

## **Machine Problem 5: Kernel-Level Thread Scheduling**

**Gaurangi Sinha**

**UIN: 734004353**

**CPSC611: Operating System**

### **Assigned Task**

**Main Task:** Completed.

**Bonus Option 1:** Completed.

**Bonus Option 2:** Not Completed.

**Bonus Option 3:** Not Completed.

### **System Design**

This Machine Problem 5 aims to develop a First-In-First-Out (FIFO) scheduler to manage the execution order of multiple kernel-level threads. This involves:

1. Crafting a scheduler class tasked with the management and coordination of thread execution within the system.
2. Construct a queue mechanism that will be responsible for organizing and maintaining the threads in the order they are to be scheduled.

### **Code Description**

I mainly made several changes in the files "scheduler.C", "scheduler.H", and "thread.C". I also made the header file, "threadnode.h".

**Part I: FIFO** Design a scheduler architecture that orchestrates the execution of kernel-level threads. It should also incorporate a queue structure utilizing linked nodes, where each node holds a pointer to a thread object, to manage the sequencing of threads in a FIFO manner.

**Part II: Handling of Interrupts** We address the handling of system interrupts in the context of thread scheduling. Utilizing the provided `Machine::enableInterrupts()` and `Machine::disableInterrupts()` functions to manage interrupt states. This is critical for operations like yielding the processor from the currently active thread and adding threads into the scheduling queue, ensuring smooth thread transitions and system stability.

```
threadnode.h
~/Documents/MP5_Sources/MP5_Sources
Save

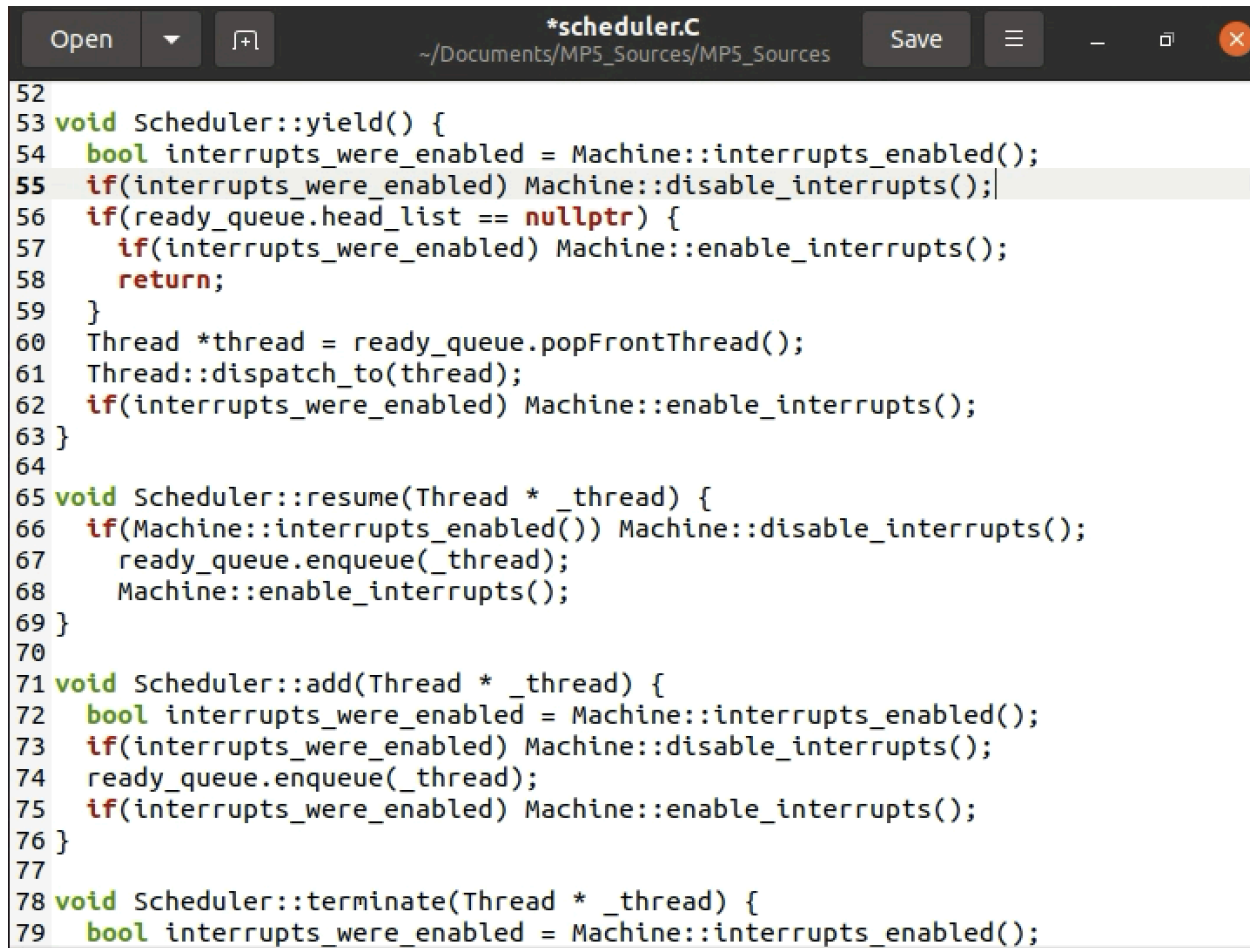
1 // Header inclusion
2 #include "thread.H"
3
4 // Definition of the Thread Container class
5 class Thread;
6 class threadnode {
7 private:
8     Thread* payloadThread;
9     threadnode* nextNode;
10
11 public:
12     // Static members to keep track of the list's head and tail
13     static threadnode* head_list;
14     static threadnode* tail_list;
15
16     // Default constructor
17     threadnode() : payloadThread(nullptr), nextNode(nullptr) {}
18
19     // Constructor with thread parameter
20     threadnode(Thread *thread1){
21         payloadThread = thread1;
22         nextNode = nullptr;
23     }
24
25     // Method to add a thread to the container
26     static void enqueue(Thread *thread1) {
27         threadnode* newNode = new threadnode(thread1);
28         ShowApplications head list) {
```

