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.

```
/*--------------------------------------------------------------------------*/
/* DATA STRUCTURE */
/*--------------------------------------------------------------------------*/

/*--------------------------------------------------------------------------*/
/* THREAD LIST */
/*--------------------------------------------------------------------------*/

You, 7 minutes ago | 1 author (You)
class Thread_List
{
public:
   Thread *thread;
   Thread_List *next;
   Thread_List *prev;
};
```

```
You, 7 minutes ago | 1 author (You)
class Scheduler
{

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

protected:
   Thread_List *head;
   Thread_List *current;
```

```
You, 7 minutes ago | 1 author (You)
class FIFOScheduler : public Scheduler
{
public:
   // constructor
   FIFOScheduler();

   virtual void yield();
   /* 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. */

   virtual void resume(Thread *_thread);
   /* Add the given thread to the ready queue of the scheduler. This is called
      for threads that were waiting for an event to happen, or that have
      to give up the CPU in response to a preemption. */

   virtual void add(Thread *_thread);
   /* Make the given thread runnable by the scheduler. This function is called
      after thread creation. Depending on implementation, this function may
      just add the thread to the ready queue, using 'resume'. */

   virtual void terminate(Thread *_thread);
   /* Remove the given thread from the scheduler in preparation for destruction
      of the thread.
      Graciously handle the case where the thread wants to terminate itself.*/
};
```

```
You, 7 minutes ago | 1 author (You)
class RRScheduler : public Scheduler
{

private:
   bool quantum_passed;

public:
   // constructor
   RRScheduler();

   void yield();
   /* 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. */

   void resume(Thread *_thread);
   /* Add the given thread to the ready queue of the scheduler. This is called
      for threads that were waiting for an event to happen, or that have
      to give up the CPU in response to a preemption. */

   void add(Thread *_thread);
   /* Make the given thread runnable by the scheduler. This function is called
      after thread creation. Depending on implementation, this function may
      just add the thread to the ready queue, using 'resume'. */

   void terminate(Thread *_thread);
   /* Remove the given thread from the scheduler in preparation for destruction
      of the thread.
      Graciously handle the case where the thread wants to terminate itself.*/

   void quantum_manager();
   /*
   Manages whenever quantum has been passed.
   */
};
```

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.

```cpp
void FIFOScheduler::yield()
{
  // assert(false);
  if (Machine::interrupts_enabled())
  {
    Console::puts("Interrupts Disabled.\n");
    Machine::disable_interrupts();
  }
  Console::puts("FIFOScheduler::yield() - start.\n");
  Thread_List *temp = head;
  if (head == NULL)
  {
    Console::puts("Empty Ready Queue\n");
    assert(false);
  }
  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("FIFOScheduler::yield() - end.\n");
  if (!Machine::interrupts_enabled())
  {
    Console::puts("Interrupts Enabled.\n");
    Machine::enable_interrupts();
  }
}
```

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.

```cpp
void FIFOScheduler::add(Thread *_thread)
{
  // assert(false);
  if (Machine::interrupts_enabled())
  {
    Console::puts("Interrupts Disabled.\n");
    Machine::disable_interrupts();
  }
  Console::puts("FIFOScheduler::add() - start.\n");
  Thread_List *new_thread = (Thread_List *)(MEMORY_POOL->allocate(sizeof(Thread_List)));
  new_thread->thread = _thread;
  new_thread->next = NULL;
  new_thread->prev = NULL;
  if (head == NULL && current == NULL)
  {
    head = new_thread;
    current = new_thread;
  }
  else
  {
    current->next = new_thread;
    new_thread->prev = current;
    current = new_thread;
  }
  Console::puts("Thread Added : ");
  Console::puti(_thread->ThreadId() + 1);
  Console::puts("\n");
  Console::puts("FIFOScheduler::add() - end.\n");
  if (!Machine::interrupts_enabled())
  {
    Console::puts("Interrupts Enabled.\n");
    Machine::enable_interrupts();
  }
}
```

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.

