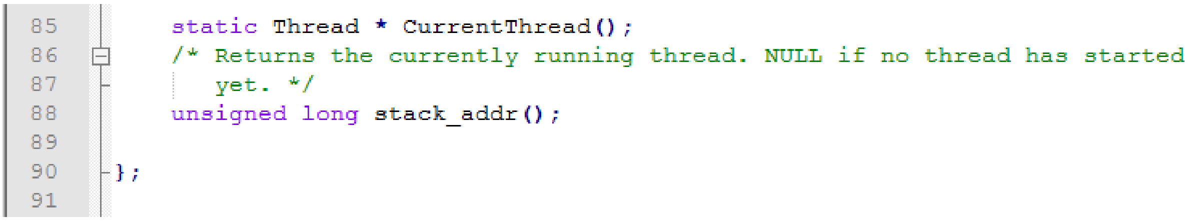
CSCE611 Liuyi Jin
225009797

In this project, our primary goal is to design an appropriate thread scheduler to handle switches among 4 threads. Besides, I also completed the **Option 1** bonus part. I will begin with the file with least modifications.

thread.H

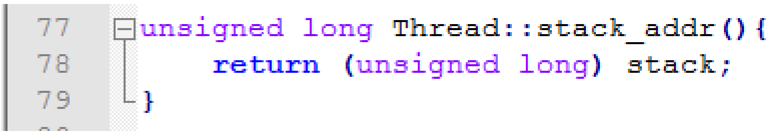


```
85        static Thread * CurrentThread();
86        /* Returns the currently running thread. NULL if no thread has started
87           yet. */
88        unsigned long stack_addr();
89
90    };
91
```

In the last part of thread.H, I declared a function stack_addr(), which is located in the line 88. This is used to return the field char pointer stack.

thread.C

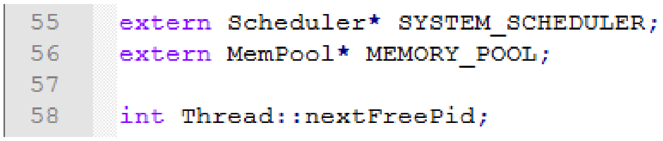
part1



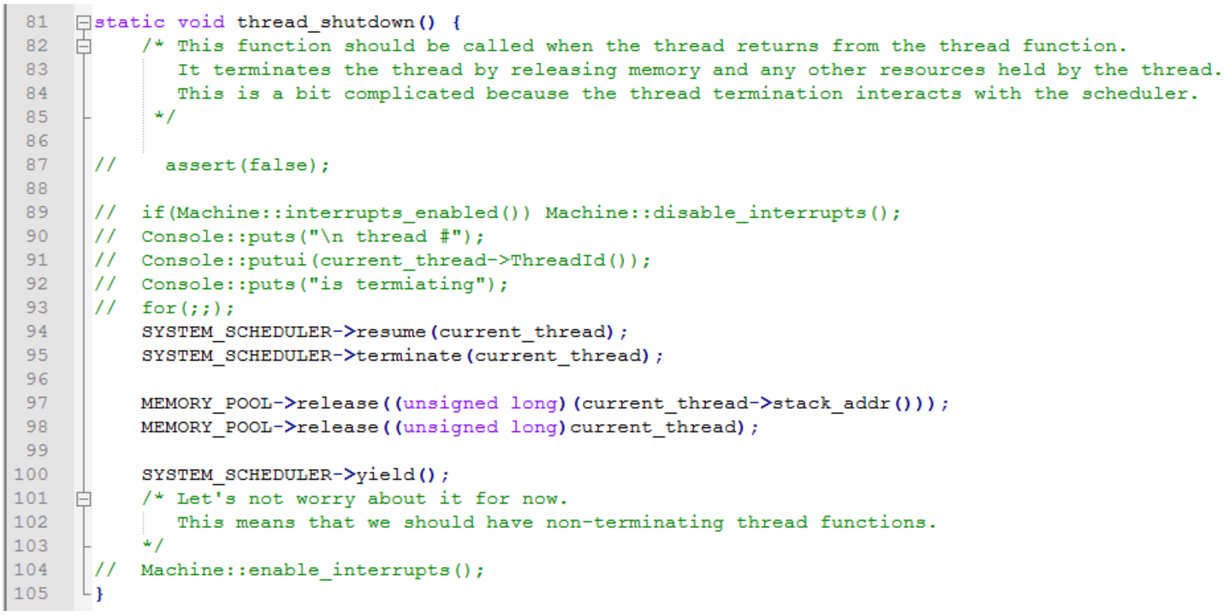
```
unsigned long Thread::stack_addr(){
    return (unsigned long) stack;
}
```

