# MP5: Simple Kernel Threads

Ashutosh Punyani UIN: 834006613 CSCE611: Operating System

### **Assigned Tasks**

Main: Completed.

Bonus Option 1: Completed. Bonus Option 2: Completed.

### System Design

The main objective of machine problem 5 is the implementation of a basic FIFO scheduler for simple kernel level threads. For bonus option 1, we need to enable the interrupts while we start the thread. For the bonus option 2, we need to implement the round robin scheduling which generates a timer based interrupt every 50 ms to make the running thread yield.

### Code Description

During the implementation of this machine problem, I made changes to the following six files:

- 1. scheduler.H
- 2. scheduler.C
- 3. simple\_timer.H
- 4. simple\_timer.C
- 5. thread.C
- 6. kernel.C
- 1. scheduler.H

In scheduler.H, I have created a new data structure Thread\_List for maintaining the the queue of threads. I have also created separate class for FIFOScheduler and RRScheduler which are derived from Scheduler. In RRScheduler, in addition to base class variables the I have added quantum\_passed and quantum\_manager() for managing the quantum timer in case of round robin scheduling.

```
Thread *thread;
    Thread_List *head;
Thread_List *current;
  // constructor
FIF0Scheduler();
 You,/minutesago|lauthor(You)
class RRScheduler : public Scheduler
private:
   bool quantum passed;
      // constructor
RRScheduler();
             id yied();
Called by the currently running thread in order to give up the CPU.
The scheduler selects the next thread from the ready queue to load onto
the CPU, and calls the dispatcher function defined in 'Thread.H' to
do the context switch. */
```

Figure 1: scheduler.H

#### 2. scheduler.C

(a) Scheduler Class: All the functions that are part of this class are used to create a derived class from this class. The constructor of this class is used to initialize the scheduler by setting the head and current members to NULL, while all the other functions console log what they are intended to do.

```
Scheduler::Scheduler()
{
    // assert(false);
    head = NULL;
    current = NULL;
    Console::puts("Constructed Scheduler::Scheduler().\n");
}

void Scheduler::yield()
{
    // assert(false);
    Console::puts("Scheduler::yield().\n");
}

void Scheduler::resume(Thread *_thread)
{
    // assert(false);
    Console::puts("Scheduler::resume().\n");
}

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

void Scheduler::terminate(Thread *_thread)
{
    Console::puts("Scheduler::terminate().\n");
}
```

Figure 2: Scheduler Class

(b) FIFOScheduler::FIFOScheduler( (Class Constructor) : The constructor is used to construct the scheduler object. It initializes the head and current members to NULL.

```
FIFOScheduler::FIFOScheduler()
{
    Console::puts("Constructed FIFOScheduler::FIFOScheduler() - start.\n");
    head = NULL;
    current = NULL;
    Console::puts("Constructed FIFOScheduler::FIFOScheduler() - end.\n");
}
```

Figure 3: FIFOScheduler::FIFOScheduler( (Class Constructor)

(c) **FIFOScheduler::yield()**: This function is responsible for yielding the threads in the ready queue. It switches to the next thread in a FIFO (First-In-First-Out) manner. This function temporarily disables interrupts, updates the thread list, dispatches control to the next thread, releases memory, and re-enables interrupts if they were previously disabled. It also logs messages for debugging.

```
FIF0Scheduler::FIF0Scheduler()
{
   Console::puts("Constructed FIF0Scheduler::FIF0Scheduler() - start.\n");
   head = NULL;
   current = NULL;
   Console::puts("Constructed FIF0Scheduler::FIF0Scheduler() - end.\n");
}
```

Figure 4: **FIFOScheduler::yield()** 

(d) **FIFOScheduler::add(Thread \*\_thread)**: This function adds a new thread to the scheduler's ready queue in a FIFO manner. This function temporarily disables interrupts, updates the queue, and logs messages for debugging purposes.

Figure 5: FIFOScheduler::resume(Thread \*\_thread)

(e) **FIFOScheduler::resume**: This function places a thread at the back of the waiting line. It's typically used when a paused thread becomes active due to a specific event or when the current running thread has to yield the CPU.

```
void FIFOScheduler::resume(Thread *_thread)
{
    // assert(false);
    Console::puts("FIFOScheduler::resume() - start.\n");
    add(_thread);
    Console::puts("Thread Resume:");
    Console::putui(_thread->ThreadId() + 1);
    Console::puts("\n");
    Console::puts("FIFOScheduler::resume() - end.\n");
}
```