Figure: class Thread

```
Activities Text Editor Mar 31 22:05
threadnode.h
~/Documents/MP5_Sources/MP5_Sources
Save

54     }
55     delete current;
56     return;
57 }
58 previous = current;
59 current = current->nextNode;
60 }
61 }
62
63 // Check if the container is empty
64 static bool isEmpty(){
65     if(head_list != nullptr) return false;
66     return true;
67 }
68
69 // Get the front thread and remove it from the container
70 static Thread* popFrontThread() {
71     if (isEmpty()) return nullptr;
72
73     threadnode* frontNode = head_list;
74     Thread* frontThread = frontNode->payloadThread;
75     head_list = frontNode->nextNode;
76     if (head_list == nullptr) {
77         tail_list = nullptr;
78     }
79     delete frontNode;
80     return frontThread;
81 }
```

The code above in the header file “threadnode.h” defines a class ‘threadnode’ which serves as a node in a queue for managing threads. This queue is of a FIFO structure, where threads are enqueued at the tail and dequeued from the head. The ‘threadnode’ class has static pointers to the first and last nodes, helping track the queue's bounds. It provides methods for adding threads to the queue (enqueue), removing specific threads (removeThread), checking if the queue is empty (isEmpty), and retrieving and removing the front thread (popFrontThread). The enqueue operation adds a new node to the end of the list, and the dequeue operation removes a node from the front, maintaining the FIFO order.



```
52
53 void Scheduler::yield() {
54     bool interrupts_were_enabled = Machine::interrupts_enabled();
55     if(interrupts_were_enabled) Machine::disable_interrupts();
56     if(ready_queue.head_list == nullptr) {
57         if(interrupts_were_enabled) Machine::enable_interrupts();
58         return;
59     }
60     Thread *thread = ready_queue.popFrontThread();
61     Thread::dispatch_to(thread);
62     if(interrupts_were_enabled) Machine::enable_interrupts();
63 }
64
65 void Scheduler::resume(Thread * _thread) {
66     if(Machine::interrupts_enabled()) Machine::disable_interrupts();
67     ready_queue.enqueue(_thread);
68     Machine::enable_interrupts();
69 }
70
71 void Scheduler::add(Thread * _thread) {
72     bool interrupts_were_enabled = Machine::interrupts_enabled();
73     if(interrupts_were_enabled) Machine::disable_interrupts();
74     ready_queue.enqueue(_thread);
75     if(interrupts_were_enabled) Machine::enable_interrupts();
76 }
77
78 void Scheduler::terminate(Thread * _thread) {
79     bool interrupts_were_enabled = Machine::interrupts_enabled();
```

**Figure: yield, resume, add, and terminate threads in scheduler.**

In the above scheduler.C file, this code defines a ‘Scheduler’ class responsible for managing threads in the system. It maintains a FIFO queue of threads, handling operations to yield the CPU to the next thread, resumes a waiting thread by adding it to the queue, adds new threads to the queue, and terminates threads by removing them from the queue. The scheduler ensures that system interrupts are appropriately disabled before manipulating the queue to maintain thread safety and re-enables them afterward if they were previously enabled.

## Compilation

To compile the code, we run the following commands:

```
make clean  
make
```

To run the simulator we run the following command:

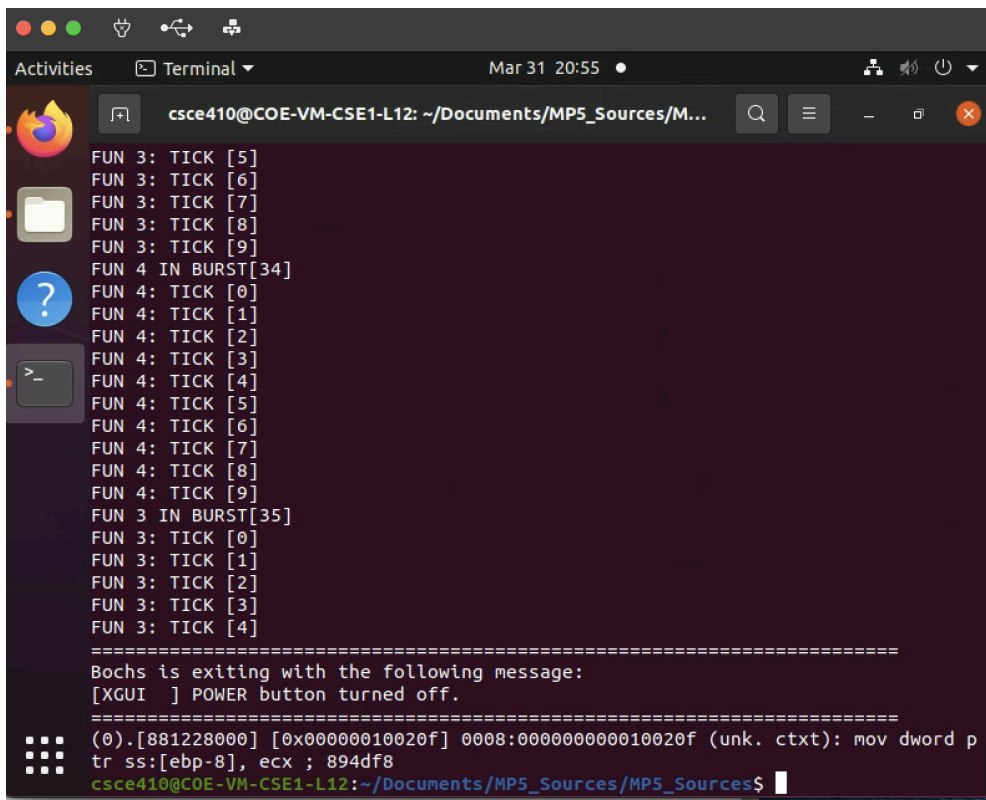
```
./copykernel.sh
```

and then to start the Bochs simulator we run the following command:

```
bochs -f bochsrc.bxrc
```

