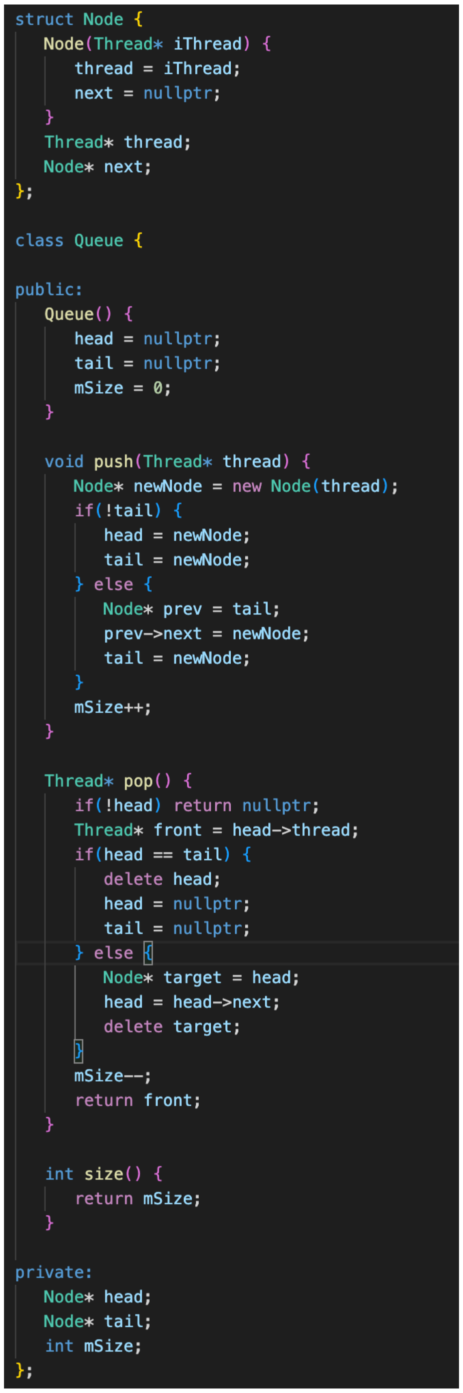
MP5 design

In kernel.C, we create thread 1, 2, 3, 4 and add thread 2, 3, 4 to scheduler. Dispatch thread 1 and scheduler is managing running of threads. Thread 1, 2, 3, 4 will take turns. And when enable _TERMINATING_FUNCTIONS_, it terminates thread 1, 2. Thus, thread 3, 4 will take turns.

We need to modify scheduler.H and scheduler.C to provide scheduling of thread and modify thread.H and thread.C to provide termination.



```
struct Node {
    Node(Thread* iThread) {
        thread = iThread;
        next = nullptr;
    }
    Thread* thread;
    Node* next;
};

class Queue {

public:
    Queue() {
        head = nullptr;
        tail = nullptr;
        mSize = 0;
    }

    void push(Thread* thread) {
        Node* newNode = new Node(thread);
        if(!tail) {
            head = newNode;
            tail = newNode;
        } else {
            Node* prev = tail;
            prev->next = newNode;
            tail = newNode;
        }
        mSize++;
    }

    Thread* pop() {
        if(!head) return nullptr;
        Thread* front = head->thread;
        if(head == tail) {
            delete head;
            head = nullptr;
            tail = nullptr;
        } else {
            Node* target = head;
            head = head->next;
            delete target;
        }
        mSize--;
        return front;
    }

    int size() {
        return mSize;
    }

private:
    Node* head;
    Node* tail;
    int mSize;
};
```

Scheduler in this project is based on FIFO strategy and we need to implement ready queue to provide scheduling.

Node of ready queue should contain thread and pointing next node.

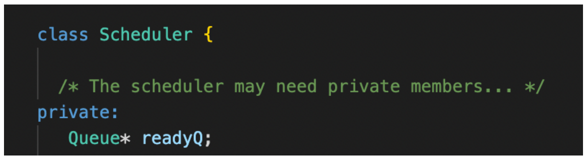
And Queue will contain head and tail pointer. Head will point the first Node of queue and tail pointer will point last node of queue.

Push method makes new Node using parameter thread and push to the last of queue using tail pointer.

Pop method delete first Node in the queue and return the thread in the first Node.

To provide terminate function in scheduler, size method and member variable is implemented.

Scheduler
In scheduler.H

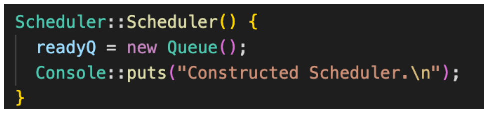


```
class Scheduler {

  /* The scheduler may need private members... */
private:
   Queue* readyQ;
```

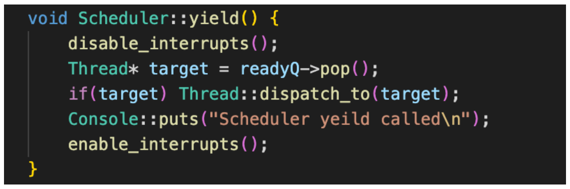
Scheduler has Queue* readyQ to provide FIFO scheduling.

