# Project 5:

**Due Date: 4/23/21 11:59 pm CT - early bird bonus deadline: 4/21/21**

Upcoming dates of interest

| Item | Deadline | Comment |
|---|---|---|
| P5 | 4/23/21 | 5 points if submitted by 4/21/21 |
| P6 | 5/3/21 | 10 points if submitted by 4/30/21 |
| Final | 5/5/21 | |

# Contents



# 1 Introduction

In this project you will add **scheduling of multiple kernel-level threads** to your code base. The emphasis is on **scheduling**. The test code for this project (file `kernel.C`) illustrates that a scheduler is not necessary for multiple threads to happily execute in a system. For example, the following code displays two routines (each executed by a different thread) that explicitly hand the control back and forth, and so achieve a simple form of multithreading:

```
1 void fun1() {
2     console_puts("FUN 1 INVOKED BY THREAD 1\n");
3     for(;;) {
```

```
        < do something ... >
        thread_dispatch_to(thread2);
    }
}
void fun2() {
    console_puts("FUN 2 INVOKED BY THREAD 2\n");
    for(;;) {
        < do something ... >
        thread_dispatch_to(thread1);
    }
}
```

In this piece of code, the CPU is passed back and forth between *thread1* and *thread2*. Although conceptually primitive, this code works well enough until we want to add a third thread. Do we really want to modify the code of `fun2()` to dispatch *thread3* on the CPU? And does the programmer of this third thread really want to know that the CPU needs to be dispatched back to *thread1*? What if she does not want to release the CPU at all? We cannot rely on threads playing nice, so we need a third party that allocates the CPU on behalf of the running threads: we need a **scheduler**.

The interface of the scheduler is defined in file `scheduler.H`. It exports the following functionality:

```
class Scheduler {

  /* The scheduler may need private members... */

public:

   Scheduler();
   /* Setup the scheduler. This sets up the ready queue, for example.
      If the scheduler implements some sort of round-robin scheme,
      then the  end_of_quantum handler is installed in the
      constructor as well. */

   /* NOTE: We are making all functions virtual. This may come in
      handy when you want to derive RRScheduler from this class. */

   virtual void yield();
   /* Called by the currently running thread in order to give up
      the CPU. The scheduler selects the next thread from the ready
      queue to load onto the CPU, and calls the dispatcher function
      defined in 'Thread.H' to do the context switch. */

   virtual void resume(Thread * _thread);
   /* Add the given thread to the ready queue of the scheduler.
      This is called for threads that were waiting for an event to
      happen, or that have to give up the CPU in response to a
      preemption. */

   virtual void add(Thread * _thread);
   /* Make the given thread runnable by the scheduler. This function
```

```
30        is called after thread creation. Depending on implementation,
31        this function may just add the thread to the ready queue,
32        using 'resume'. */
33
34    virtual void terminate(Thread * _thread);
35    /* Remove the given thread from the scheduler in preparation for
36       destruction of the thread.  Graciously handle the case where
37       the thread wants to terminate itself.*/
38 };
```

You will implement a simple First-In-First-Out (FIFO) scheduler. This is very easy to achieve: the scheduler maintains a so-called **ready queue**, which is a list of threads (or, more precisely, their thread control blocks) that are waiting to get to the CPU. One thread is running on the CPU, typically. Whenever a running thread calls `yield()`, the scheduler finds the next thread to run at the head of the ready queue, and calls the dispatcher to invoke a context switch. Whenever the system decides that a thread, say Thread `t1`, should become ready to execute on the CPU again, it calls `resume(t1)`, which adds `t1` to the end of the ready queue. Other threads may be waiting for events to happen, such as for a page fault or some other I/O operation to complete. We call these threads **blocked**. We don't worry about blocked threads in this project, as all the threads are either busy executing or waiting on the ready queue. Later, when threads access devices, we will have to deal with blocked threads as well.

# 2 Implementation of the Scheduler

You need to implement the functions declared in file `scheduler.H`): the constructor and the functions `yield()`, `add()`, and `resume()`. When you are ready to handle terminating thread functions, you may want to implement the function `terminate()` as well.

Depending on how you want to implement the ready queue, you may have to modify the thread control block in `thread.H`.

# 3 Testing your Scheduler

The file `kernel.C` is set up to make testing of your scheduler easier. The file creates four threads, each of which is assigned a different function – called `fun1` for Thread 1 to `fun4` for Thread 4. These thread functions call a function `pass_on_CPU()` to explicitly dispatch the next thread on the CPU in the way described above. A macro `_USES_SCHEDULER_` defines how control is passed among threads in file `kernel.C`. If the macro is defined, then the code makes use of the scheduler to hand control from one thread to the next. For details, look at the source code in `kernel.C`.

# 4 Threads with Terminating Thread Functions

All thread functions defined in `kernel.C` are non-terminating (basically infinite loops). They only stop executing when they pass control to another thread. The system at this point does not know what to do when the thread function returns. In other words, there is no support in the low-level thread management for handling functions that return and therefore stop to execute.

You are to study the thread management code, propose, and implement a solution that allows a thread to cleanly terminate when its thread function returns. You need to worry about releasing the CPU, releasing memory, giving control to the next thread, etc. Test your solution by making the thread functions in `kernel.C` return. You can easily do this by modifying the `for` loops to have an upper bound. For example, modify the line

```
for(int j = 0;; j++)
```

to read

```
for (int j = 0; j < DEFAULT_NB_ITERS; j++)
```

or similar.

A macro `_TERMINATING_FUNCTIONS_` defines whether the thread functions for the first two threads terminate or not. For details, look at the source code in `kernel.C`.

## Opportunities for Bonus Points

If you want to extend your work beyond the bare minimum, here are a few options.

