Machine Problem 5 : Kernel Level Thread Scheduling

I have attempted the main problem of FIFO Scheduling, Thread termination and the following bonus tasks:

1. Interrupt handling
2. Round Robin Scheduling

Following files have been modified in the Machine problem:
Descriptive comments have been added in the code and following is a brief explanation:

---

1. Scheduler.H

i. Here I first declared the queue members currThread that points to the thread current being referred and the queue size.
ii. We then have constructors wot first initialize the queue members to NULL and then setting them to add elements.
iii. Enqueue() : this function basically enqueues a new thread to the queue at its end.
iv. Dequeue(): this functions removes the first thread from the queue. This is intended to get the FIFO ordered threads to either yield or analyze.
v. Further we declare the ready queue for our FIFO scheduler.
vi. Class RoundRobinScheduler : public Scheduler, public InterruptHandler: this inherits from the scheduler but also has members to execute the end of quantum timer and handles the eoq interrupts.

```
class Queue
{
   private:
   // Queue will have two members - the current thread pointer and the pointer to the next thread in teh queue
   Thread *currThread;
   Queue *next;
   // we now declare the queue methods- constructors- in order to initialize and then add the nodes to queue
   public:
   Queue()
   {
      currThread = NULL;
      next = NULL;
   }
   Queue(Thread * th)
   {
      currThread = th;
      next = NULL;
   }
```

```
// enqueue defined to add a thread to the ready queue
void enqueue(Thread * add_th)
{
   if(currThread == NULL)
      currThread = add_th; // if no node in queue then add this as the first node of the queue
   else{
      if(next == NULL) // if we reached the end thenn add this node/thread
         next = new Queue(add_th);
      else
         next->enqueue(add_th); // call enqueue till you get the end of the queue
   }
}
// dequeue defined to remove threads from the queue to yeild or to terminate
Thread* dequeue()
{
   if(currThread == NULL) // if the queue is empty
      return NULL;

   if (next)
      {
         Thread* th_to_del = currThread; // getting the current thread
         currThread = next->currThread; // moving the pounter to next thread as the current thread is to be dequeued
         Queue* used = next;
         next = next->next; // traverse
         delete used; // delete the dequeued thread
         return th_to_del;  // return the dequeued thread - now this is deleted from ready queue but itn will be passed to yield and it gets CPU next in that case
      }

      Thread* t = currThread;
      currThread = NULL;
      return t;
}

};
```

```
// creating RR scheduler which inherits from the scheduler and interruptn handler
class RoundRobinScheduler : public Scheduler, public InterruptHandler
{
   // new queue for RR
   Queue RoundRobinQ;
   int RRQ_size; // size
   int sec; // ticks as done in system timer
   int hz; // frequency
   void init_freq(int _hz); // function that sets frequency as per the required quantum

   public:
   RoundRobinScheduler(); // constuctor
   virtual void yield(); // yileding the next thread
   virtual void resume(Thread * _thread); // resume the blocked thread
   virtual void add(Thread * _thread); // adding ot the ready queue
   virtual void terminate(Thread * _thread); // terminate the thread in RR
   virtual void handle_interrupt(REGS *_r); // handle the timer or end of quantum interrupt!
};



#endif
```

---

2. Scheduler.C

a. Scheduler::Scheduler()

i. This simply initializes a zero sized ready queue

```
Scheduler::Scheduler() {
    size_q = 0; // initializing the ready queue size as 0
    Console::puts("Constructed Scheduler.\n");
}
```

b. void Scheduler::yield()

i. Here we first disable interrupts if they are in enable mode
ii. Get the first/ front element from queue using dequeue method of the queue
iii. Since we dequeued – reduce the queueu saize by 1
iv. Enable interrupts
v. Dispatch the thread to the CPU as it got the turn

```
void Scheduler::yield() {
  //assert(false);
  // get next thread in ready queue
  // disabling the interrupts before yeilding the next thread to CPU
  if (Machine::interrupts_enabled())
    Machine::disable_interrupts();
  // get the thread in the froint of the queue to give it a change to get the CPU as per the expectation of FIFO schedule
  Thread * next_thread = readyQ.dequeue();
  //decrementing the size of the queueu as we rermoved one element
  size_q--;
  // dequeue is done - now we re-enable the interrrupts
  if (!Machine::interrupts_enabled())
    Machine::enable_interrupts();
  // dispatch the dequeued thread to the CPU
  Thread::dispatch_to(next_thread);
}
```

