

Design Document
Kernel Level Thread Scheduling

CSCE 611
Operating Systems
MP5 - Fall 2018
by
Himanshu Gupta

Scheduler Design

Bonus Attempted – OPTION 1 , OPTION 2 , OPTION 3

NOTE: makefile is modified

We design a very basic FIFO scheduler for kernel level threads. To implement this FIFO functionality, we need a simple class that has this property of FIFO queue. Therefore, we first create a simple class “Queue” which is nothing but simply implements a FIFO queue. It will maintain a list of Thread structures and the next pointer for thread. We have two basic constructors for this class, with and without the Thread argument as given in the source code. This class provides two basic queue operations which are enqueue and dequeue. For enqueueing a thread object in the ready queue, we just check if there is an existing member in the queue already or not. If not we update the next pointer. For dequeue operation, we take the top thread and return.

The scheduler class declares a queue object as a ready queue and also has a size member. Next, to implement the scheduler api, we first write its constructor. This only sets the queue size to 0. The yield api will first check if there is any thread available to switch and then dequeue the thread from ready queue. The resume and add in the scheduler are very similar where we enqueue the given thread to the ready queue.

We also add the terminate api, where we loop through the ready queue and find the thread with same thread id and then delete it. We also need to implement the thread shutdown api the thread class. This will first call the terminate from the scheduler and then delete the current thread. Finally, the scheduler will call the yield to bring in another thread from the ready_queue.

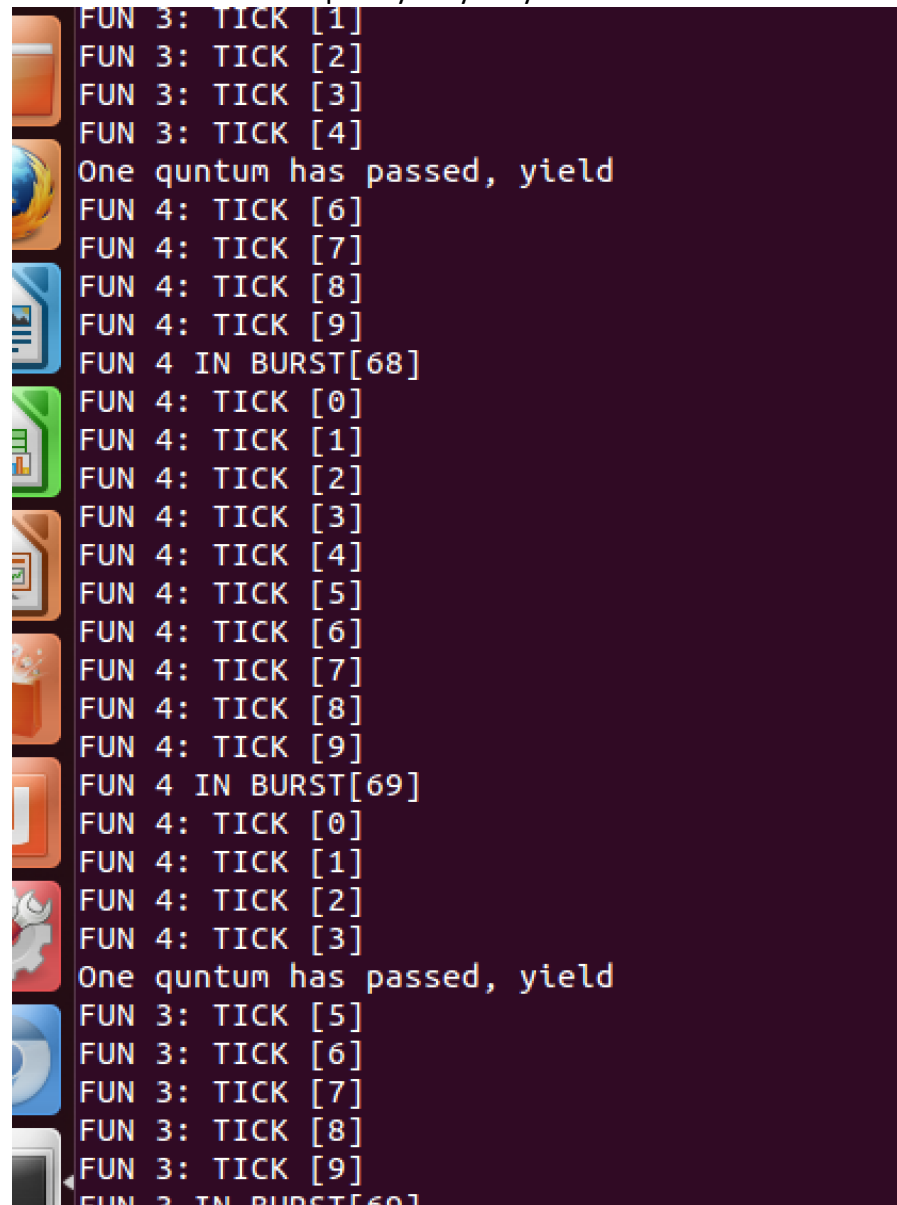
Bonus options: I implemented the bonus **option 1 and 2**. For **option 1**, we add the enable interrupts api while we start the thread. For **option 2**, we need to enable round robin scheduling. This needs to generate a timer based interrupt at every 50 ms to allow make the running thread yield. To make this work, we update the dispatch interrupt api to allow EOI signal after the interrupt has occurred and then in the handle interrupt api, we do a yield to next thread. I added a new static api for handling this yield of thread, which simply does a resume and yield on the scheduler, which will put the current thread in the end and load a new thread.

TESTING

For testing the basic functionality of scheduler, we first define the macro `_USES_SCHEDULER_` which will start calling the new scheduler. This will test the basic scheduler functionality. We also define the macro `_TERMINATING_FUNCTIONS_` for testing the terminating part.

