MP5: Kernel-Level Thread Scheduling

Kai-Chih Huang
UIN: 333000021
CSCE611: Operating System

Assigned Tasks

Main Part: Finished
Bonus Option 1: Finished
Bonus Option 2: Not Finished
Bonus Option 3: Not Finished

System Design

The goal of this machine problem is to implement scheduling of multiple kernel-level threads. Basically I had built the scheduler by using a TQueue Object, which is implemented by a linked list, to manage the FIFO scheduling. As for the Bonus Option 1, I manage the interrupt by turning it off whenever the scheduler function is operating. After the function(or method) finished its job, then I turn the interrupt back on. This can prevent the interrupt when we are scheduling, causing unexpected consequenses.

Code Description

I changed thread.H, thread.C, scheduler.H, scheduler.C for this machine problem. To compile code, simply run following command lines under MP4_Sources directory:

```
$ make clean    // clean the old compile files before we compile

$ make          // compile files

$ ./copykernel.sh (if permission denied, try chmod u+x ./copykernel.sh then do it again)    //copy kernel

$ bochs -f bochsrc.bxrc     // run bochs
```

I will walk through the functions/methods defined in above files as follows.

thread.H / thread.C
I added a class function delete_thread. When the thread is terminated, the scheduler will call this function to delete the stack variable when the thread is terminated.

```
void delete_thread();
/* Clean the memory of the thread */

void Thread::delete_thread() {
    delete stack;
}
```

scheduler.H
I added a class TQueue to manage threads. The structure is basically a linked list, with static head and tail variable pointing to the first and last TQueue object. The object has 2 attributes, which is the thread it is representing, and the pointer to the next TQueue object that contains the next thread. Beside constructing function, I designed 2 methods, which is add and pop, which is to add new threads and pop the oldest thread.

```
class TQueue {
   public:
      static TQueue * head;
      static TQueue * tail;
      Thread * thread;
      TQueue * next;

      TQueue(Thread * _thread);
      static void add(TQueue * _queue);
      static Thread * pop();
};
```

After we define the TQueue class, we add a new private variable to the scheduler so that we can use our linked list to manage the threads in the scheduler.

```
private:
   // TQueue * ready_queue;
   TQueue * ready_queue;

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

scheduler.C

In this file we implemented functions of TQueue class and Scheduler class as follows:

TQueue(Thread * _thread):

The constructing function of the class. It simply set the thread to its attribute

```
TQueue::TQueue(Thread * _thread) {
  thread = _thread;
  next = NULL;
}
```

void TQueue::add(TQueue * _queue):

This static function add a new TQueue object (which is basically a node contains a thread) to the tail, which is end of the TQueue node linked list.

```
void TQueue::add(TQueue * _queue) {
  if(head == NULL){
    head = _queue;
    tail = _queue;
  }
  else{
    tail->next = _queue;
    tail = _queue;
  }
}
```

Thread * TQueue::pop():

This function remove a thread node from the beginning of the linked list.

```
Thread * TQueue::pop() {
  Thread * deleting_thread = head->thread;
  TQueue * temp_node = head;
  if(head == tail){
    head = NULL;
    tail = NULL;
  }
  else{
    head = head->next;
  }
  delete temp_node;
  return deleting_thread;
}
```

The following will be the implementation of the functions of Scheduler Class. Note that for all the functions besides constructor I disable the interrupt at the beginning of the function,

```
if(Machine::interrupts_enabled()) Machine::disable_interrupts();
```

and enable the interrupt after the function finish its job.

```
if(!Machine::interrupts_enabled()) Machine::enable_interrupts();
```

Scheduler::Scheduler():

This is the constructing function of the scheduler object. Since we have already defined our TQueue object in the header file. We don't have to do anything in this function.

