Machine Problem 5: Kernel-Level Thread Scheduling

Introduction

In this machine problem you will add **scheduling of multiple kernel-level threads** to your code base. Our threads will provide the following interface:

```
/* -- THREAD FUNCTION (CALLED WHEN THREAD STARTS RUNNING) */
typedef void (*Thread_Function)(); /*The thread function takes no arguments and
     returns nothing.*/

class Thread {
public:
Thread(Thread_Function _tf, char * _stack, unsigned int _stack_size);
/* Create a thread that is set up to execute the given thread function. The
    thread is given a pointer to the stack to use. NOTE: _stack points to the
    beginning of the stack area, i.e., to the bottom of the stack. */
int ThreadId();
/* Returns the thread id of the thread. */
static void dispatch_to(Thread * _thread);
/* This is the low-level dispatch function that invokes the context switch code
    . This function is used by the scheduler. NOTE: dispatch_to does not return
    until the scheduler context-switches back to the calling thread. */
static Thread * CurrentThread();
/* Returns the currently running thread. NULL if no thread has started yet. */
};
```

The emphasis in this MP is on **scheduling**. The test code for this MP (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:

```
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 piece of code 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? So, we cannot rely on the threads to play nice, and 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 {
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 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 class Thread 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 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 Scheduler::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 very 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 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 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.

Implementation of the Scheduler

You realize the scheduler by implementing the exported functions (declared in file `scheduler.H`) inside a file called `scheduler.C`. Implement primarily 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`.

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 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`.

Threads with Terminating Thread Functions

Currently, all thread functions 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 threads with thread functions that return and therefore stop executing. 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 (see below), 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 < DEFAULT_NB_ITERS; 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`.

Releasing resources allocated by the thread on return of the thread function can be slightly tricky. For example, the thread cannot release its own stack. If it did, the thread would continue running without a stack, which in general would lead to errors. The best way to handle this is to have the terminating thread "park" itself somewhere and have other threads handle the release of resources. This approach is used in some operating systems, where we have a "zombie queue" of threads that own resources that need to be released before the thread can be completely terminated, hence the term "zombie". Handling zombie threads can be done by a separate thread, or it can be part of some regularly called function in the scheduler, such as the `resume` function.

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 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 **6 bonus points**.[1]) 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).

[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!

You should preferably do this by deriving a class `EOQTimer` from the provided `SimpleTimer` that implements the end-of-quantum functionality for the scheduler. In `kernel.C` you then construct an EOQ timer instead of a simple timer. The interrupt handler of the EOQ timer then triggers the 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 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 **12 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; basically, implement processes. If you plan to attack 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 this option!

A Note about the new Main File

The main file for this MP is somewhat similar in nature to the earlier MPs. 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.

The two classes `ContFramePool` (defined in files `cont_frame_pool.C/H`) and `VMPool` (defined in files `vm_pool.C/H`), which together implement virtual memory pools in previous MPs, have been replaced by the two very simplified classes `FramePool` (defined in files `frame_pool.C/H`) and `MemPool` (defined in files `mem_pool.C/H`), which implement a very simple physical-memory allocator. You will not have to worry about memory management in this MP. In particular, there will be no virtual memory, and paging will not be turned on.[3]

The Assignment

1. Download the provided source code for this MP.
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.

[2] This option is very challenging and only for students who feel totally under-challenged. You absolutely have to contact the instructor before starting on this option! If you do not, we will not consider your submission for this option.

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

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 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.

What to Hand In

You are to hand in a ZIP file, with name `mp5.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 options. **Clearly identify at the beginning of your design document and in your submitted code which options you have selected, if any.**

2. All the source files and the `makefile` needed to compile the code.

3. The assignment should not require modifications to files other than `scheduler.H/C` and `thread.H/C`, in addition to the implementation of the timers. Any modifications to these files must be documented. Any modification to other files must be clearly motivated and documented. **Be very careful when you modify the public interface provided by the `.H` files.** We are using our own test code to check the correctness of your implementations, and we have to ensure that your code is compatible with our tests.

4. Clearly identify and comment changes to existing code.

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

Note: Pay attention to the capitalization in file names. For example, if we request a file called `file.H`, we want the file name to end with a capital `H`, not a lower-case one. While Windows does not care about capitalization in file names, other operating systems do. This then causes all kinds of problems when the TA grades the submission.

Failure to follow the handin instructions will result in lost points.