```cpp
void FIFOScheduler::terminate(Thread *_thread)
{
  // assert(false);
  if (Machine::interrupts_enabled())
  {
    Console::puts("Interrupts Disabled.\n");
    Machine::disable_interrupts();
  }
  Console::puts("FIFOScheduler::terminate() - start.\n");
  if (Thread::CurrentThread() == _thread)
  {
    yield();
  }
  else if (head->thread == _thread)
  {
    head = head->next;
    head->prev = NULL;
  }
  else
  {
    Thread_List *itr = head;
    for (itr = head; itr->next->thread != _thread; itr = itr->next)
    {
    }

    if (itr != NULL)
    {

      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("FIFOScheduler::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");
    assert(false);
  }
  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 *_thr`

```
void RRScheduler::add(Thread *_thread)
{
  Console::puts("RRScheduler::add() - start.\n");
  Thread_List *new_thread = (Thread_List *)(MEMORY_POOL->allocate(sizeof(Thread_List)));
  new_thread->thread = _thread;
  new_thread->next = NULL;
  new_thread->prev = NULL;
  if (head == NULL && current == NULL)
  {
    head = new_thread;
    current = new_thread;
  }
  else
  {
    current->next = new_thread;
    new_thread->prev = current;
    current = new_thread;
  }
  Console::puts("Thread Added : ");
  Console::puti(_thread->ThreadId() + 1);
  Console::puts("\n");
  Console::puts("RRScheduler::add() - end.\n");
}
```

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

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

```
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(`

```
void RRScheduler::terminate(Thread *_thread)
{
  Console::puts("RRScheduler::terminate() - start.\n");
  if (Thread::CurrentThread() == _thread)
  {
    yield();
  }
  else if (head->thread == _thread)
  {
    head = head->next;
    head->prev = NULL;
  }
  else
  {
    Thread_List *itr = head;
    for (itr = head; itr->next->thread != _thread; itr = itr->next)
    {
    }

    if (itr != NULL)
    {

      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("RRScheduler::terminate() - end.\n");
}
```

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("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
  */
};
```

Figure 14: **simple_timer.C**

4. **simple_timer.C**

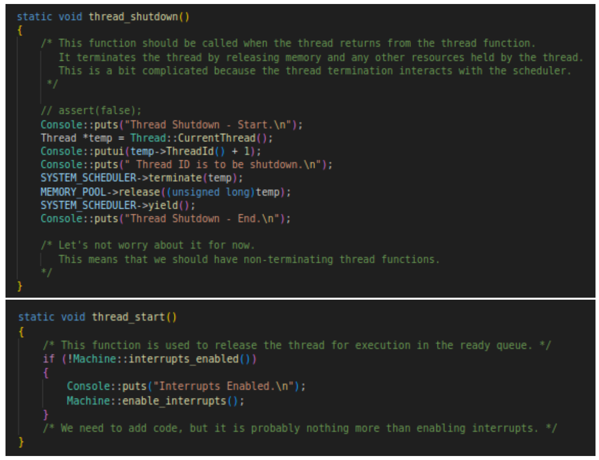
In the constructor, "`EOQTimer::EOQTimer(int _hz)`", I initialize the timer with a given frequency (_hz). When handling timer interrupts in the "`EOQTimer::handle_interrupt(REGS *_r)`" method, I update a tick counter and trigger the quantum manager when a specific time has passed. Additionally, I've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

```
void EOQTimer::handle_interrupt(REGS *_r)
{
  /* What to do when timer interrupt occurs? In this case, we update "ticks",
     and maybe update "seconds".
     This must be installed as the interrupt handler for the timer in the
     when the system gets initialized. (e.g. in "kernel.C") */

  /* Increment our "ticks" count */
  ticks++;

  /*      You, 3 hours ago • formatted
  qunatum is 50 ms
  1 s =1000 ms
  which  means 50 ms = hz/20
  Whenever a second is over, we update counter accordingly.
