# Design of MP5

The objective of machine problem 5 is to implement a FIFO scheduler firstly, and then to add new threads, and to perform the scheduling operations. And in this design, we also show the first two bonus of this problem and discuss our results. What I submit include 8 files. In order to realize FIFO, I modified scheduler.H, scheduler.C, thread.H and thread.C. For Bonus 1, I also modified these 4 files. And for Bonus 2, I modified simple_timer.H, simple_timer.C and interrupts.C and kernel.C. So directly link these 8 files with other files provided on ecampus, we can realize Round Robin Scheduling with enabled interrupts. Now let's begin.

First of all, we implement a simple FIFO algorithm for thread scheduling. In order not to use "new" and "delete" operators in the scheduler, instead of a linked list, we use a fixed length array which is called ready queue and contains 4 pointers pointing to 4 threads, respectively. And we also set 3 variables, "head_number", "tail_number" and "thread_number", to record the head, the tail and the thread number of our ready queue.

The implementation of a pure FIFO is very simple. There are some tips that we present here. We use a circular array in which head and tail number are always incremented and mod the length of the array. Although resume function and add function are used in different scenarios, they always do the same thing, adding the current thread to the tail of the ready queue. Thus, we set them with the same program. And in yield function, we need to do the context switch, which is realized by calling a low-level dispatch_to function in class Thread.

Finally, we enter into the most important part in FIFO scheduler, the terminate function. In this function, we simply invoke yield() to do the context switch to the next thread without resuming the current thread and leave the memory assignment to thread_shutdown function in thread.C. When the first two threads finish their thread function, in which j reaches 10, their thread function will return with popping the stack until empty. And the last thing popped out is the pointer to thread_shutdown function. Then if we build this function, we can terminate the thread successfully. In thread_shutdown function, we first save a copy of current thread pointer, then invoke the terminate function in the scheduler to yield CPU. Now the current thread is actually the next thread, but don't worry, we just save the copy of the thread we want to terminate and we can use this copy to release the "stack" dynamically allocated in main and the thread in the stack of the memory pool, as is shown in Figure 1.


Figure 1. The modification of thread_shutdown function

Now we start to deal with the bonus. First, we let interrupts work. Interrupts must be disabled during operations of the scheduler like resume and context switch since these critical sections should not be interrupted. However, interrupts should be activated when a thread is running, otherwise, the periodic clock update message is missed. So we only enable interrupts in thread_start function and the end of the yield function. And in the resume, add function and the beginning of the yield function, we disable interrupts. And in order to test our code, we set the timer in kernel.C to 5 instead of 100, now as is shown in Figure 2, the sentence of “one second pass” is output every basically the same interval during a thread is running.

