will swap with the idle thread.

Test 0_1: Context Swapping (2 threads)
This test creates two identical threads which alternate. This simple test is meant to verify that your kernel properly handles thread creation and context switching.

Test 0_2: Context Swapping (14 threads)
This test creates 14 identical threads, and alternates between them. You should implement the `get_priority` SVC if you want the thread numbers to print.

Test 1_0: Thread vargp
This test checks that the thread argument is passed correctly.

Test 1_1: Thread T
This test checks that threads wake up at the right time (T is properly observed).

Test 1_2: Thread C
This test checks that thread computation time is enforced (C is properly enforced).

Test 1_3: WUNP (1 thread)
This test checks that `wait_until_next_period` is properly implemented by putting a single thread to sleep. Needs the idle thread.

Test 1_4: WUNP (2 threads)
This test checks that `wait_until_next_period` is properly implemented using two threads. Needs the idle thread.

Test 2_0: Idle Thread
This test checks that the idle thread is properly run, saved and restored.

Test 2_2: Stack Size
This test checks that stack size limits are checked in `thread_init`.

Test 2_3: Implicit Idle Thread
This test checks the creation of an implicit idle thread and that `max_threads` supplied in `thread_init` is respected.

Test 3_0: RMS: UB Test
This test tries to create threads that will violate the UB test.

Test 3_1: RMS: Priority
This test checks that thread priority is respected (lower priority number runs first).

Test 3_3: RMS Algorithm: Preemption
This test runs 3 threads, where one of the threads should be preempting the others while they are running.

Test 4_0: Revival: Thread Kill
This test checks that thread killing works and that the user program switches back to the default thread once there are no more threads.

Test 4_1: Revival: Spawner Thread
This test checks that `thread_create` can be called from threads other than the default thread, and that dead threads' resources can be re-used by re-created threads.

Test 4_2: Revival: Unschedulable
This test checks that the UB test still runs even when creating threads after the scheduler has started.

Grade RMS
This test is a benchmark to verify your RMS implementation and the necessary syscalls. This test will still mostly run without thread killing.

Test 5_0: Memory Protection: Buffer Overflow
This test checks that threads cannot access each others' stacks.

Test 5_1: Memory Protection: Heap Access
This test checks that threads have access to the heap but not to each others' stacks.

Grade Revive
This test is a benchmark to test thread killing, revival, and memory protection.

Grade Recursion
This test is a benchmark to verify stack protection, memory protection, and fault handling. You may supply the recursion depth by adding the flag `-n` and specify stack size by adding the flag `-s`. For example, `make flash USER_PROJ=grade_recursion USER_ARG="-n 200 -s 512"`.

Test 6_0: Mutex: No Double Lock/Unlock
This test checks that `mutex_lock` and `mutex_unlock`is properly implemented, and that they ensure a thread does not lock a mutex twice or unlock a mutex twice.

Test 6_1: Mutex: Ceiling
This test checks that thread cannot lock mutexes that have a lower priority ceiling.

Test 6_2: Mutex: Exclusive
This test checks that a mutex cannot be locked by multiple threads.

Test 7_0: Mutex: Yielding
This test checks that higher priority threads yield to lower priority threads when higher priority threads attempt to lock mutexes that lower priority threads hold.

Test 7_1: Mutex: Deyielding
This test checks that a higher priority thread can preempt a lower priority thread once the lower priority thread releases the mutex that the higher priority thread is trying to lock.

Test 7_2, 7_3: Mutex: Nesting
This test checks that nested mutexes work properly.

Test 8_0: PCP: Inheritance
This test checks if priority inheritance only happens when a thread is blocking another thread from locking a mutex.

Test 9_0: Written Problem 1
This is the written problem from Section 7.

Test 9_1: Written Problem 2
This is the written problem from Section 7.

Grade PCP
This test is a benchmark to verify your PCP implementation.

15.2 Implementation Notes

This list gives a summary of the functions in `syscall_thread.c`. For more technical details on the new system call prototypes in `newlib/349include/syscall_thread.h` from the user program prospective refer to the doxygen documentation.

```
int thread_init(uint32_t max_threads, uint32_t stack_size, void *idle_fn,
    protection_mode memory_protection, uint32_t max_mutexes)
```

(Section 3.3, Section 4.5, Section 9) This function gives the kernel a time to initialize the threading data structures. `stack_size` is the size of each thread's user stack and kernel stack and is in units of words (4 bytes each). The kernel should save the `idle_fn` for use when there are no other threads to run. `memory_protection` is a flag indicating if memory protection for thread user-space stacks will be in effect.

```
int thread_create(void *fn, uint32_t prio, uint32_t C, uint32_t T, void *vargp)
```

(Section 3.4) This function adds new threads to the scheduler. The thread should start by running `fn` with argument `vargp`. `prio` is passed by the user. You do not need to determine the priority of threads based on RMS; you can assume the user has already done that properly. The kernel should check the UB admittance test (Section 4.1) to ensure that the resulting threads are schedulable.

```
int scheduler_start(uint32_t freq)
```

(Section 3.7, 4.1) This function tells the kernel to start the scheduler. There must be at least one non-idle thread. This function only ever returns normally if all non-idle threads have returned/been killed.

```
uint32_t get_priority(void)
```

(Section 3.5, Section 11) This call returns the current effective priority of the calling thread. This should be a very straightforward since you have to keep track of this anyway. When PCP is implemented, this should return the inherited priority if it is not the same as the static priority.

```
uint32_t get_time(void)
```

(Section 4.4) Gets the time passed since the scheduler started in terms of ticks. Since your kernel and scheduler have to keep a tick counter anyway, this implementation should be simple. The user tests use this to make your output more readable with timestamps, and also to time user actions.

```
uint32_t thread_time(void)
```

(Section 4.7) This function returns the total actual computation time for the thread since it first ran (the sum over all periods).

```
void wait_until_next_period(void)
```

(Section 4.6) When a thread is done with the work it needs to do in its current period it can call this function to yield to another thread or the idle thread.

```
mutex_t *mutex_init(uint32_t max_prio)
```

(Section 10.1) The user calls this function to request a mutex handle from the kernel. The priority ceiling of the mutex is provided by the user. The kernel does not need to check the validity of the priority ceiling in this function.

```
void mutex_lock(mutex_t *mutex)
```

(Section 10.2) This function should not return until the thread has acquired the mutex. A check should be performed here to make sure a thread is not locking a mutex with insufficient priority ceiling.

```
void mutex_unlock(mutex_t *mutex)
```

(Section 10.2) A thread calls this function to unlock a mutex when it is done with the protected resource. You do not have to worry about threads that capture a mutex across multiple periods.