  */
  if (ticks >= hz / 20)
  {
    ((RRScheduler *)SYSTEM_SCHEDULER)->quantum_manager();
    reset_timer_counter();
    Console::puts("Quantum has passed\n");
  }
}
void EOQTimer::reset_timer_counter()
{
  /*
    reset the timer when thread is yielded or time quantum is over
  */
  ticks = 0;
}

int EOQTimer::get_timer_counter()
{

  /*
    get the timer counter
  */
  return ticks;
}
```

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::putui(temp->ThreadId() + 1);
    Console::puts(" Thread ID is to be shutdown.\n");
    SYSTEM_SCHEDULER->terminate(temp);
    MEMORY_POOL->release((unsigned long)temp);
    SYSTEM_SCHEDULER->yield();
    Console::puts("Thread Shutdown - End.\n");

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

```
static void thread_start()
{
    /* 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 initailized.

```
/* -- COMMENT/UNCOMMENT THE FOLLOWING LINE TO EXCLUDE/INCLUDE SCHEDULER CODE */

#define _USES_SCHEDULER_
/* This macro is defined when we want to force the code below to use
   a scheduler.
   Otherwise, no scheduler is used, and the threads pass control to each
   other in a co-routine fashion.
*/

/* -- UNCOMMENT THE FOLLOWING LINE TO MAKE THREADS TERMINATING */

#define _TERMINATING_FUNCTIONS_
/* This macro is defined when we want the thread functions to return, and so
   terminate their thread.
   Otherwise, the thread functions don't return, and the threads run forever.
*/

#define _FIFO_SCHEDULER_
// #define _RR_SCHEDULER_

/*--------------------------------------------------------------------------*/
/* INCLUDES */
/*--------------------------------------------------------------------------*/
```

```
/*--------------------------------------------------------------------------*/
/* SCHEDULRE and AUXILIARY HAND-OFF FUNCTION FROM CURRENT THREAD TO NEXT */
/*--------------------------------------------------------------------------*/

#ifdef _USES_SCHEDULER_

/* -- A POINTER TO THE SYSTEM SCHEDULER */
Scheduler *SYSTEM_SCHEDULER;

#endif

#ifdef _RR_SCHEDULER_

EOQTimer *timer;

#endif

#ifdef _FIFO_SCHEDULER_

SimpleTimer *timer;

#endif
```

```
    /* -- MEMORY ALLOCATOR IS INITIALIZED. WE CAN USE new/delete! --*/

    /* -- INITIALIZE THE TIMER (we use a very simple timer).-- */

    /* Question: Why do we want a timer? We have it to make sure that
                 we enable interrupts correctly. If we forget to do it,
                 the timer "dies". */
#ifdef _FIFO_SCHEDULER_
    timer = new SimpleTimer(100); /* timer ticks every 10ms. */
#endif

#ifdef _RR_SCHEDULER_
    timer = new EOQTimer(100); /* timer ticks every 10ms. */
#endif

    InterruptHandler::register_handler(0, timer);
    /* The Timer is implemented as an interrupt handler. */
```

```
#ifdef _USES_SCHEDULER_

/* -- SCHEDULER -- IF YOU HAVE ONE -- */

// SYSTEM_SCHEDULER = new Scheduler();
#ifdef _FIFO_SCHEDULER_
    SYSTEM_SCHEDULER = new FIFOScheduler();
#endif

#ifdef _RR_SCHEDULER_
    SYSTEM_SCHEDULER = new RRScheduler();
#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.