c. void Scheduler::resume(Thread * _thread)

i. Here we first disable interrupts if they are in enable mode
ii. We simple call the add function to add the queue to ready queue as that's the purpose of resume!
iii. Enable the interrupts back

```
void Scheduler::resume(Thread * _thread) {
    // disabling the interrupts before yeilding the next thread to CPU
  if (Machine::interrupts_enabled())
    Machine::disable_interrupts();
    // adding the new thread to ready queue
  Scheduler::add(_thread);
  // resume done- re-enable interrupts
  if (!Machine::interrupts_enabled())
      Machine::enable_interrupts();
}
```

d. void Scheduler::terminate(Thread * _thread)

i. Here we first disable interrupts if they are in enable mode
ii. Find the thread passed if it is in the list
iii. We do this by dequeuing each element and then if it matches id of the passed thread we break else we enqueue it back.
iv. If the thread is found we delete it as dequeued already

```
void Scheduler::terminate(Thread * _thread) {
// find the thread in the lsit
// if the thread is present then dequeue it and remove it
// if not found then enqueue it back
// we had to dequeue becasue there is no other way to traverse
 if (Machine::interrupts_enabled())
   Machine::disable_interrupts();
  int found_th = 0;
  for (int i = 0; i < size_q; i++) {
    Thread * th_this = readyQ.dequeue();
    if(th_this->ThreadId() == _thread->ThreadId())
    {
      found_th = 1;
      break;
    }
    else{
      readyQ.enqueue(th_this);
    }
  }
  if(found_th == 1)
  {
    size_q--;
    Console::puts("Thread terminated!\n");
  }
   if (!Machine::interrupts_enabled())
     Machine::enable_interrupts();
}
```

e. void Scheduler::add(Thread * _thread)

i. here we just enqueue the element and increment the counter of number of threads in queue.

```
void Scheduler::add(Thread * _thread) {
    // disabling the interrupts before yeilding the next thread to CPU
  if (Machine::interrupts_enabled())
    Machine::disable_interrupts();
    // enqueue the thread
  readyQ.enqueue(_thread);
  size_q++; // queue size increases
    // add done- re-enable interrupts
  if (!Machine::interrupts_enabled())
      Machine::enable_interrupts();
}
```

f. RoundRobinScheduler::RoundRobinScheduler()

i. Here we first initialize the queue size, frequency and interrupt handler.

ii. We also now call the set frequency function which will help us to get interrupt after the quantum is passed.

```
RoundRobinScheduler::RoundRobinScheduler()
{
  // we define  a new Q for RR
  RRQ_size = 0;
  sec = 0; //ticks
  hz = 5; // frequencyt to be set
//  SimpleTimer * RRQtimer = new SimpleTimer(1000/EOQ);  // here i need 20Hz frequency for
  InterruptHandler::register_handler(0, (InterruptHandler *)this); // registering the interrupt tyo the handler
  init_freq(hz); // initiate the timer with quantum selected
  Console::puts("RoundRobin Scehduler is created\n");
}
```

g. void RoundRobinScheduler::init_freq(int _hz):

i. This functions I picked from machine code of timer where we set the timer frequency to the required Hz value.

```
void RoundRobinScheduler::init_freq(int _hz) {
    // here we do dome low level stuf to initiate the frequency of the timer
    hz = _hz;
    int factor = 1193180 / _hz;
    Machine::outportb(0x43, 0x34);
    Machine::outportb(0x40, factor & 0xFF);
    Machine::outportb(0x40, factor >> 8);
}
```

h. void RoundRobinScheduler::handle_interrupt(REGS *_r):

i. Here we resume the thread after the interrupt of eoq arrives after quantum arrives.

ii. Finally yield() is called from interrupt handler.

```
void RoundRobinScheduler::handle_interrupt(REGS *_r) {
    sec++; // we increase the timer count
    if (sec >= hz )
    {
      sec = 0;
      Console::puts("Quantum passed- pre-empting required\n"); // quantum passing interrupt
      resume(Thread::CurrentThread());
      yield(); // get new thread and yioeld
    }
}
```

i. **void RoundRobinScheduler::yield():**

i. Here we inform the handler not to worry as we have handled the EOQ interrupt.

```
void RoundRobinScheduler::yield()
{
  Machine::outportb(0x20, 0x20); // informing interrupt handler that its being handled
  if (Machine::interrupts_enabled())
  Machine::disable_interrupts();
  Thread * next_thread = RoundRobinQ.dequeue();
  RRQ_size--;
  if (!Machine::interrupts_enabled())
    Machine::enable_interrupts();
  Thread::dispatch_to(next_thread);
}
```

j. **We keep the add, resume and terminate as it is for RR.**

---

3. **Thread.C**

a. **static void thread_shutdown()**

i. Here we terminate the current thread and delete it.