Correspondingly, the definition of the function stack_addr() returns the stack pointer.

part2



```
extern Scheduler* SYSTEM_SCHEDULER;
extern MemPool* MEMORY_POOL;

int Thread::nextFreePid;
```

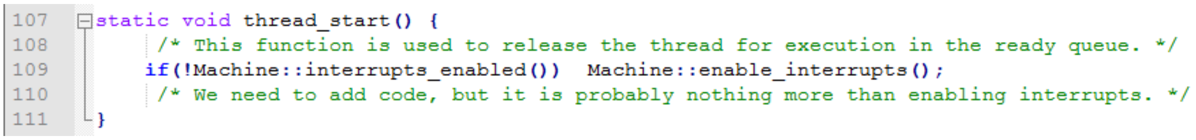


```
static void thread_shutdown() {
    /* This function should be called when the thread returns from the thread function.
       It terminates the thread by releasing memory and any other resources held by the thread.
       This is a bit complicated because the thread termination interacts with the scheduler.
     */

//    assert(false);

//  if(Machine::interrupts_enabled()) Machine::disable_interrupts();
//  Console::puts("\n thread #");
//  Console::putui(current_thread->ThreadId());
//  Console::puts("is termiating");
//  for(;;);
    SYSTEM_SCHEDULER->resume(current_thread);
    SYSTEM_SCHEDULER->terminate(current_thread);

    MEMORY_POOL->release((unsigned long)(current_thread->stack_addr()));
    MEMORY_POOL->release((unsigned long)current_thread);

    SYSTEM_SCHEDULER->yield();
    /* Let's not worry about it for now.
       This means that we should have non-terminating thread functions.
    */
//  Machine::enable_interrupts();
}
```

CSCE611 Liuyi Jin
225009797

In the shutdown function, I add the code to let the current thread firstly add itself to the end of ready queue and then terminate itself. After that, it need to release the memory that it is allocated before. Finally, the scheduler takes over to grab the first thread in the ready queue to execute. To implement these, I have declared two extern variables **SYSTEM_SCHEDULER** and **MEMORY_POOL**.

part3

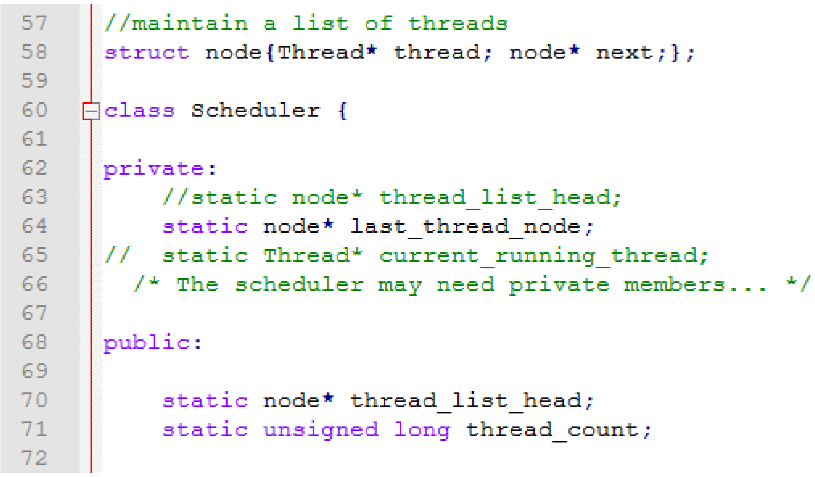


```
static void thread_start() {
    /* This function is used to release the thread for execution in the ready queue. */
   if(!Machine::interrupts_enabled())  Machine::enable_interrupts();
    /* We need to add code, but it is probably nothing more than enabling interrupts. */
}
```

The third part modifications that I have made to the thread.C file involves with the interrupt handling. Which is also the bonus part.