**OPTION 1: Correct handling of interrupts.** (This option carries **6 bonus points**.) You will notice that interrupts are disabled after we start the first thread. One symptom is that the periodic clock update message is missing. This is caused by the way we create threads: we set up the context so that the Interrupt Enable Flag (IF) in the EFLAGS status register is zero, which disables interrupts once the created thread returns from the fake "exception" when it starts. This is ok, because we may not want to deal with interrupts when we are just starting up our thread. But, at some point, we need to re-enable interrupts and turn them off again when we need to ensure mutual exclusion.

It is up to you how to modify the code in the scheduler and the thread management to ensure correct enabling/disabling of interrupts. The functions to do that are provided in `machine.H`.

**OPTION 2: Round-Robin Scheduling:** (This option carries **8 bonus points**.[1]) Once you have interrupts set up correctly, you can expand the simple FIFO scheduler into a basic round-robin scheduler. For this, we generate an interrupt at a periodic interval (say 50 msec). This can be done by either modifying or replacing the interrupt handler for the simple timer. The new interrupt handler triggers preemption of the currently running thread. During the preemption, the currently running thread puts itself on the ready queue and then gives up

[1] Attack this problem only after you have convinced yourself that you are managing interrupts correctly. Otherwise you will waste a lot of time debugging!

the CPU.

If you think this is just a `resume()` followed by a `yield()`, think again. The situation is a bit more complicated than that. For example, the current thread is giving up the CPU inside an interrupt handler, and the new thread may or may not be returning from a preemption, and therefore may or may not be returning from an interrupt handler. In both cases we need to let the interrupt controller (PIC) know that the interrupt has been handled. In addition, we need to stop the original thread from informing the PIC when it returns from the interrupt possibly much later. It is very likely that you will need to modify the low-level exception and interrupt handling code to get this to work correctly.

**OPTION 3: Processes:** (This option carries **20 bonus points** [2].) Until now, all threads share the same address space. Extend the system to allow for the creation of threads that have different address spaces, i.e., implement processes. If you plan to tackle this option, you will have to handle a number of different aspects, including creation of multiple address spaces (multiple page tables,) and switching from one address space to another as part of the thread switch. The code for the thread switch needs to be somewhat extended to handle the switch from one page table to the other. We keep the specification of this option intentionally vague in order to have you come up with your own design. Again: do not underestimate the difficulty of getting this option to work properly.

# 5 About the new kernel.C

The main file for P5 is somewhat similar in nature to the earlier ones. We have removed page tables and paging, and we have added code to initialize threading, for scheduling, and for creating a small number of threads.[3]

# 6 About memory management

You do not need your P2/P3/P4 for this homework. P5 comes with a very simple memory manager that offers operators `new` and `delete`. Now you can implement dynamically allocated data structures such as linked lists easily. The powerful C++ libraries with support for container data structures (e.g., `std::list`, `std::vector`) are NOT available with the provided environment.

[2] This option is very challenging and only for students who feel totally under-challenged. Contact the instructor before starting on this option!

[3] Students who decide to attack Option 3 will have to add paging and page tables back in to test their approach.

# 7 The Assignment

1. Download the provided source code.
2. Implement the routines defined in file `scheduler.H` (and described above) to initialize the scheduler, to add new threads, and to perform the scheduling operations.
3. You may need to modify file `scheduler.H`. If so, document how and why.
4. Modify file `kernel.C` to replace explicit dispatching with calls to the scheduler. This can be done by uncommenting the definition of macro `_USES_SCHEDULER_` that enables scheduling. Details about how to do this are given in file `kernel.C`.
5. Add support for threads with terminating thread functions. This may require modifications to file `thread.C`. If so, document how and why. Test your solution by uncommenting the definition of the appropriate macro in file `kernel.C`.
6. If you decide to pursue one or more of the bonus options, proceed as follows.

   **Option 1** Fix the interrupt management so that interrupts remain enabled outside of critical sections.

   **Option 2** Modify the scheduler to implement round-robin with a 50 msec time quantum. (Note: Since the preemption interrupt arrives periodically, this is not a very good round-robin scheduler. Whenever a thread gives up the CPU voluntarily, the next thread is short-changed.)

   **Option 3** Contact the instructor to discuss your plan to implement multiple address spaces. Modify the thread creation, page table initialization, and thread switching to allow for multiple address spaces. Add page tables back into the code and exercise your new system. It will be up to you to come up with a design, the implementation, and the test suite for your solution.

# 8 What to Hand In

You are to hand in the following items:

- A ZIP file, with name `p5.zip`, containing the following files:
  1. A design document, called `design.pdf` (in PDF format) that describes your design and the implementation of the FIFO scheduler, and any of the selected bonus options. **Clearly identify at the beginning of your design document and in your submitted code which options you have selected, if any.**
  2. Two files, called `scheduler.H` and `scheduler.C`, which contain the definition and implementation of the functions to initialize the FIFO scheduler and to execute the scheduler operations.

3. Two files, called `thread.H` and `thread.C`, which contain the modified definition and implementation of the thread description block. Document what changes you have made to the original files, and why.
4. Submit any other modified file, and clearly identify and comment the portions of code that you have modified.

Grading these project is a tedious chore. These instructions are meant to mitigate the difficulty of grading, and to ensure that the grader does not overlook any of your efforts.

**Failure to follow the handing instructions will result in lost points.**

# 9 Rubric

- Implement the scheduler without thread termination: 75 points
- Support for thread termination: 25 points
- Bonus points
  - Option 1: 6 points
  - Option 2: 8 points
  - Option 3: 20 points

# 10 Document Release Log

- Version 2021-A.0: initial release on 4/12/21