## Testing

For testing, the machine problem, I ran the above command and, I used the given test function from kernel.C. I ran two cases or scenarios where the `_USES_SCHEDULER_` and `_TERMINATING_FUNCTIONS_` macro is set and unset. All the tests are passing, for given and additional scenarios.



```
Activities  Terminal  Mar 31 20:55  csc410@COE-VM-CSE1-L12: ~/Documents/MP5_Sources/M...  
FUN 3: TICK [5]  
FUN 3: TICK [6]  
FUN 3: TICK [7]  
FUN 3: TICK [8]  
FUN 3: TICK [9]  
FUN 4 IN BURST[34]  
FUN 4: TICK [0]  
FUN 4: TICK [1]  
FUN 4: TICK [2]  
FUN 4: TICK [3]  
FUN 4: TICK [4]  
FUN 4: TICK [5]  
FUN 4: TICK [6]  
FUN 4: TICK [7]  
FUN 4: TICK [8]  
FUN 4: TICK [9]  
FUN 3 IN BURST[35]  
FUN 3: TICK [0]  
FUN 3: TICK [1]  
FUN 3: TICK [2]  
FUN 3: TICK [3]  
FUN 3: TICK [4]  
=====
```

Bochs is exiting with the following message:  
[XGUI ] POWER button turned off.

```
=====
```

(0).[881228000] [0x00000010020f] 0008:00000000010020f (unk. ctxt): mov dword p  
tr ss:[ebp-8], ecx ; 894df8  
csc410@COE-VM-CSE1-L12:~/Documents/MP5\_Sources/MP5\_Sources\$

```
csce410@COE-VM-CSE1-L12: ~/Documents/MP5_Sources/M...
<DOCNS:1> c
Installed exception handler at ISR <0>
Allocating Memory Pool... done
Installed interrupt handler at IRQ <0>
Constructed Scheduler.
Hello World!
CREATING THREAD 1...
esp = <2098108>
done
DONE
CREATING THREAD 2...esp = <2099156>
done
DONE
CREATING THREAD 3...esp = <2100204>
done
DONE
CREATING THREAD 4...esp = <2101252>
done
DONE
STARTING THREAD 1 ...
Thread: 0
FUN 1 INVOKED!
FUN 1 IN BURST[0]
FUN 1: TICK [0]
FUN 1: TICK [1]
FUN 1: TICK [2]
FUN 1: TICK [3]
FUN 1: TICK [4]
FUN 1: TICK [5]
FUN 1: TICK [6]
```

```
Bochs x86-64 emulator, http://bochs.sourceforge.net/
FUN 1: TICK [8]
FUN 1: TICK [9]
FUN 2 IN BURST[125]
FUN 2: TICK [0]
FUN 2: TICK [1]
FUN 2: TICK [2]
FUN 2: TICK [3]
FUN 2: TICK [4]
FUN 2: TICK [5]
FUN 2: TICK [5]
FUN 2: TICK [6]
FUN 2: TICK [7]
FUN 2: TICK [8]
FUN 2: TICK [9]
FUN 3 IN BURST[125]
FUN 3: TICK [0]
FUN 3: TICK [1]
FUN 3: TICK [2]
FUN 3: TICK [3]
FUN 3: TICK [4]
FUN 3: TICK [5]
FUN 3: TICK [6]
FUN 3: TICK [7]
FUN 3: TICK [8]
FUN 3: TICK [9]
FUN 4: TICK [1]
FUN 4: TICK [2]
FUN 4: TICK [3]
FUN 4: TICK [4]
```