ii. Finally, we yield it to the next thread in queue.

```
/* EXTERNS */
/*--------------------------------------------------------------------------*/
extern Scheduler* SYSTEM_SCHEDULER;
Thread * current_thread = 0;
/* Pointer to the currently running thread. This is used by the scheduler,
   for example. */
```

```
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.
     */
    // we call the terminate function of system scheduler to shut down the thread
    SYSTEM_SCHEDULER->terminate(Thread::CurrentThread());
    delete current_thread; // deleting the current thread
    SYSTEM_SCHEDULER->yield(); // yeild the next thread from the ready queue
   // assert(false);
    /* 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. */
    Machine::enable_interrupts();
    /* We need to add code, but it is probably nothing more than enabling interrupts. */
}
```

b. static void thread_start()

i. enable interrupts

4. Thread.H

a. Include headers

5. Kernel.C

a. Here we uncomment the code to define the FIFO scheduling, terminating threads, interrupt handling and RR Scheduling.

b. Following changes were made:

i. We write a define statement for selecting either FIFO for RR Scheduling.

ii. If we define pointer to the scheduler we selected using ifdef

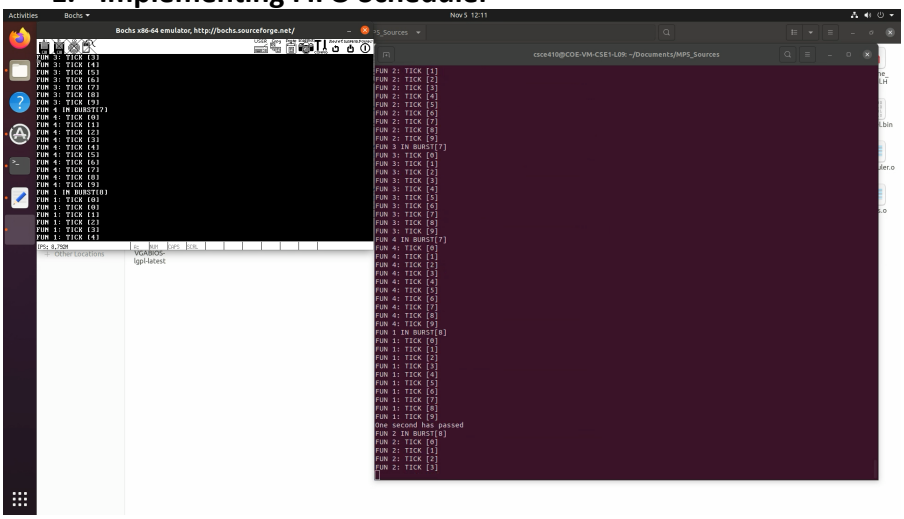
iii. If we use RR scheduler then we don't pass the CPU else we pass it.

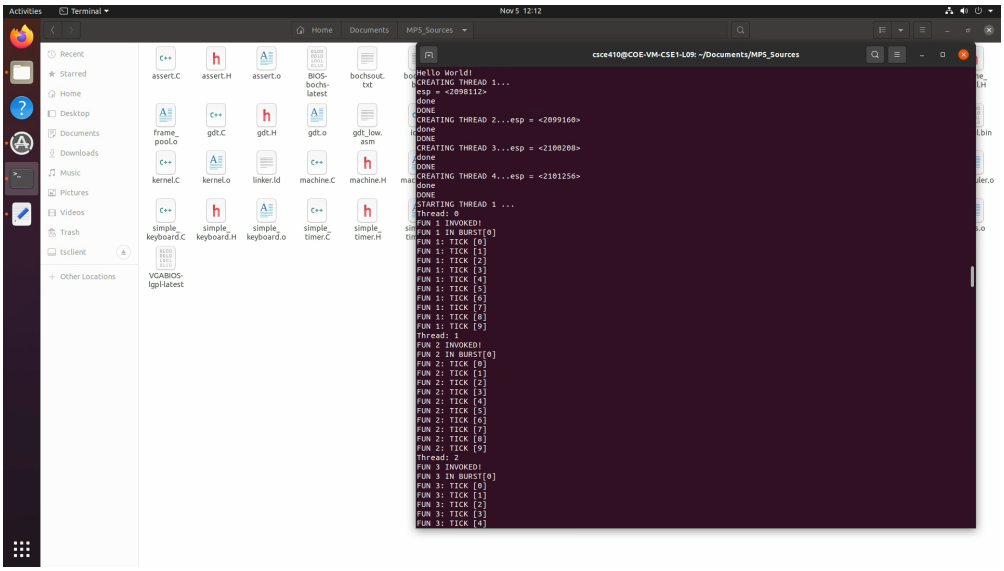
iv. Code has been commented very well wherever changes are made