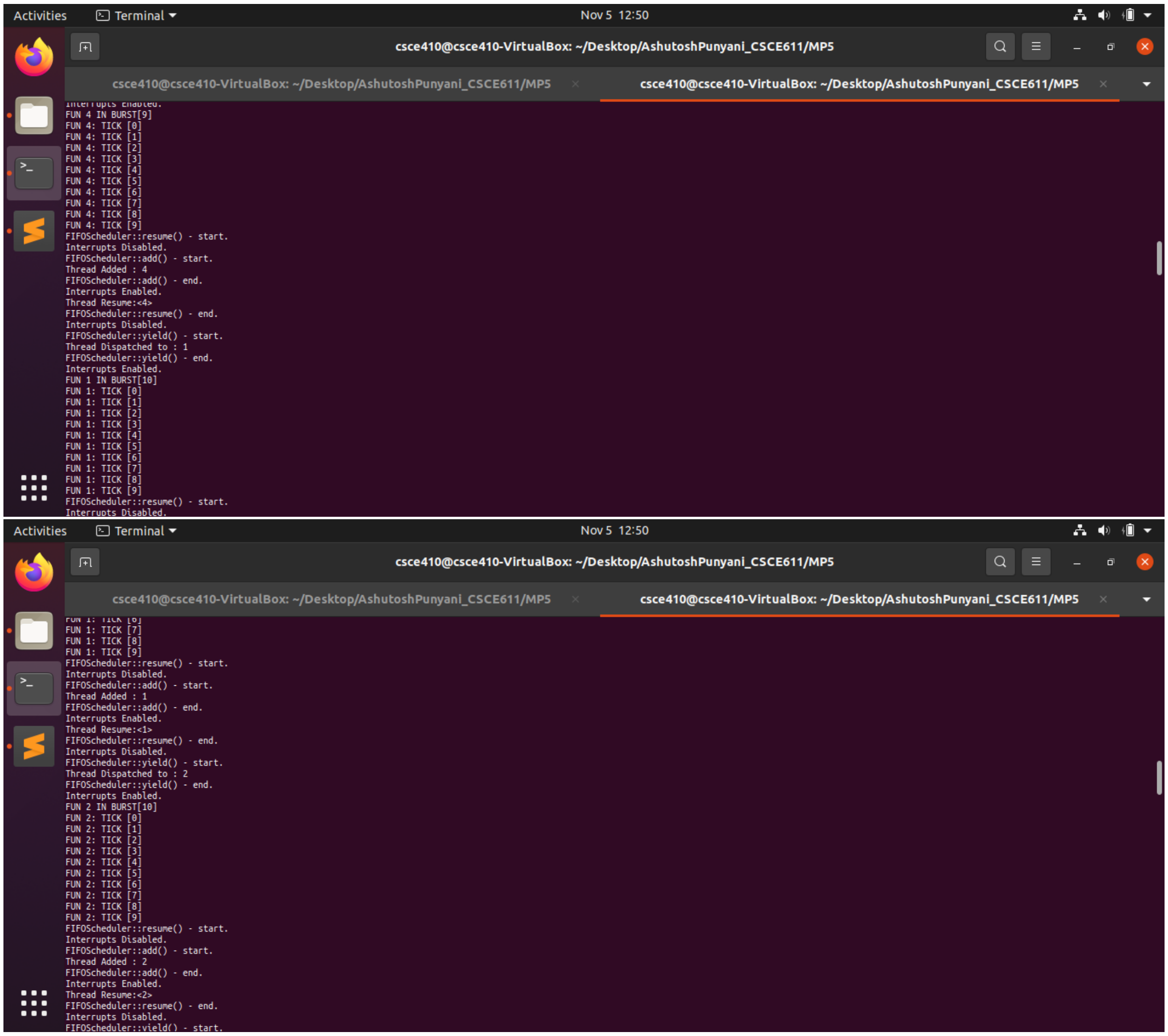


Figure 18: **Testing (with and_FIFO_SCHEDULER_ uncommented)**