In scheduler.C



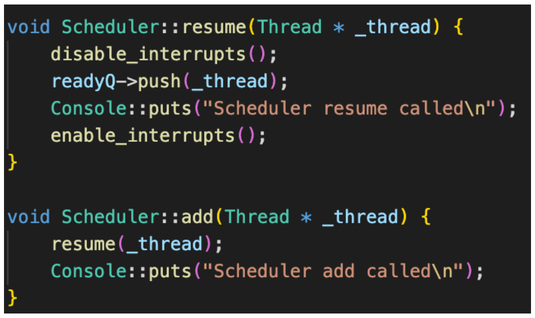
```
Scheduler::Scheduler() {
  readyQ = new Queue();
  Console::puts("Constructed Scheduler.\n");
}
```

Constructor of scheduler will make Queue to the heap and pointing it using readyQ pointer



```
void Scheduler::yield() {
    disable_interrupts();
    Thread* target = readyQ->pop();
    if(target) Thread::dispatch_to(target);
    Console::puts("Scheduler yeild called\n");
    enable_interrupts();
}
```

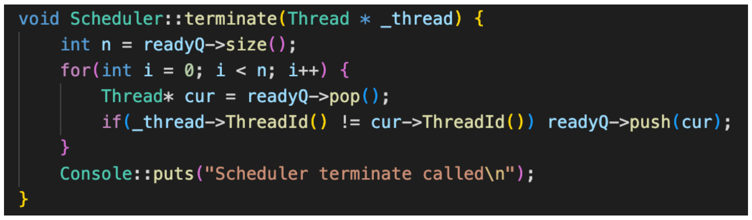
Yield method in scheduler will get next target thread from ready queue and dispatch to target. Disable and enable interrupts is implemented to support additional handling.



```
void Scheduler::resume(Thread * _thread) {
    disable_interrupts();
    readyQ->push(_thread);
    Console::puts("Scheduler resume called\n");
    enable_interrupts();
}

void Scheduler::add(Thread * _thread) {
    resume(_thread);
    Console::puts("Scheduler add called\n");
}
```

Resume method in scheduler will push _thread parameter to ready queue. Disable and enable interrupts is implemented to support additional handling.
Add method is nothing but resume method.

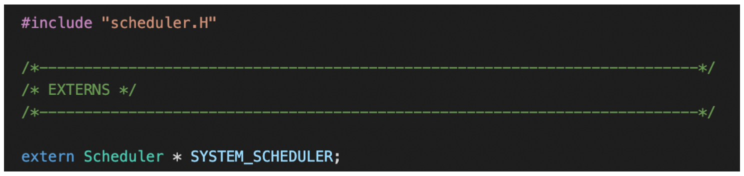


```
void Scheduler::terminate(Thread * _thread) {
    int n = readyQ->size();
    for(int i = 0; i < n; i++) {
        Thread* cur = readyQ->pop();
        if(_thread->ThreadId() != cur->ThreadId()) readyQ->push(cur);
    }
    Console::puts("Scheduler terminate called\n");
}
```

Terminate method will check entire Nodes int the ready queue. If the node’s thread is target thread to terminate, it will not push back again to ready queue, just remove it from the ready queue. Pop method of queue already supports deletion of Node.

Thread

In thread.H

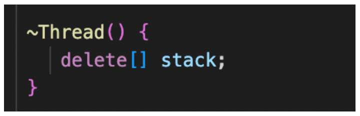


```
#include "scheduler.H"

/*--------------------------------------------------------------------------*/
/* EXTERNS */
/*--------------------------------------------------------------------------*/

extern Scheduler * SYSTEM_SCHEDULER;
```

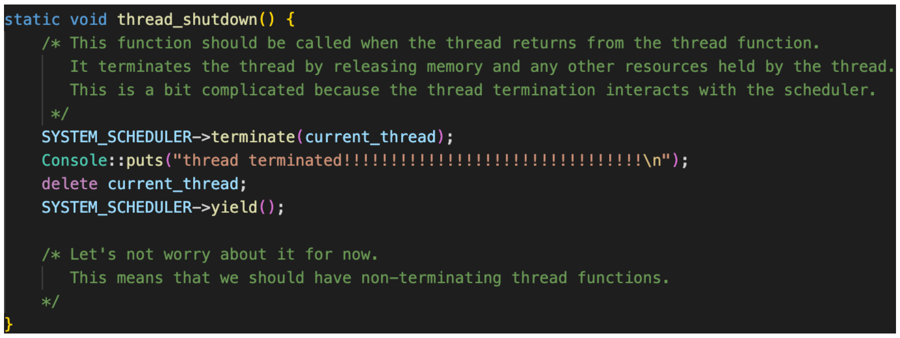
To provide termination of thread in thread class, we need to get extern scheduler variable STYTEM_SCHEDULER from scheduler.H



```
~Thread() {
    delete[] stack;
}
```

Also I added destructor of thread to delete stack of the thread. It will be called when delete of thread called.

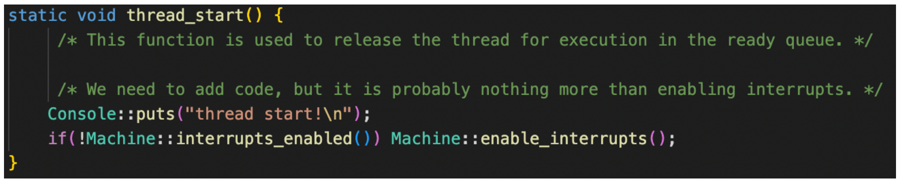
In thread.C



```
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.
     */
    SYSTEM_SCHEDULER->terminate(current_thread);
    Console::puts("thread terminated!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n");
    delete current_thread;
    SYSTEM_SCHEDULER->yield();

    /* Let's not worry about it for now.
       This means that we should have non-terminating thread functions.
    */
}
```

To terminate thread, call terminate method through SYSTEM_SCHEDULER, delete that thread, and make next target thread to run by calling yield through SYSTEM_SCHEDULER.



```
static void thread_start() {
    /* This function is used to release the thread for execution in the ready queue. */

    /* We need to add code, but it is probably nothing more than enabling interrupts. */
    Console::puts("thread start!\n");
    if(!Machine::interrupts_enabled()) Machine::enable_interrupts();
}
```

As written in the annotation, nothing more than enabling interrupts.