In the thread_start function, I enabled interrupt by firstly making sure the interrupt is disabled, otherwise, there would be errors asserted.

scheduler.H

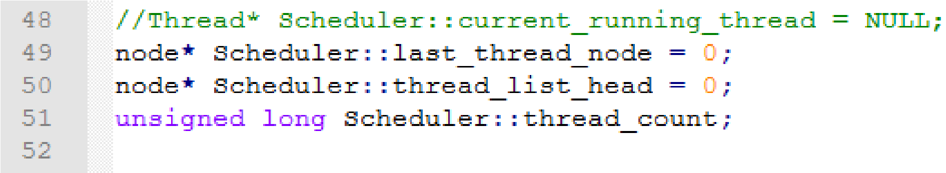


```
//maintain a list of threads
struct node{Thread* thread; node* next;};

class Scheduler {

private:
    //static node* thread_list_head;
    static node* last_thread_node;
//  static Thread* current_running_thread;
  /* The scheduler may need private members... */

public:

    static node* thread_list_head;
    static unsigned long thread_count;

```

To implement the scheduler, I have added several fields to the class Scheduler and one struct to maintain a ready queue for the scheduler. The **last_thread_node** and **thread_list_head** pointed to the head and tail of the thread node list, respectively.

scheduler.C

part1

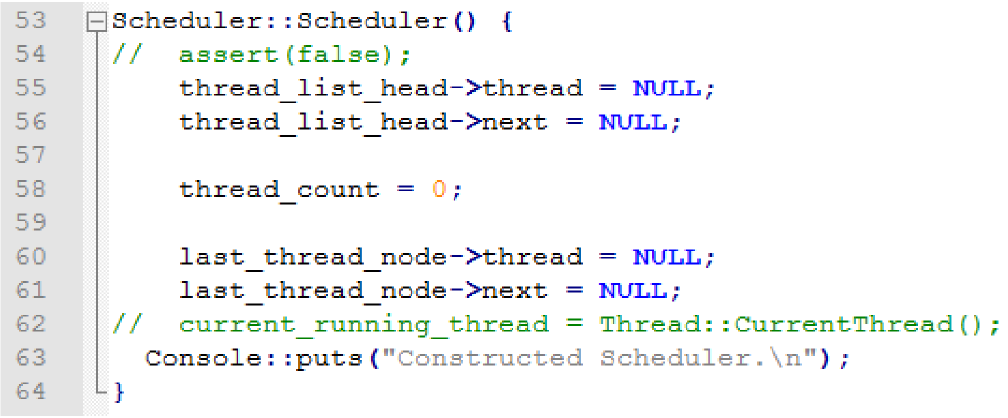


```
//Thread* Scheduler::current_running_thread = NULL;
node* Scheduler::last_thread_node = 0;
node* Scheduler::thread_list_head = 0;
unsigned long Scheduler::thread_count;

```

Before I implemented the functions in the scheduler, I set the fields ahead of everything in the thread.C, and initialize them.