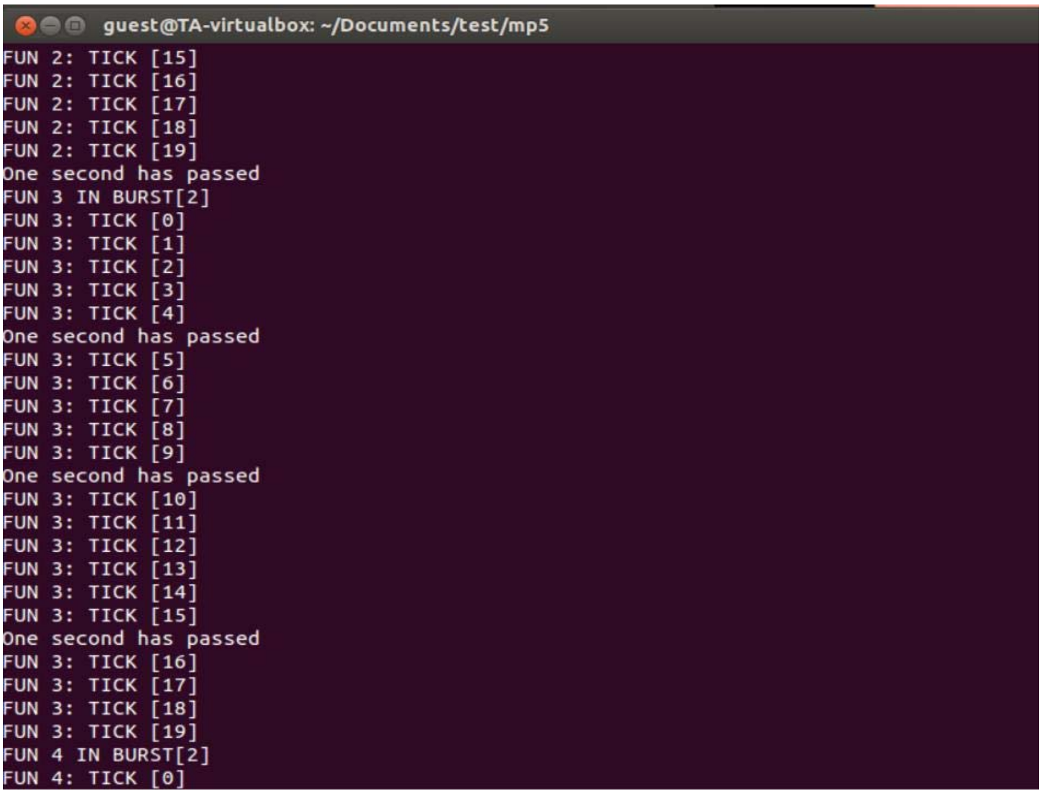


Figure 2. Test of Bonus 1

Now we are in Bonus 2. We change the interrupts handler of SimpleTimer, adding resume and yield, just as the handout tells us. And every time a new thread starts running, the variable “ticks” in timer needs to be reset as 0, therefore we add a reset function in the SimpleTimer class. And we also comment “Machine::enable_interrupts” since we enable interrupts only after threads first start running. At this time, the threads have not been constructed yet, so we do not want interrupts to be activated here. Otherwise, we will face the problem in Figure 3 if we set (SimpleTimer) timer as small value.