```
Scheduler::Scheduler() {
  Console::puts("Constructed Scheduler.\n");
}
```

void Scheduler::yield():

This function is called by a current thread to give up CPU. If there is no thread in the queue, the function will print the message and do nothing. If there is other threads in the queue, the function will call TQueue:pop() to get the earliest thread in the queue and to the context switch

```
void Scheduler::yield() {
  if(Machine::interrupts_enabled()) Machine::disable_interrupts();
  Console::puts("Yielding...\n");

  if(ready_queue->head == NULL) {
    Console::puts("No threads to run.\n");
    if(!Machine::interrupts_enabled()) Machine::enable_interrupts();
    return;
  }else {
    Thread * next_thread = TQueue::pop();
    Console::puts("Next thread to run: "); Console::puti(next_thread->ThreadId()); Console::puts("\n");
    Thread::dispatch_to(next_thread);
  }

  Console::puts("Yield finish.\n");
  if(!Machine::interrupts_enabled()) Machine::enable_interrupts();
}
```

void resume(Thread * _thread):

This function add the given thread to the ready queue. It simply create a new TQueue object and enqueue by calling TQueue add method

```
void Scheduler::resume(Thread * _thread) {
  if(Machine::interrupts_enabled()) Machine::disable_interrupts();
  Console::puts("Resuming...\n");

  TQueue * new_queue = new TQueue(_thread);
  TQueue::add(new_queue);

  Console::puts("Resume thread to ready queue: "); Console::puti(_thread->ThreadId()); Console::puts("\n");
  if(!Machine::interrupts_enabled()) Machine::enable_interrupts();
}
```

void Scheduler::add(Thread * _thread):

This function is called after thread creation to add the thread to queue. Since the purpose is to add it to the queue, we can just call resume() function to enqueue the thread.

```
void Scheduler::add(Thread * _thread) {
  if(Machine::interrupts_enabled()) Machine::disable_interrupts();
  Console::puts("Adding thread: "); Console::puti(_thread->ThreadId()); Console::puts("\n");

  resume(_thread);

  if(!Machine::interrupts_enabled()) Machine::enable_interrupts();
}
```

void Scheduler::terminate(Thread * _thread):

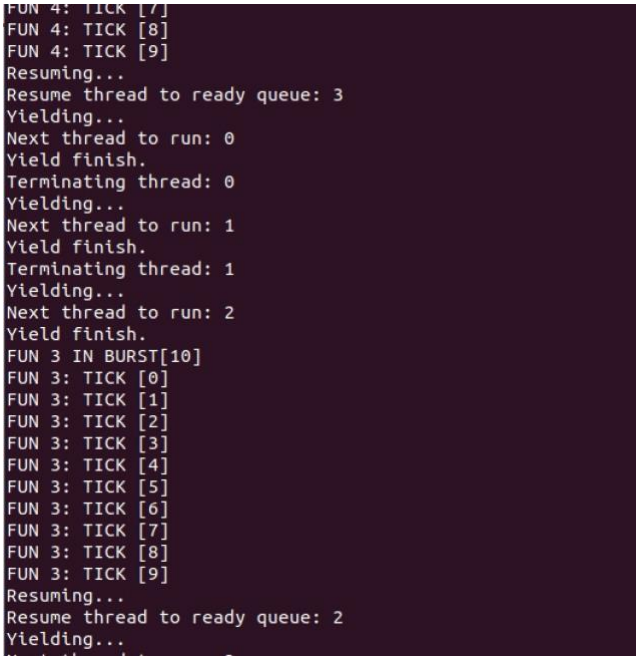
This function is called to terminate a given thread. The if statement is to prevent the situation that if a thread call this function to terminate another thread, which is not the case that we will deal with in this MP. If the thread terminates itself, we will call the delete_thread() to delete the stack of the thread, then give up CPU to another thread in the queue.

```
void Scheduler::terminate(Thread * _thread) {
  if(Machine::interrupts_enabled()) Machine::disable_interrupts();

  Console::puts("Terminating thread: "); Console::puti(_thread->ThreadId()); Console::puts("\n");
  if(_thread == Thread::CurrentThread()) {
    _thread->delete_thread();
    yield();
  }else {
    // Currently we don't deal with this case.
    Console::puts("Thread terminating another thread.\n");
  }

  if(!Machine::interrupts_enabled()) Machine::enable_interrupts();
}
```

Testing

I did not add additional test data in this MP5, since I believe the provided test set is sufficient to check if the page table system works as expected. The test result is shown as follows.

Following screenshot shows that the threads can terminate appropriately.



As for the Option Bonus 1, I correctly handled interrupt so that the SimpleTimer can function normally. Following screenshot shows that timer message “One second has passed” is printed in the middle of the scheduling.