CSCE611 Liuyi Jin
225009797

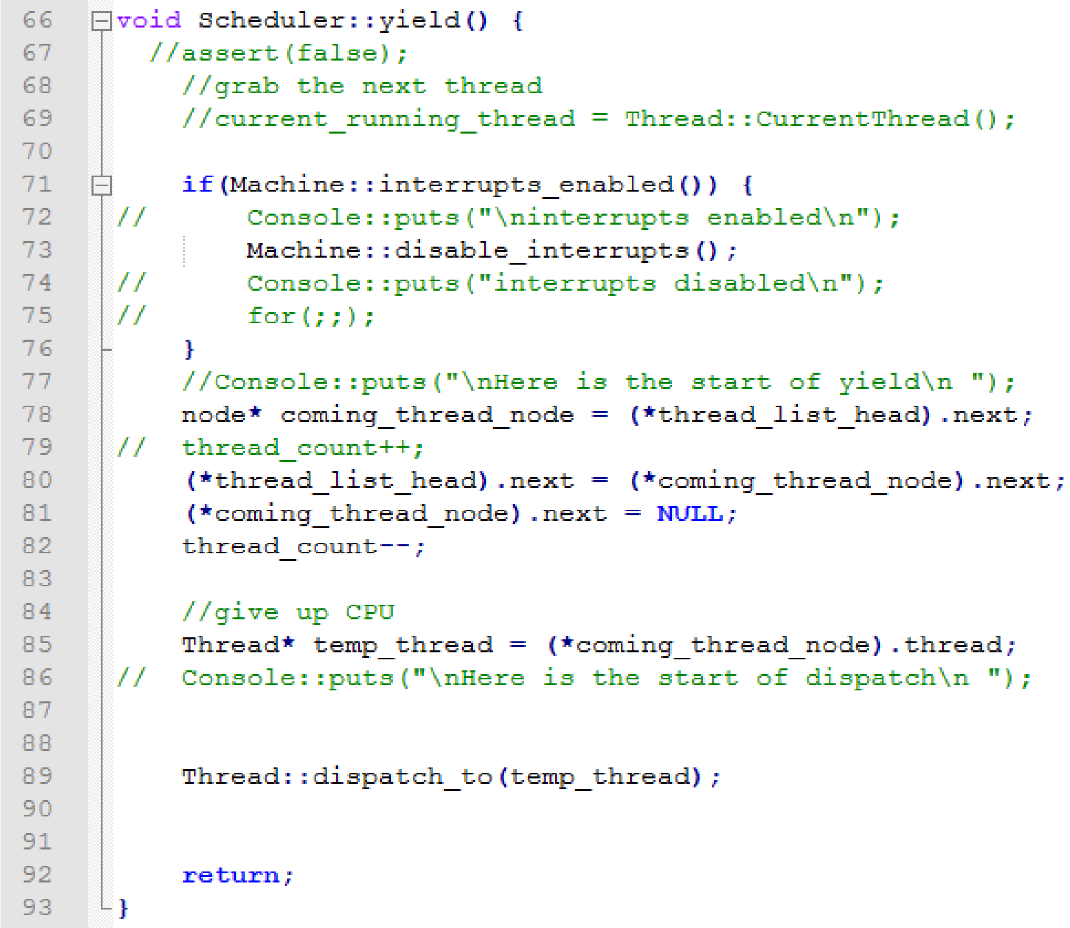
part2



```
Scheduler::Scheduler() {
//  assert(false);
    thread_list_head->thread = NULL;
    thread_list_head->next = NULL;

    thread_count = 0;

    last_thread_node->thread = NULL;
    last_thread_node->next = NULL;
//  current_running_thread = Thread::CurrentThread();
  Console::puts("Constructed Scheduler.\n");
}
```

In the constructor, I initialized the head node and tail node, and also the counter, which is used to count the number of the nodes in the ready queue list.