Figure 6: **FIFOScheduler::resume(Thread \*\_thread)** 

(f) FIFOScheduler::terminate(Thread \*\_thread): This function temporarily disables interrupts. If that thread is the current running thread, the scheduler just need to execute yield() and releases allocated memory resources upon thread termination. If that thread is the not the current running thread, \_thread is searched in the ready queue using its threadID by traversing through the list Thread\_List objects and once a match is found it is removed from the ready queue, and releases allocated memory resources upon thread termination. Otherwise we don't do any thing. This function is crucial for managing thread lifecycles in a First-In-First-Out scheduling context.

```
// assert(false);
if (Machine::interrupts_enabled())
   Machine::disable_interrupts();
f
Console::puts("FIFOScheduler::terminate() - start.\n");
if (Thread::CurrentThread() == _thread)
   vield();
else if (head->thread == thread)
   head->prev = NULL;
   Thread_List *itr = head;
for (itr = head; itr->next->thread != _thread; itr = itr->next)
      Thread_List *n_temp = itr->next;
     Thread List *p_temp = itr->prev;
itr->next = n_temp->next;
itr->prev = p_temp->prev;
      MEMORY_POOL->release((unsigned long)n_temp);
MEMORY_POOL->release((unsigned long)p_temp);
 else if (head->thread == thread)
   head->prev = NULL;
   Thread List *itr = head;
    for (itr = head; itr->next->thread != _thread; itr = itr->next)
      Thread_List *n_temp = itr->next;
      Thread List *p_temp = itr->prev;
itr->next = n_temp->next;
itr->prev = p_temp->prev;
      MEMORY_POOL->release((unsigned long)n_temp);
MEMORY_POOL->release((unsigned long)p_temp);
Console::puts("Thread Terminated : ");
Console::puti{ thread->ThreadId() + 1);
Console::puts("\n");
Console::puts("TiPOScheduler::terminate() - end.\n");
if (!Machine::interrupts_enabled())
   Console::puts("Interrupts Enabled.\n");
Machine::enable interrupts();
```

Figure 7: FIFOScheduler::terminate(Thread \*\_thread)

(g) RRScheduler::RRScheduler()(Class Constructor): The constructor initializes an instance of the Round-Robin (RR) scheduler. It sets the quantum\_passed to false. quantum\_passed is used to differentiate between a voluntary yield action and a passive yield action.

```
RRScheduler::RRScheduler()
{
   Console::puts("Constructed RRScheduler::RRScheduler() - start.\n");
   quantum_passed = false;
   Console::puts("Constructed RRScheduler::RRScheduler() - end.\n");
}
```

Figure 8: RRScheduler::RRScheduler()(Class Constructor)

(h) RRScheduler::yield(): When the currently executing thread voluntarily calls yield(), this function behaves in the same way as the FIFOScheduler::yield() function. However, when this function is called by RRScheduler::quantumhandler(), it must send a signal to instruct the EOQ handler to return first, after which it can perform the context switch.

```
void RRScheduler::yield()
 Console::puts("RRScheduler::yield() - start.\n");
 if (quantum passed)
   quantum passed = false;
 Thread List *temp = head;
 if (head == NULL)
   Console::puts("Empty Ready Queue\n");
 if (head->next == NULL)
   Console::puts("Before last Thread\n");
 head = head->next;
 head->prev = NULL;
 Console::puts("Thread Dispatched to : ");
 Console::puti(temp->thread->ThreadId() + 1);
 Console::puts("\n");
 Thread::dispatch to(temp->thread);
 MEMORY POOL->release((unsigned long)temp);
 Console::puts("RRScheduler::yield() - end.\n");
```

Figure 9: **RRScheduler::yield()** 

(i) RRScheduler::add(Thread \*\_thread): This function works similar to FIFOScheduler::add(Thread \*\_thread)

Figure 10: RRScheduler::add(Thread \*\_thread)

(j) RRScheduler::resume(Thread \*\_thread): This function works similar to FIFOScheduler::resume(Thread

```
void RRScheduler::resume(Thread *_thread)
{
   Console::puts("RRScheduler::resume() - start.\n");
   add(_thread);
   Console::puts("Thread Resume:");
   Console::putui(_thread->ThreadId() + 1);
   Console::puts("\n");
   Console::puts("RRScheduler::resume() - end.\n");
}
```

Figure 11: RRScheduler::resume(Thread \*\_thread)

(k) RRScheduler::terminate(Thread \*\_thread): This function works similar to FIFOScheduler::terminate(

Figure 12: RRScheduler::terminate(Thread \*\_thread)

