Project 5: Kernel-Level Thread Scheduling
Due: 4/5/16 11:59pm CT

Introduction

In this project you will add **scheduling of multiple kernel-level threads** to your code base. The emphasis is on **scheduling**. There is no dependence on P2/P3/P4; actually P5 does not even enable paging. The test code for this project (as in previous projects, in file `kernel.C`) illustrates that a kernel scheduler is not strictly necessary for getting multiple threads to take turns in execution 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:

```
void fun1() {
    console_puts("FUN 1 INVOKED BY THREAD 1\n");
    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 code excerpt, the CPU is passed back and forth between Thread 1 and Thread 2. And this works fine, until we want to add a third thread. Do we really want to modify the code of `fun2()` to dispatch Thread 3 on the CPU? And does the programmer of Thread 3 really want to know that the CPU needs to be dispatched back to Thread 1? What if she does not want to release the CPU at all? What if Thread 1 is busy waiting for I/O, and not ready to execute? So, we cannot rely on the threads themselves to take care of who should go to the CPU next, and we need a third party that allocates the CPU on behalf of the running threads, making decisions that affect the usage of resources in the whole system. We need a **scheduler**. The interface of the scheduler is defined in file Scheduler.H. It exports the following methods:

```
class Scheduler {
public:
    Scheduler();
    /* Setup the scheduler. This sets up the ready queue, for example.
       If the scheduler implements some sort of round-robin scheme (not
       required of 410 students), then the
       end_of_quantum handler is installed here as well. */

    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 'threads.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. */

    virtual void add(Thread * _thread);
    /* Make the given thread runnable by the scheduler. This function is
       typically  called after a thread has been created. Depending on the
       implementation, this may not entail more than simply adding the thread
       to the ready queue (see resume). */

    virtual void terminate(Thread * _thread);
    /* Remove the given thread from the scheduler in preparation for
       destruction of the thread. */
};
```

You are to implement a simple First-In-First-Out (FIFO) scheduler. This is easy to achieve. The scheduler maintains a so-called **ready queue**, which is a list of threads (or, better, 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 `thread` 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 for now, as for this project all the threads are either busy executing or waiting on the ready queue. (And the paging support we did in P2/P3/P4 was not handling I/O anyway.) In P5, where threads access devices, we will have to deal with blocked threads as well.

Implementation of the Scheduler

You realize the scheduler by implementing the exported functions (declared in file `Scheduler.H`) inside a file called `Scheduler.C`. Start by implementing the constructor and the functions `yield()`, `add()`, and `resume()`. When you are ready to handle terminating thread functions, you should add the implementation of 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` to have the information needed by your choice of data structure.

Testing your Scheduler

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

Threads with Terminating Thread Functions

Currently, there is no support in the low-level thread management for threads with thread functions that return and therefore stop to execute. For example, all the thread functions in `kernel.C` contain infinite loops. The system at this point does not know what to do when the thread function

returns. You are to study the thread management code and 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 lines `for (int j = 0;; j++)` to read `for (int j = 0; j < 10; j++)` or similar. A macro `TERMINATING_FUNCTIONS` defines whether the thread functions for the first two threads terminate after 10 iterations or not. For details, look at the source code in `kernel.C`.

Correct handling of interrupts. 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 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`.

Round-Robin Scheduling I advise you not to start this part until you have convinced yourself that you are managing interrupts correctly. Otherwise you may waste a lot of time debugging!
Once we have interrupts set up correctly, we can expand our simple FIFO scheduler into a basic round-robin scheduler. For this, we want to 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 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 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.

A note about the main routine

The `Kernel.C` file for this project is somewhat similar in nature to the earlier projects. 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.

What to do

1. Get P5-provided-code.zip. Notice that the makefile.linux64 has been fixed to add compile flags that many students needed to have added in P2/P3/P4 (as Piazza posts documented). If you use another makefile, take a look at the changes. The configuration file for bochs has also been modified to enable the use of a macro (in debug.H) for printing at stdout in a way that can be captured in a file. It is likely that there are other things you may need to fix in the Makefile, so please ask and give help in our Piazza forum.
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 a definition of a macro 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. Fix the interrupt management so that interrupts remain enabled outside of critical sections.
7. 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.)

What to Hand In

You are to hand in the following items using eCampus (as in previous projects):

- A ZIP file containing the following files:
    1. A design document (in PDF format) that describes your design and the implementation of the FIFO scheduler, and any of the additional parts you have worked on.
    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. Thread termination may require modifications to other files (such as `thread.C`). If so, document what you have changed and why, and provide your new versions.
    4. Submit any other modified file, and clearly identify and comment the portions of code that you have modified.

Grading of these MPs is a very tedious chore (and also time intensive). These hand-in instructions are meant to mitigate the difficulty of grading, and to ensure that the grader does not overlook any of your efforts

Grading Rubric

Element	Points
(A) = report + all required methods in Scheduler.H were implemented in a reasonable way, the code compiles, and the kernel boots with USES_SCHEDULER defined. Code is appropriately documented.	20
(B) = (A) + FIFO scheduling works for test without implementing support for terminating functions	25
(C) = (B) + test with terminating functions work	25
(D) = (C) + proper interruptions (with corresponding information on the report)	15
(D) + round-robin	15