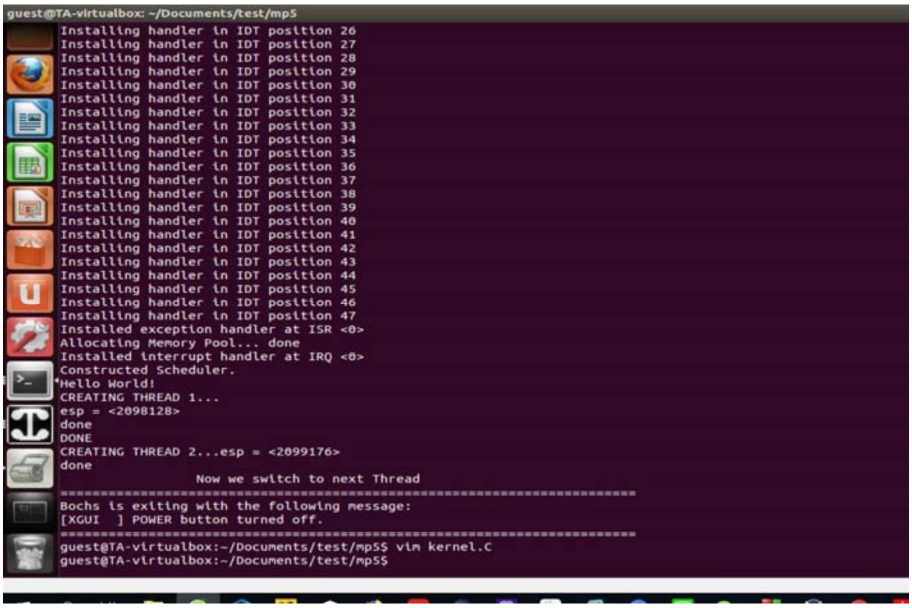


Figure 3

After doing that, we still do not succeed. We only get the results in Figure 4 even if we set (SimpleTimer) timer as very small value, for example, 5. It seems just like a FIFO Scheduling. The problem is that the new thread may not return from an interrupt handler, and the EOI signal may not be sent to the interrupt controller in time. We provide a temporary solution, in which we modified interrupts.C, sending the EOI prior to the interrupt handler. And the results are shown in Figure 5(a), (b) and (c), in which we set the value of (SimpleTimer) timer as 2, 5 and 10 respectively. However, this solution is in some degree dangerous if interrupt handler does not return with a correct statement. Since interrupts.C is really a bottom layer file, that is all things I can do.

