

CSCE 611

Machine Problem 5: Kernel-Level Thread Scheduling

Submitted By – Vaibhav Rawat

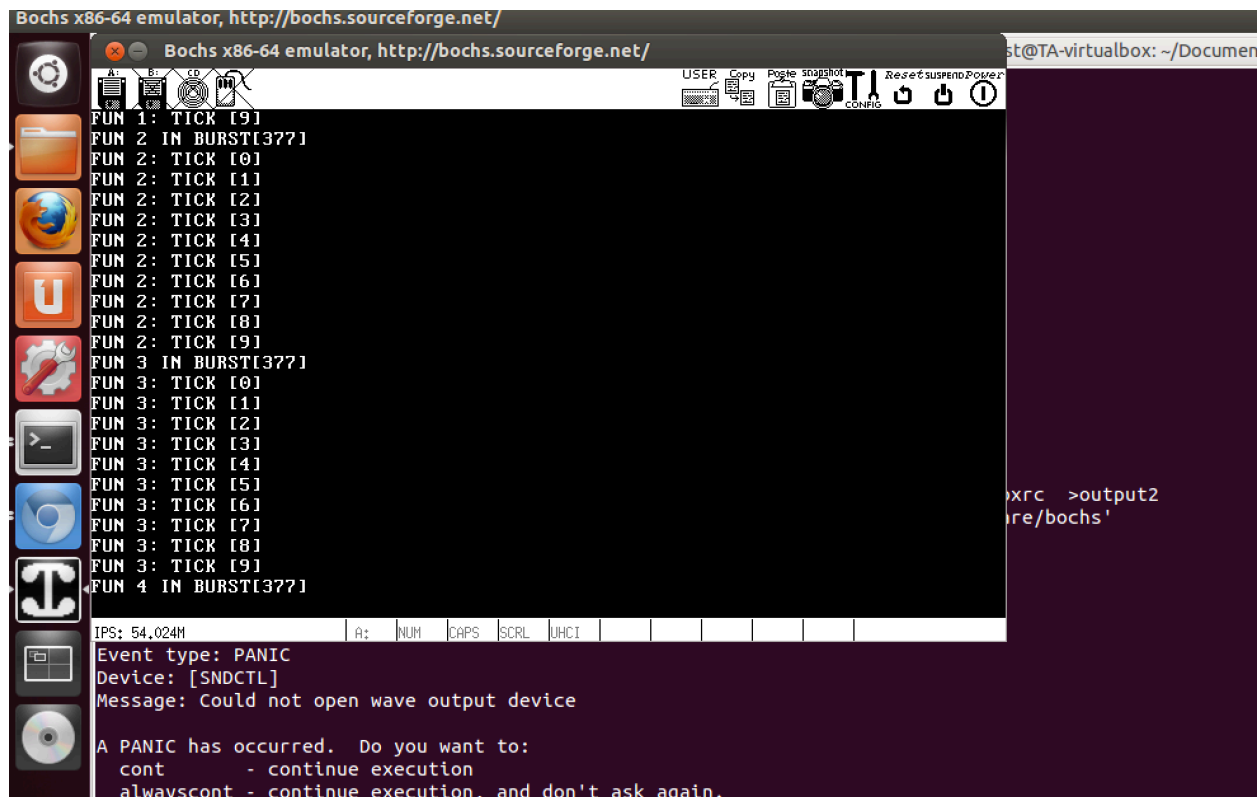
UIN: 626008171

Main task in this assignment is to work on top of existing source code for threading and context switching interface to implement a scheduler for kernel level threads. Initially all threads in kernel.C are non-terminating. With changes for scheduler, it supports handling of terminating threads as well.

I implemented a First-In-First-Out scheduler. The scheduler maintains a ready queue. It is a linked list of threads (thread control block) that are waiting to get to the CPU. Each node of the list points to the thread it corresponds to and the next element in the queue. Ready queue has two operations enqueue and dequeue – enqueue puts the thread to be queued to the back of the list and dequeue pops element from the front of the list.

Ready queue is used by the add(), resume(), yield() and terminate() functions. yield() function checks for whether the queue is empty or not. If queue is not empty, it dequeues thread from front of the list and uses dispatch_to() function to invoke a context switch to run the new thread. resume() and append() functions take in thread as parameter and enqueues the thread to the back of the ready queue, they are called for threads that were waiting for an event and for making threads runnable after creation. Terminate function takes a thread as parameter and traverses through the ready queue until it finds the thread which has to be removed from the queue. While traversing through the queue, it dequeues thread from the front of the list and enqueues it at back, until it finds the matching thread. Terminate may be called from other thread or same thread, thread_shutdown() terminates threads releasing all resources associated with it and calling yield to dispatch to next ready queue thread.