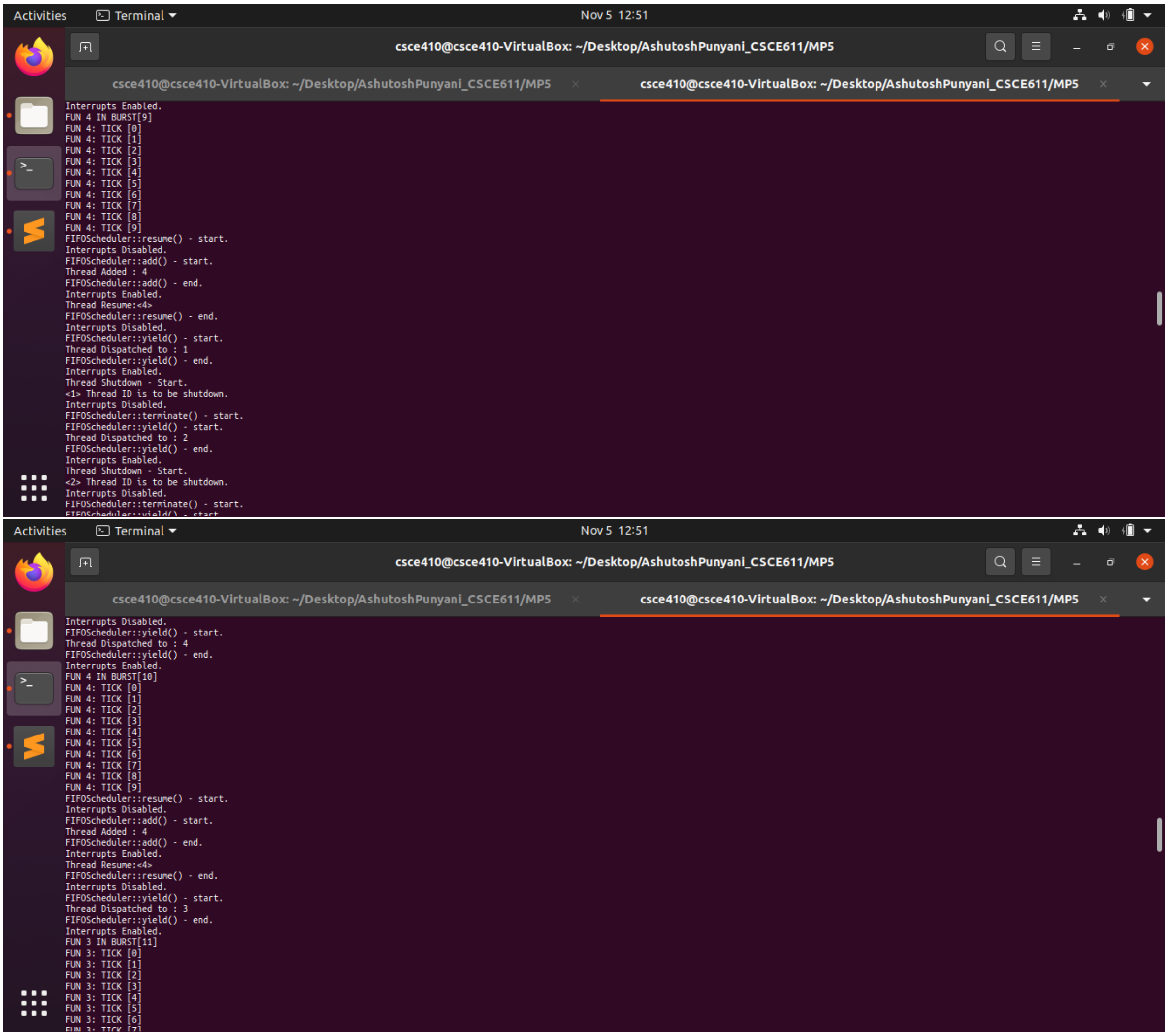


Figure 19: **Testing (with _TERMINATING_FUNCTIONS_ and _FIFO_SCHEDULER_ uncommented)**