---

OUTPUTS SNAPSHOTS:

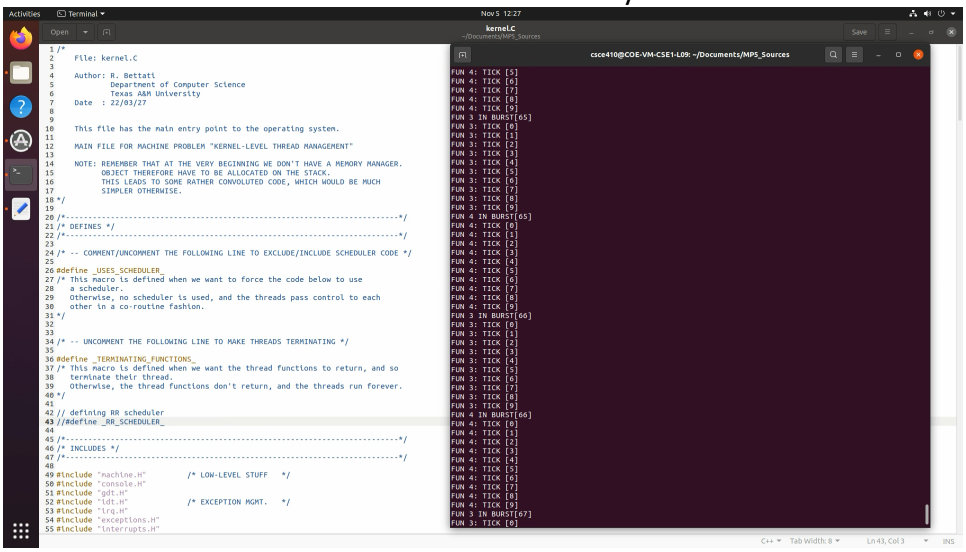
1. Implementing FIFO Scheduler



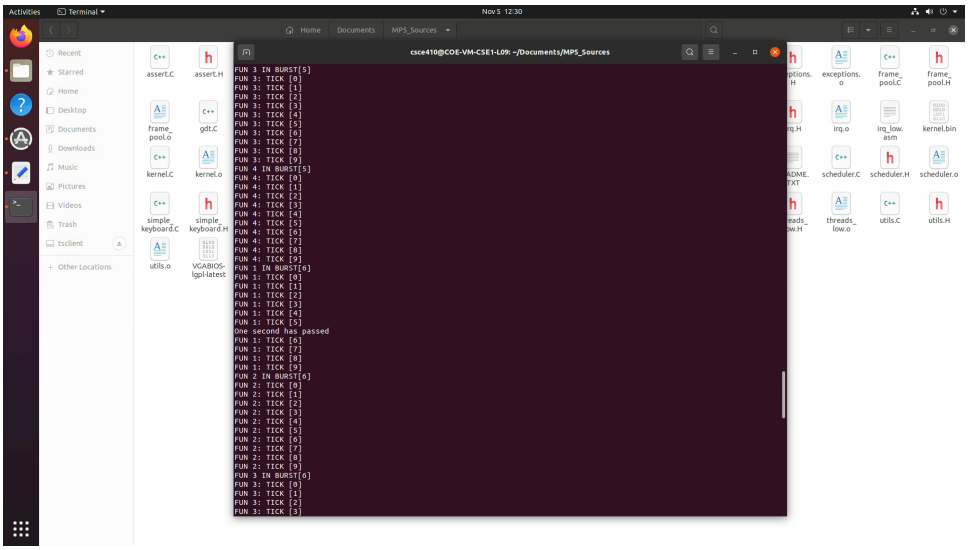


2. Implementing Terminating Thread

Thread 1 and 2 terminate and then 3 and 4 run infinitely



3. Bonus 1: Starting Interrupts



4. Bonus 2 : RR Scheduler