I also tried to test out cases when there is no thread in queue and the current running thread wants to give up CPU. For that scenario, I successfully tried using a dummy looping thread, code for which is commented in kernel.C

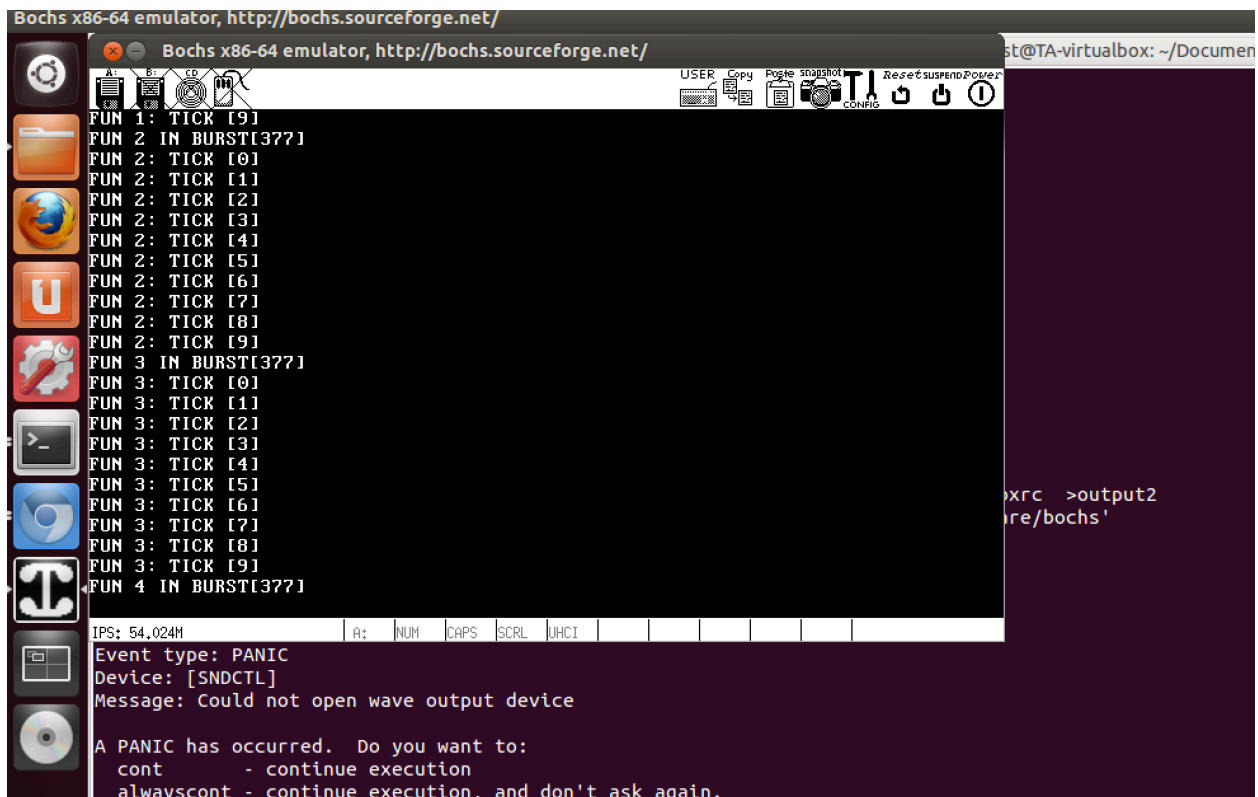


Once we schedule thread 1 and push thread2, thread3 and thread4 in ready queue in kernel.C, threads relay switching in sequence 1,2,3,4. I enabled my scheduler using `_USES_SCHEDULER` and also enabled termination of threads 1 and 2 using `_TERMINATING_FUNCTIONS`. So, after some time thread1 and thread2 terminate and CPU switches among thread3 and thread4.

I have attempted BONUS OPTION1 and OPTION2. For option1, problem seen is that threads are disabled once we start the first thread. It can be observed from the fact that clock update message is missing. Interrupt Enable Flag(IF) in EFLAGS status register is set to zero in our context when we create threads. So, I tackled this problem by enabling interrupts in start thread method. It will work because start thread function is called after interrupts are disabled while creating threads. I verified the solution by checking the output files which correctly display timer messages of One second passed. I have shared output file for the same.

For option2 – implementation of Round Robin, I modified the code in dispatch interrupt. It informs the interrupt controller(PIC) that interrupt is being handled before call to `handle_interrupt`. As a result, context switch is allowed during handler process. Round Robin requires interrupt at periodic interval of 50

milliseconds in our case. For this, I modified simple timer's handle interrupt function to count till 50 milliseconds and at every 50 milliseconds, put the current thread to ready queue and switch to next thread in queue. Kernel.C is also modified to remove pass_on_CPU since we implicitly switch threads using yield() after quantum of 50 milliseconds in a Round Robin manner. With above changes, PIC is always aware while implicit context switches happen in a Round Robin and so the above solution works correctly. I have verified my solution with flags `_USES_SCHEDULER` and `_TERMINATING_FUNCTIONS`. I can see that initially all threads – thread1, thread2, thread3 and thread4 execute in a Round Robin quantum of 50 msec. After some time, thread1 and thread2 terminate and from then thread3 and thread4 execute in a Round Robin manner with quantum of 50 msec. I have included a final snapshot of my output as follows. Output files are also included with my submission.



The screenshot shows a Bochs x86-64 emulator window. The main terminal area displays a log of thread execution:

```
FUN 1: TICK [9]
FUN 2 IN BURST[377]
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 IN BURST[377]
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 IN BURST[377]
```

Below the log, the status bar shows "IPS: 54,024M". The bottom panel displays an error message:

```
Event type: PANIC
Device: [SNDCTL]
Message: Could not open wave output device

A PANIC has occurred. Do you want to:
cont      - continue execution
alwayscont - continue execution, and don't ask again.
```

The emulator window has a toolbar with icons for USER, Copy, Paste, Snapshot, CONFIG, and Reset/Suspend/Power. The title bar indicates the URL "Bochs x86-64 emulator, http://bochs.sourceforge.net/".

For option3, I used my code from previous MPIs to have multiple processes requesting memory from process pool using page tables (having separate address spaces). But I couldn't handle switching across multiple address spaces.