(l) RRScheduler::quantum\_manager(): This function is invoked by EOQTimer::handle\_interrupt(REGS \*\_r when the current running thread has exhausted its allocated time quantum. Its purpose is to trigger a forced preemption of the currently executing thread as the current running thread has exhausted its allocated time quantum.

```
void RRScheduler::quantum_manager()
{
   Console::puts("RRScheduler::quantum_manager() - start.\n");
   Console::puts("One Quantum is over.\n");
   quantum_passed = true;
   /* Send an EOI message to the master interrupt controller. */
   Machine::outportb(0x20, 0x20);
   resume(Thread::CurrentThread());
   yield();
   Console::puts("\n");
   Console::puts("\n");
   Console::puts("RRScheduler::quantum_manager() - end.\n");
}
```

Figure 13: RRScheduler::quantum\_manager()

#### 3. simple\_timer.H

In simple\_timer.H, I have created the interface for EOQTimer derived from SimpleTimer. In addition to constructor and handle\_interrupt((REGS \*\_r)), it also has two functions reset\_timer\_counter() to reset the tick counter and get\_timer\_counter() to get the current number of ticks.

```
You, 3 hours ago | 1 author (You) class EOQTimer : public SimpleTimer

public: You, 3 hours ago * formatted

EOQTimer(int_hz);

/* Initialize the end of quantum timer, and set its frequency. */

void handle_interrupt(REGS *_r);

/* This must be installed as the interrupt handler for the timer

| when the system gets initialized. (e.g. in "kernel.C")

*/

void reset_timer_counter();

/*

| reset the timer when thread is yielded or time quantum is over

*/

int get_timer_counter();

/*

get the timer counter

*/

get the timer counter
```

Figure 14: simple\_timer.C

#### 4. simple\_timer.C

In the constructor, "EOQTimer::EOQTimer(int \_hz)", I initialize the timer with a given frequency (\_hz). When handling timer interrupts in the "EOQTimer::handle\_interrupt(REGS \*\_r)" method, I update a tick counter and trigger the quantum manager when a specific time has passed. Additionally, I've provided methods like "EOQTimer::reset\_timer\_counter()" to reset the timer counter and "EOQTimer::get\_timer\_counter()" to retrieve the current timer counter value

Figure 15:  $simple\_timer.C$ 

#### 5. thread.C

When a thread's function completes, it calls 'thread\_shutdown()', which then triggers Scheduler's terminate() function to determine the next action, releasing the thread's memory, including its stack. In contrast, the 'thread\_start()' function solely enables interrupts through the 'enable\_interrupts' function declared in 'Machine'.

```
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);
Console::puts("Thread Shutdown - Start.\n");
Thread *temp = Thread::CurrentThread():
Console::puts("Thread Shutdown - Start.\n");
SYSTEM SCHEDULER.>terminate(temp);
MEMORY POOL->release((unsigned long)temp);
SYSTEM SCHEDULER.>terminate(temp);
SYSTEM SCHEDULER.>terminate(temp);

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

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

Figure 16: thread.C

#### 6. kernel.C

I have defined macros \_FIFO\_SCHEDULER\_ to execute in scheduler in FIFO manner and \_RR\_SCHEDULER\_ to execute in scheduler in Round Robin manner. Based on which macro is uncommented the timers are being initialized SimpleTimer for FIFO while EOQTimer for Round Robin scheduler and also respective object type is being initialized.