part3

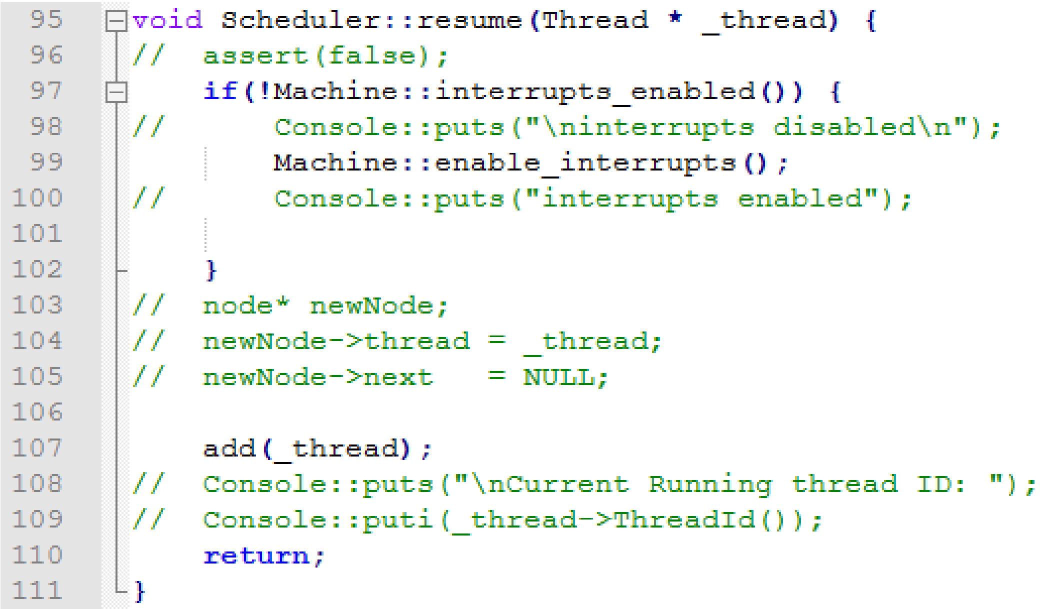


```
void Scheduler::yield() {
  //assert(false);
    //grab the next thread
    //current_running_thread = Thread::CurrentThread();

    if(Machine::interrupts_enabled()) {
//      Console::puts("\ninterrupts enabled\n");
        Machine::disable_interrupts();
//      Console::puts("interrupts disabled\n");
//      for(;;);
    }
    //Console::puts("\nHere is the start of yield\n ");
    node* coming_thread_node = (*thread_list_head).next;
//  thread_count++;
    (*thread_list_head).next = (*coming_thread_node).next;
    (*coming_thread_node).next = NULL;
    thread_count--;

    //give up CPU
    Thread* temp_thread = (*coming_thread_node).thread;
//  Console::puts("\nHere is the start of dispatch\n ");


    Thread::dispatch_to(temp_thread);


    return;
}
```

In the yield function, I disabled the interrupts to ensure the mutual exclusion for the coming thread. After that, I grab the first node in the list and dispatch the CPU to it. Additionally, I reduce the counter by 1.

