MP5: Simple Kernel Threads

Nikhil Nunna
UIN: 525008698
CSCE611: Operating System

Assigned Tasks

Main: Completed.
Option I: Omitted.
Option II: Omitted.
Option III: Omitted.

System Design

The goal of this program was to create a Scheduler that could manage threads. This is done by managing a ready queue or list of threads. When a thread is either ready to return or wishes to pass off the CPU it tells the scheduler. In the first case where the thread tries to shutdown it calls the scheduler from the thread shutdown function and tells the scheduler that it (the currently executing thread) wants to shutdown and that it needs to be removed from the list or processes, so first we change the currently executing thread to the next one in the list and then we de-allocate the thread in it's shutdown function. In the latter case where the thread wants to just pass off the CPU to the next thread in the list, we just add the current thread to the last position in the list and then pass the CPU off to the next thread in the list.

Code Description

The code compiles and runs normally. All the grader needs to do is run 'make', './copykernel.sh' and then 'bochs -f bochsrc.bxrc'. As an alternative to this, I've created test.sh which can be run using './test.sh'. The test.sh shell script just runs make, copykernel and the bochs command for you.

test.sh: code block : This shell file compiles and runs the project

```
#!/bin/bash

make
./copykernel.sh
bochs -f bochsrc.bxrc
```

Thread.C: getnextthread : In this function, I return the private thread pointer variable I added to the Thread control block to help point at the next thread in the ready queue. I had to add this public function to get the private variable I created in the Thread class when in the scheduler class functions.

```
Thread * Thread::getnextthread(){
    /* Returns the next pointer from private part of the object/
        next thread in ready queue. nullptr if there is no next thread
        yet. */
    return next;

};
```

Thread.C: setnextthread : In this function, I set the private thread pointer variable I added to the Thread control block to help point at the next thread in the ready queue. I had to add this public function to set the private variable I created in the Thread class from the scheduler class functions.

```
void Thread::setnextthread(Thread * _thread){
    /* Sets the next pointer in private part of the object/
        next thread in ready queue. */
    next = _thread;
};
```

Thread.C: delete_stack : In this function, I delete the stack of the thread I'm shutting down. I had to add this public function to delete the stack from the static shutdown function.

```
void Thread::delete_stack(){
    delete [] stack;
};
```

Thread.C: thread_shutdown : In this function, I shut down the thread that returns, this is done by calling yield in the scheduler to update the current thread after de-allocating any heap allocations made for and in the thread object.

```
static void thread_shutdown() {
    /* This function should be called when the thread returns from the thread function.
       It terminates the thread by releasing memory and any other resources held by the
       This is a bit complicated because the thread termination interacts with the sched
     */
    // Copy the current thread as the pointer is about to change
    Thread * remove_thread = current_thread;
    // Remove the thread's stack
    remove_thread->delete_stack();
```

```
    // Deallocate the thread object
    delete remove_thread;
    // Update the current thread
    SYSTEM_SCHEDULER->yield();

}
```

scheduler.C: constructor : In this function, I set initialize the ready queue by setting up the head and tail pointers.

```
Scheduler::Scheduler() {
  head = nullptr;
  tail = nullptr;
  Console::puts("Constructed Scheduler.\n");
}
```

scheduler.C: yield : In this function, I get the next thread off the ready queue/list, check that there is another thread in the queue after it. then call dispatch to change the current thread.

```
void Scheduler::yield() {
  // get the next thread in ready queue
  Thread * callthread = head;
  // remove the current one from queue
  head = head->getnextthread();
  // check if next is nullptr
  if (head == nullptr) {
    tail = nullptr;
  }
  // dispatch for context switch
  Thread::dispatch_to(callthread);
}
```

scheduler.C: resume : In this function, I append the thread to resume to the back of the ready queue/list if there are other threads in the list otherwise I put it at the front of the list.

```
void Scheduler::resume(Thread * _thread) {
  // Check if anything in ready queue
  // if nothing place at head of ready queue
  if (head == nullptr)
  {
    head = _thread;
    head->setnextthread(nullptr);
    tail = _thread;
  }
  // If others in queue append to end of queue
  // then set as new end of queue
  else
  {
    tail->setnextthread(_thread);
    tail = _thread;
    tail->setnextthread(nullptr);
  }
}
```

scheduler.C: add : In this function, I just use as a wrapper for the resume function since add and resume have the same functionality in my implementation.

```
void Scheduler::add(Thread * _thread) {
  // call resume as it has same functionality
  resume(_thread);
}
```

scheduler.C: terminate : In this function, I check where in the ready queue the thread that needs to be removed is and then remove it by adjusting the preceding pointer before it.

```
void Scheduler::terminate(Thread * _thread) {
  // set variable to traverse ready queue
  Thread * check_thread = head;
  // case where thread to be removed is head
  if (_thread == check_thread)
  {
    // if tail is equal to head set both to nullptr
    if (tail == head)
    {
      head = nullptr;
      tail = nullptr;
      return;
    }
    // otherwise just set head to next in queue
    else
    {
      head = head->getnextthread();
      return;
    }
  }
  // control for loop
  bool inqueue = true;
  // loop thru ready queue till we find thread to remove
  while (inqueue && (check_thread->getnextthread() != nullptr))
  {
    // found thread to remove
    if (_thread == check_thread->getnextthread())
    {
      //remove thread
      Thread * swap = check_thread->getnextthread();
      check_thread->setnextthread(swap->getnextthread());
      inqueue = false;
    }
    // get next thread
    check_thread = check_thread->getnextthread();
  }
}
```

Testing

I only added testing for the terminate function which I commented out in kernel.C. The only thing the test did was call terminate on thread 3 after adding all the threads. I didn't add any further testing since I believe that the given testing covers all of my code. I spent about 2 hours using the testing given to remove a bug from my code. The error I caught was in the yield function. I was skipping thread 2 as I was removing the head node from the list and in the first iteration when we called resume and

yield in the kernel I misinterpreted the code such that I thought we were called adding the first thread to the back of the ready queue and yielding to the second thread. However, this implementation does not work as the head node is thread 2 as thread 1 was never added to the scheduler in the Kernel. I found and corrected this error using the tests given to us. I feel confident that they provide enough coverage as they check the full functionality of the scheduler including all the edge cases I can think of as demonstrated by the failure I encountered.