To see the effect of correct handling of interrupts, after enabling the interrupts, we can see the debug message of “one second passed” which proves that it works fine. After this we test the Round robin functionality which can be seen by below screen shot how we switch to new thread

after each interrupt at every 50 ms. Please note that the time calculation of 50 ms is not exactly accurate as the CPU frequency may vary.

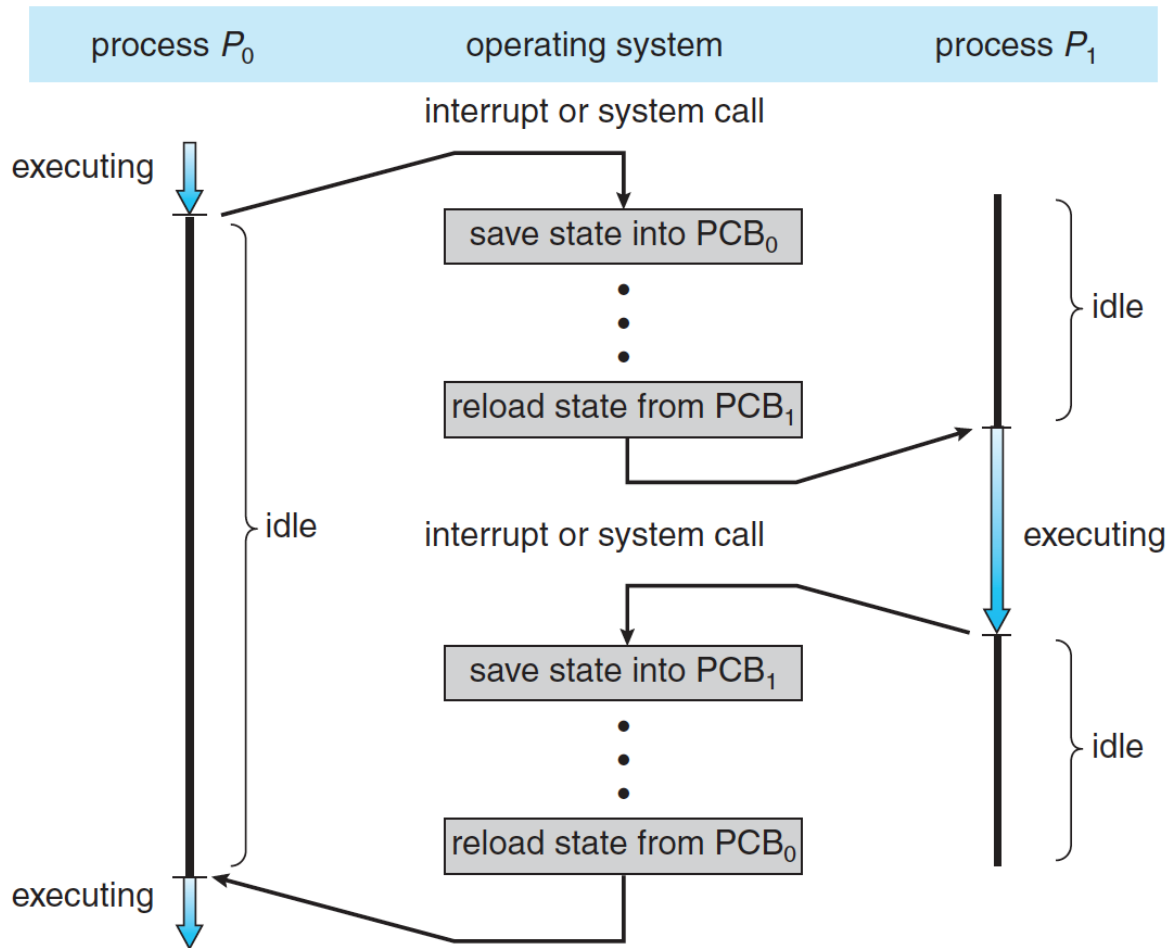


```
FUN 3: TICK [1]
FUN 3: TICK [2]
FUN 3: TICK [3]
FUN 3: TICK [4]
One quantum has passed, yield
FUN 4: TICK [6]
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
FUN 4 IN BURST[68]
FUN 4: TICK [0]
FUN 4: TICK [1]
FUN 4: TICK [2]
FUN 4: TICK [3]
FUN 4: TICK [4]
FUN 4: TICK [5]
FUN 4: TICK [6]
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
FUN 4 IN BURST[69]
FUN 4: TICK [0]
FUN 4: TICK [1]
FUN 4: TICK [2]
FUN 4: TICK [3]
One quantum has passed, yield
FUN 3: TICK [5]
FUN 3: TICK [6]
FUN 3: TICK [7]
FUN 3: TICK [8]
FUN 3: TICK [9]
FUN 3 IN BURST[69]
```

Option 3

For implementing process based scheduling in this MP, we need to add the support for memory management and page table. This needs the code that we have implemented in MP4 and MP3. The basic property that we need is implement a process control block (PCB), which will on interrupt, save the state of the running process in PCB and load the process which will run next on CPU. To implement in our solution, we will have separate page tables for each process. Therefore, on any switch we will need to reload the page table base register and the process id for the new process needs to be maintained in the process scheduler like the previous thread scheduler that we already have. Therefore, we will have a two-level scheduler each for threads

and process, which will allow switching cpu among threads from different process. The switch between process will look somewhat like the below representation of PCB



There are changes in several files for this MP listed below, If any is missing please contact me.

Scheduler.C
 Scheduler.H
 Thread.C
 Thread.H
 Kernel.C
 Queue.H (**new file**)
 Makefile
 Interrupts.C
 Simple_timer.C