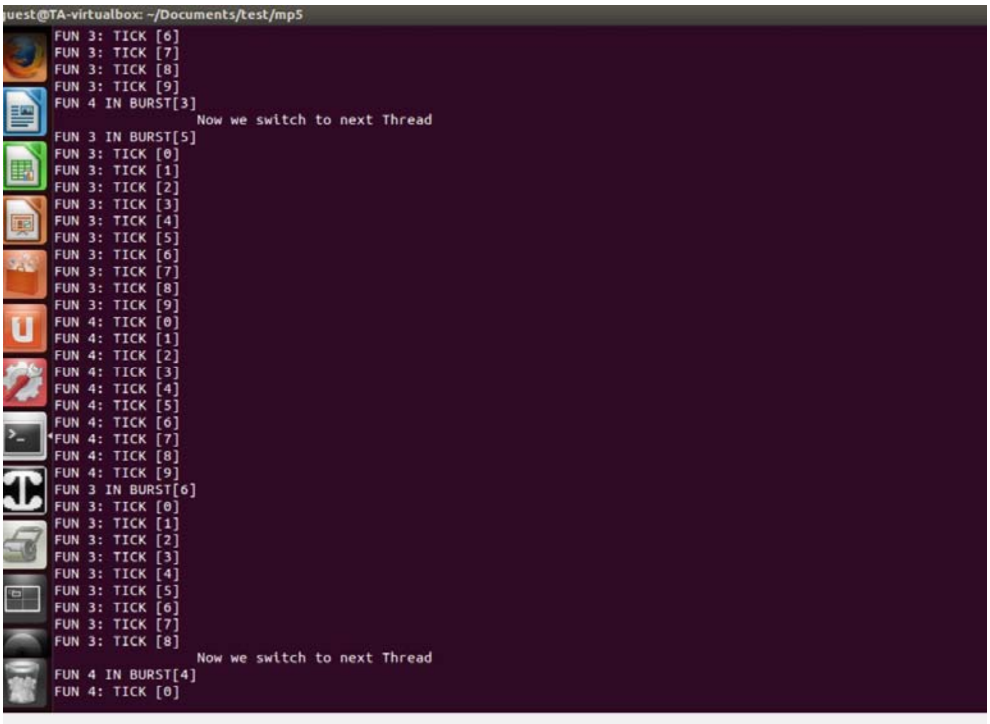


Figure 4

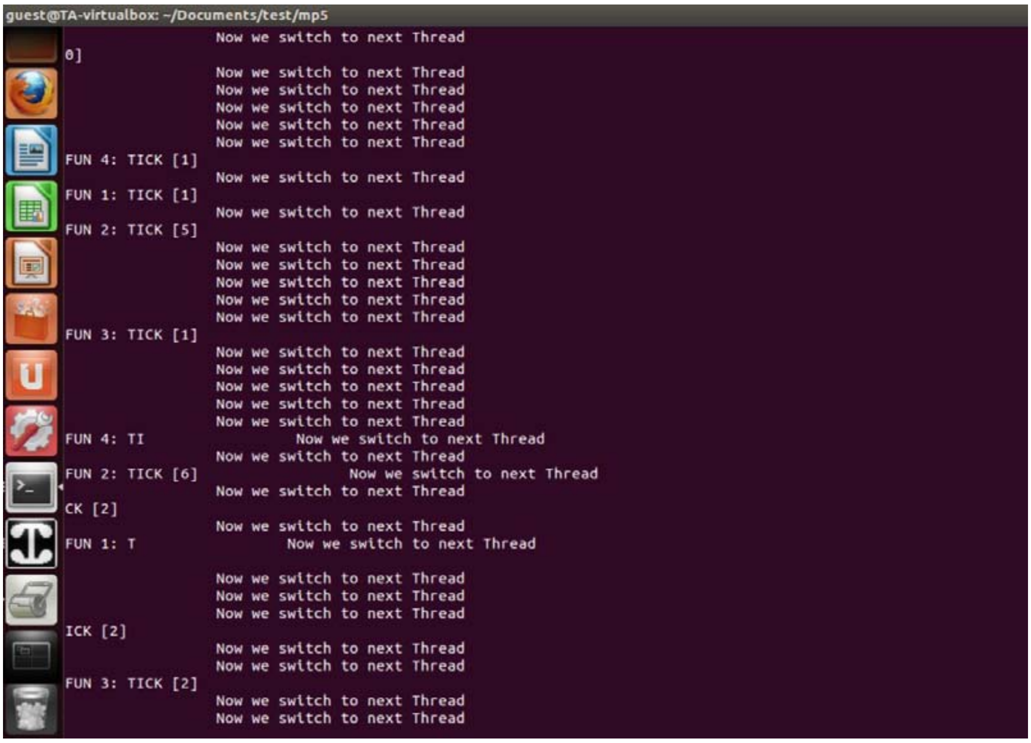


Figure 5(a)

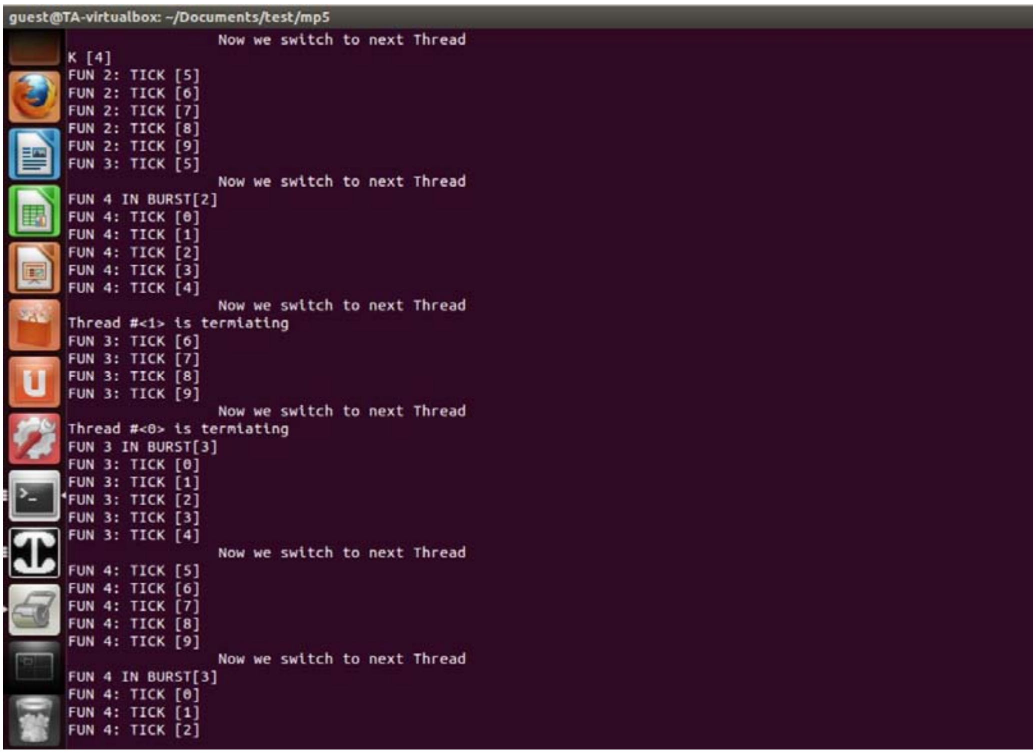


Figure 5(b)