CSCE611 Liuyi Jin
225009797

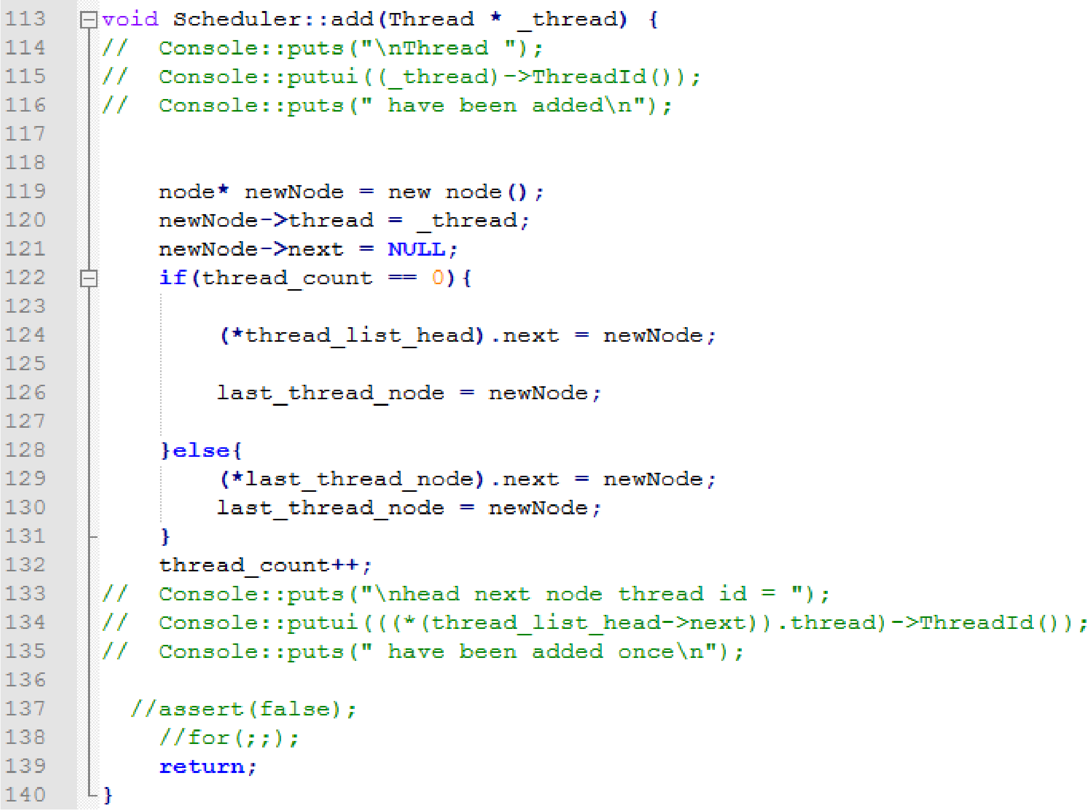
part4



```
void Scheduler::resume(Thread * _thread) {
//  assert(false);
    if(!Machine::interrupts_enabled()) {
//      Console::puts("\ninterrupts disabled\n");
        Machine::enable_interrupts();
//      Console::puts("interrupts enabled");

    }
//  node* newNode;
//  newNode->thread = _thread;
//  newNode->next   = NULL;

    add(_thread);
//  Console::puts("\nCurrent Running thread ID: ");
//  Console::puti(_thread->ThreadId());
    return;
}
```

In the fourth part, I implement the resume function, which is nothing more than adding the given thread to the end of ready queue. Before I add that thread, I enabled the interrupts. Enabling the interrupts in this part involves the option 1 bonus part to correctly handling interrupts.