```
#define _FIF0_SCHEDULER_
// #define RR SCHEDULER
/* -- A POINTER TO THE SYSTEM SCHEDULER */
Scheduler *SYSTEM SCHEDULER;
#ifdef USES SCHEDULER
#ifdef FIFO SCHEDULER
      SYSTEM SCHEDULER = new FIFOScheduler();
#endif
#ifdef RR SCHEDULER
#endif
```

Figure 17: kernel.C

## Testing

During the development of the code, I wrote several Console:puts() and Console:putui() statements to pinpoint where my code was breaking and to understand if the logic was incorrect or not performing as expected. I removed some Console statements.

For terminating threads, such as thread 1 and 2, they are terminated, and context switching happens every 50 ms between thread 3 and 4.

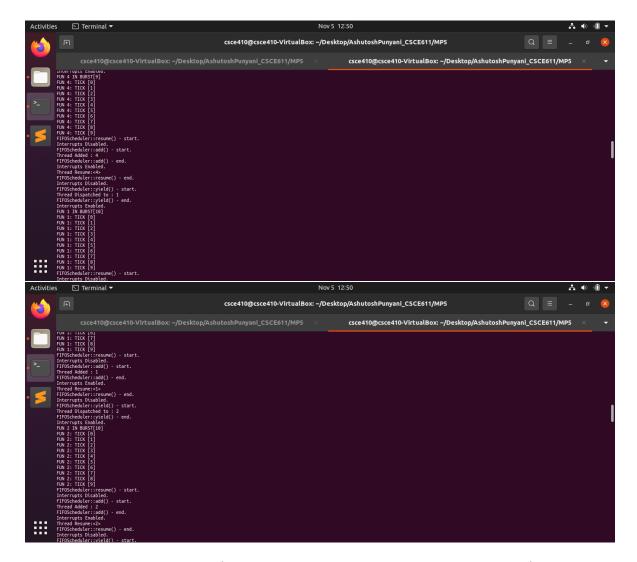


Figure 18: Testing (with and\_FIFO\_SCHEDULER\_ uncommented)

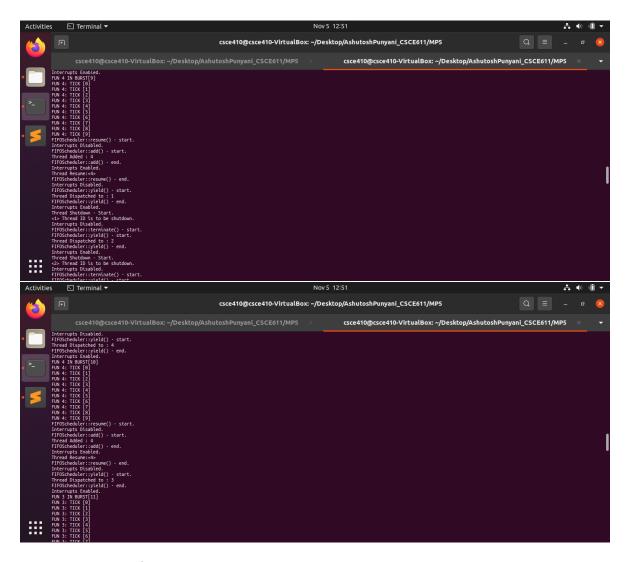


Figure 19: Testing (with  $\_$ TERMINATING\_FUNCTIONS $\_$  and  $\_$ FIFO $\_$ SCHEDULER $\_$  uncommented)

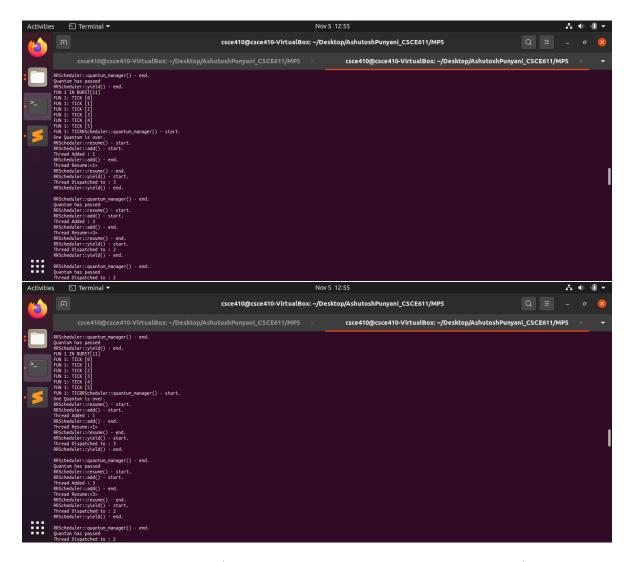


Figure 20: Testing (with and  $\_RR\_SCHEDULER\_$  uncommented)

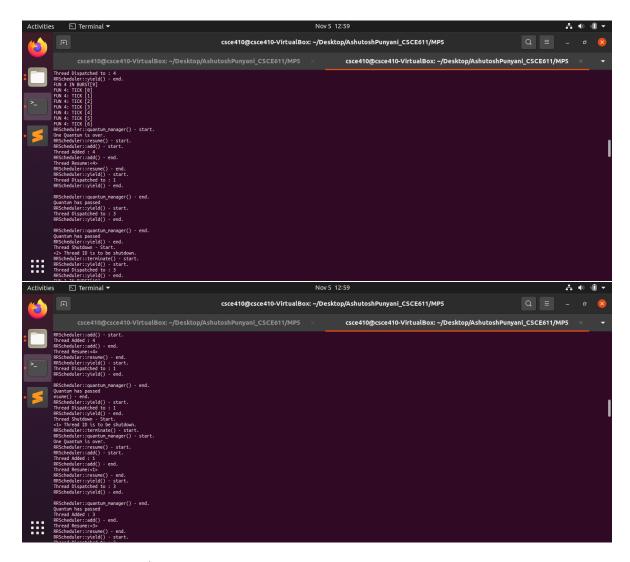


Figure 21: Testing (with \_TERMINATING\_FUNCTIONS\_ and \_RR\_SCHEDULER\_ uncommented)