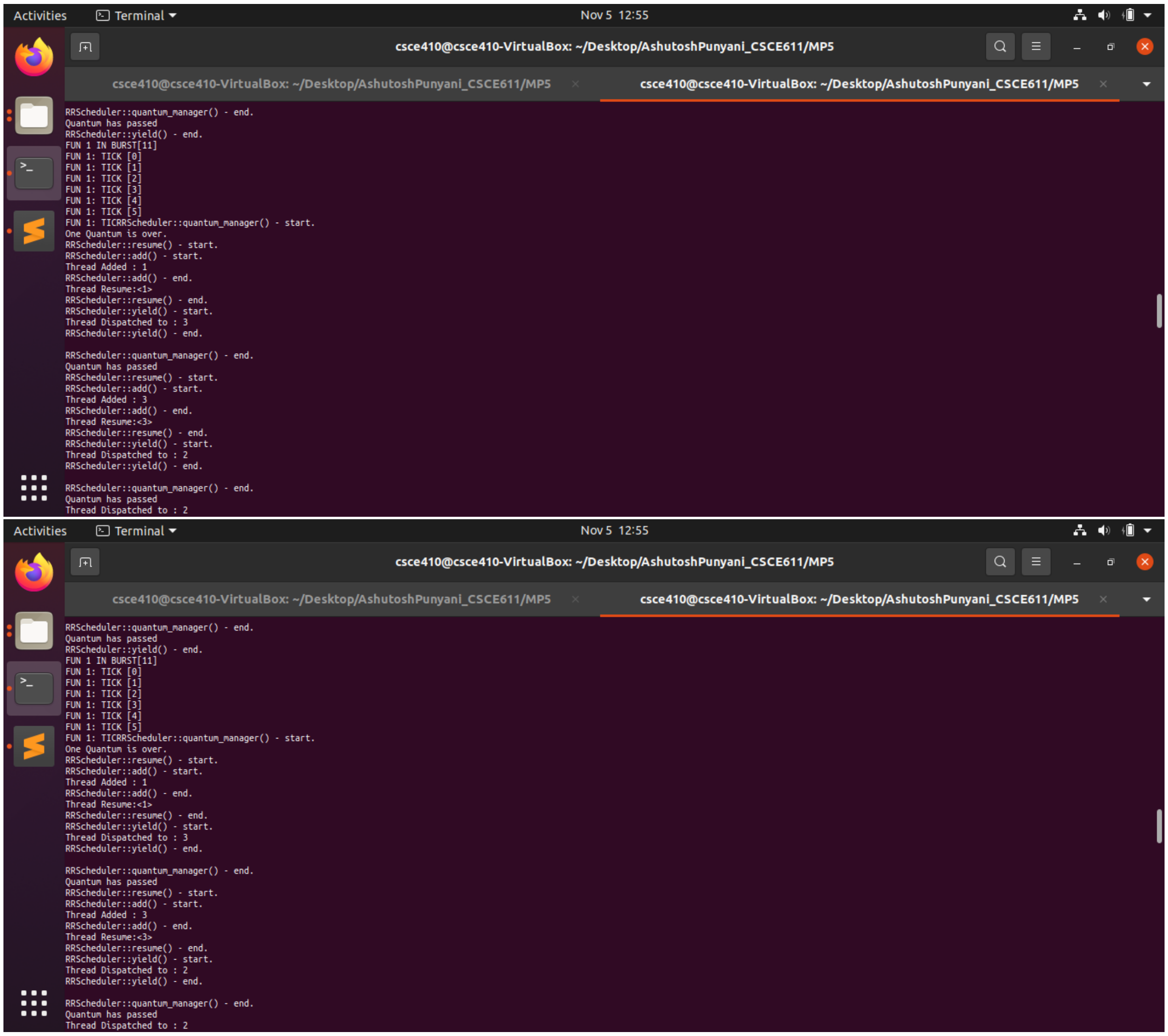


Figure 20: **Testing (with and _RR_SCHEDULER_ uncommented)**