part5

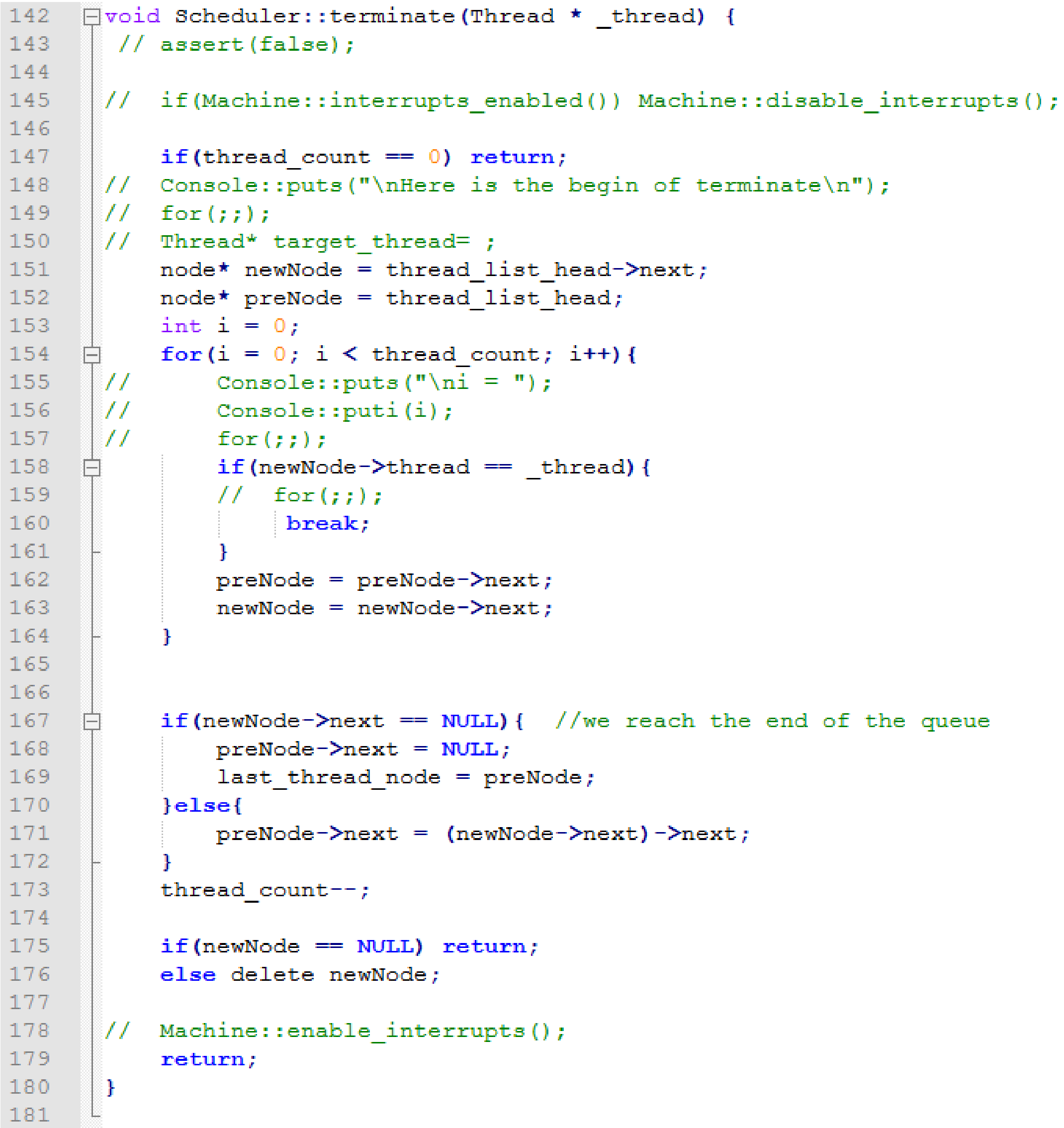


```
void Scheduler::add(Thread * _thread) {
//  Console::puts("\nThread ");
//  Console::putui((_thread)->ThreadId());
//  Console::puts(" have been added\n");


    node* newNode = new node();
    newNode->thread = _thread;
    newNode->next = NULL;
    if(thread_count == 0){

        (*thread_list_head).next = newNode;

        last_thread_node = newNode;

    }else{
        (*last_thread_node).next = newNode;
        last_thread_node = newNode;
    }
    thread_count++;
//  Console::puts("\nhead next node thread id = ");
//  Console::putui(((*(thread_list_head->next)).thread)->ThreadId());
//  Console::puts(" have been added once\n");

  //assert(false);
    //for(;;);
    return;
}
```

CSCE611 Liuyi Jin
225009797

In the add function, things I have done are nothing more than adding the given thread to the end of the ready queue. Here is where the last_thread_node come to involve, this kind of operations is similar to insert the node at the tail of list.

part6



```
void Scheduler::terminate(Thread * _thread) {
 // assert(false);

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

    if(thread_count == 0) return;
//  Console::puts("\nHere is the begin of terminate\n");
//  for(;;);
//  Thread* target_thread= ;
    node* newNode = thread_list_head->next;
    node* preNode = thread_list_head;
    int i = 0;
    for(i = 0; i < thread_count; i++){
//      Console::puts("\ni = ");
//      Console::puti(i);
//      for(;;);
        if(newNode->thread == _thread){
        //  for(;;);
            break;
        }
        preNode = preNode->next;
        newNode = newNode->next;
    }


    if(newNode->next == NULL){  //we reach the end of the queue
        preNode->next = NULL;
        last_thread_node = preNode;
    }else{
        preNode->next = (newNode->next)->next;
    }
    thread_count--;

    if(newNode == NULL) return;
    else delete newNode;

//  Machine::enable_interrupts();
    return;
}
```

In the terminate function, things are a little bit complicated, since we need to locate the node in the list and then we remove that node from list. After I have done the node element extraction, I freed the node memory and reduce the counter by 1.