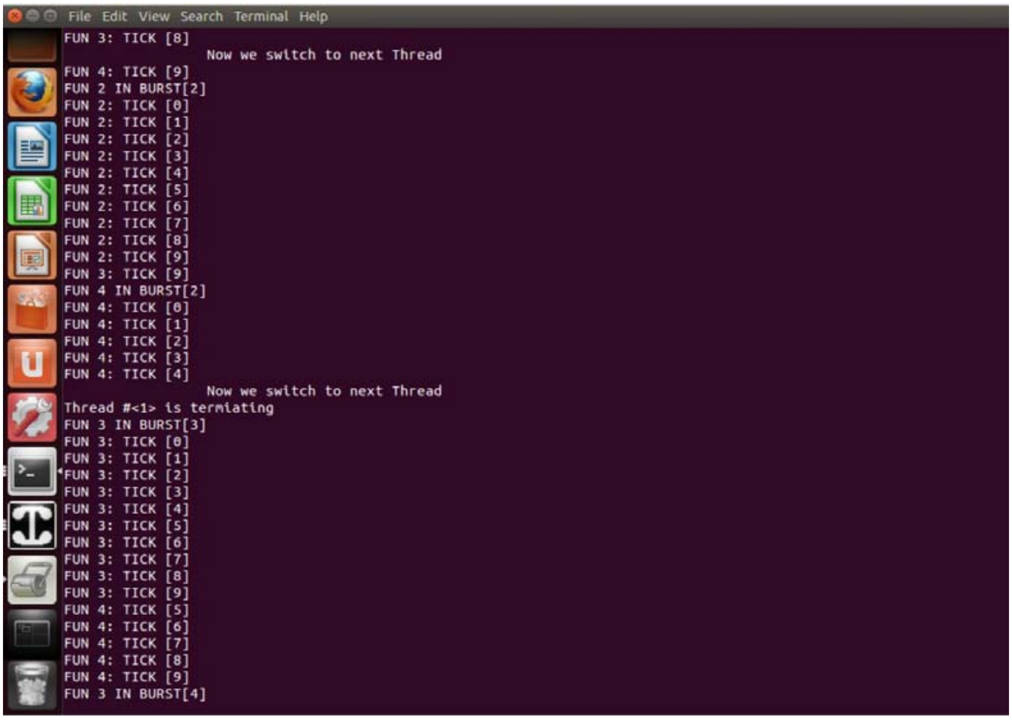
File Edit View Search Terminal Help
FUN 3: TICK [8]
Now we switch to next Thread
FUN 4: TICK [9]
FUN 2 IN BURST[2]
FUN 2: TICK [0]
FUN 2: TICK [1]
FUN 2: TICK [2]
FUN 2: TICK [3]
FUN 2: TICK [4]
FUN 2: TICK [5]
FUN 2: TICK [6]
FUN 2: TICK [7]
FUN 2: TICK [8]
FUN 2: TICK [9]
FUN 3: TICK [9]
FUN 4 IN BURST[2]
FUN 4: TICK [0]
FUN 4: TICK [1]
FUN 4: TICK [2]
FUN 4: TICK [3]
FUN 4: TICK [4]
Now we switch to next Thread
Thread #<1> is termiating
FUN 3 IN BURST[3]
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 [5]
FUN 4: TICK [6]
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
FUN 3 IN BURST[4]

Figure 5(c)