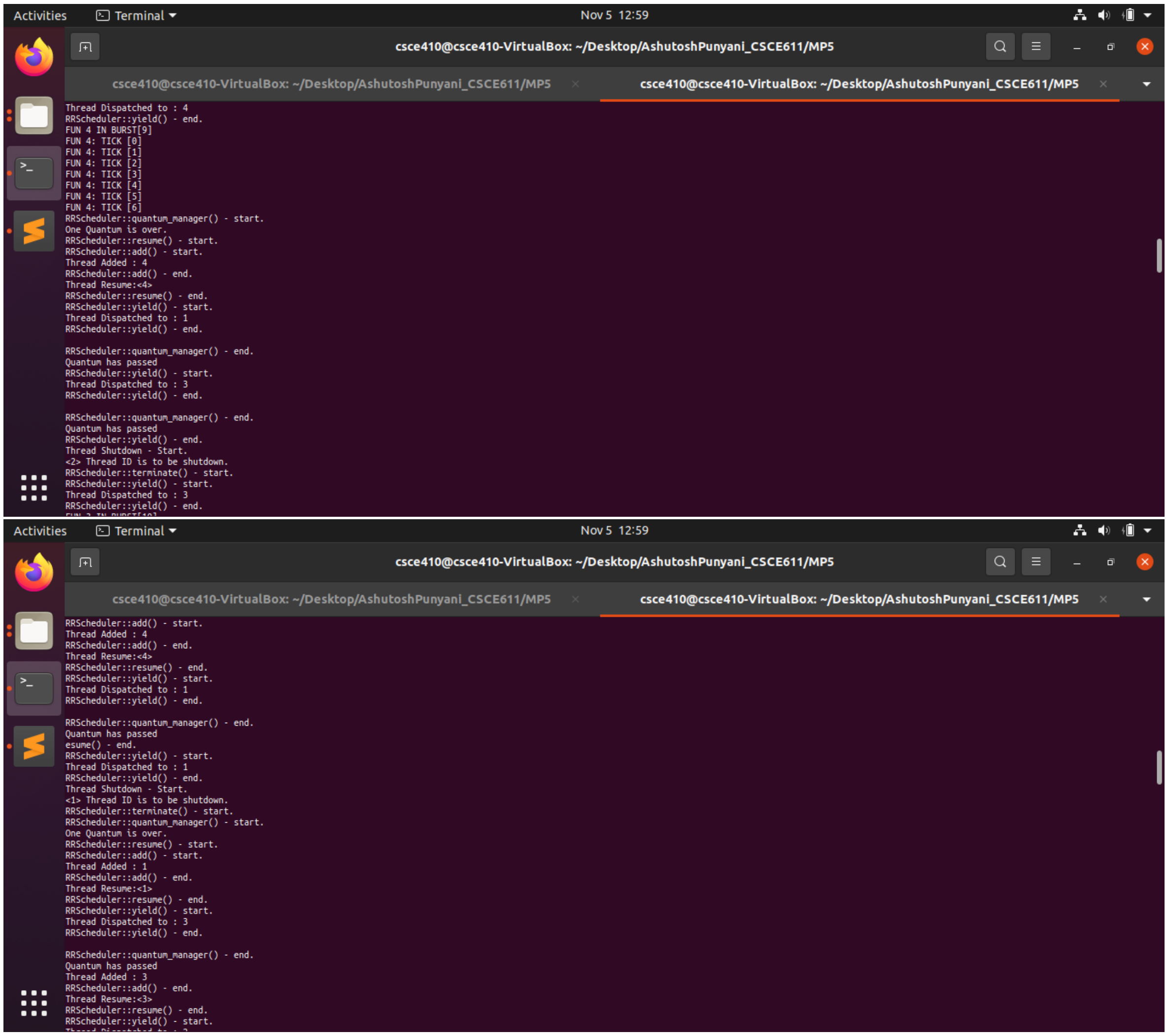


Figure 21: **Testing (with _TERMINATING_FUNCTIONS_ and _RR_SCHEDULER_ uncommented)**