Design Document
Kernel-Level Thread Scheduling

CSCE611
Operating Systems
MP5- Fall 2022

by

Tanu Shree

DEPARTMENT OF

COMPUTER SCIENCE AND ENGINEERING

TEXAS A&M UNIVERSITY, COLLEGE STATION

This Machine problem is mainly about adding the scheduling of multiple kernel-level threads. Following things are to be achieved as part of this problem:

1. Implement FIFO scheduler with non-terminating threads
2. Implement FIFO scheduler with terminating threads
3. Enabling/Disabling interrupts (bonus option 1)
4. Implement Round Robin Scheduler(bonus option 2)

A ready queue is maintained containing the list of the threads that are waiting to execute. This is done by creating a class queue using structure. The corresponding enQueue(thread) and deQueue () methods are included to proceed with the scheduling. The structure contains the variables, thread, and next pointer of queue: Thread * thread and Queue * next.
The method enQueue(thread) is defined to add the thread into the ready queue when either it is executed first or is waiting for an event to occur or when preempted to give away the CPU.
The method deQueue () is used to take the thread out of the queue when it is dispatched to the CPU.

Implement FIFO scheduler with non-terminating threads

To implement FIFO scheduler, three methods in scheduler.C are defined: add (), resume () and yield (). For now, the change is done only for non-terminating threads.

Resume (): adds the currently running thread at the end of the queue when either it waits for some event to occur, or it has been preempted to give away the CPU.
Add (): makes the given thread runnable by the CPU. It adds the thread into the queue using resume ().
Yield (): is called by the running thread to give up the CPU and the scheduler selects the next thread in the queue to dispatch to the CPU.

Implement FIFO scheduler with the terminating threads

To implement this, the changes are done in the thread.C and scheduler.C files, methods thread_shutdown () and terminate (Thread * thread) are modified respectively.

thread_shutdown (): is called whenever the thread returns from the thread function. This is a locally created function. The current thread is taken out of the ready queue, its memory is released and then yield is called to dispatch the next thread waiting in the queue.
Terminate (): is called in the thread_shutdown method. In this function, the currently running thread id is stored in a local variable and then checked if it is same as the element in the front of the queue. If yes, it is taken out and deleted. If not, the list is traversed and when it finds the currently running thread in the queue, it deletes it.

Enabling/Disabling interrupts (bonus option 1)

When the thread starts, it is necessary to enable the interrupts. Also, at the time of any enqueue or dequeue operation, interrupts should be disabled to maintain the mutual exclusion. Changes are made in the Thread.C file, where interrupt is enabled in the thread_start () and in the file scheduler.C, where the interrupts is enabled in the yield () after the dequeue operation and interrupts is disabled in the resume (), before the enqueue operation.

Implement Round Robin Scheduler (bonus option 2)

To implement the Round Robin scheduler, a class RRscheduler is derived from two classes scheduler and InterrruptHandler with few changes in the methods. Few additional methods are added to manage the time quantum. In the RR scheduling implemented here, each thread gets the time quantum of 50ms, after which the interrupt is raised, the current thread is added to the end of the ready queue and the next thread is dispatched through yield ().
The methods add (), resume () and terminate () are directly derived from the base class, so no need to define again. Changes are made in the yield (), to reset the number of ticks and inform the PIC about the interrupts handled, by sending EOI message.

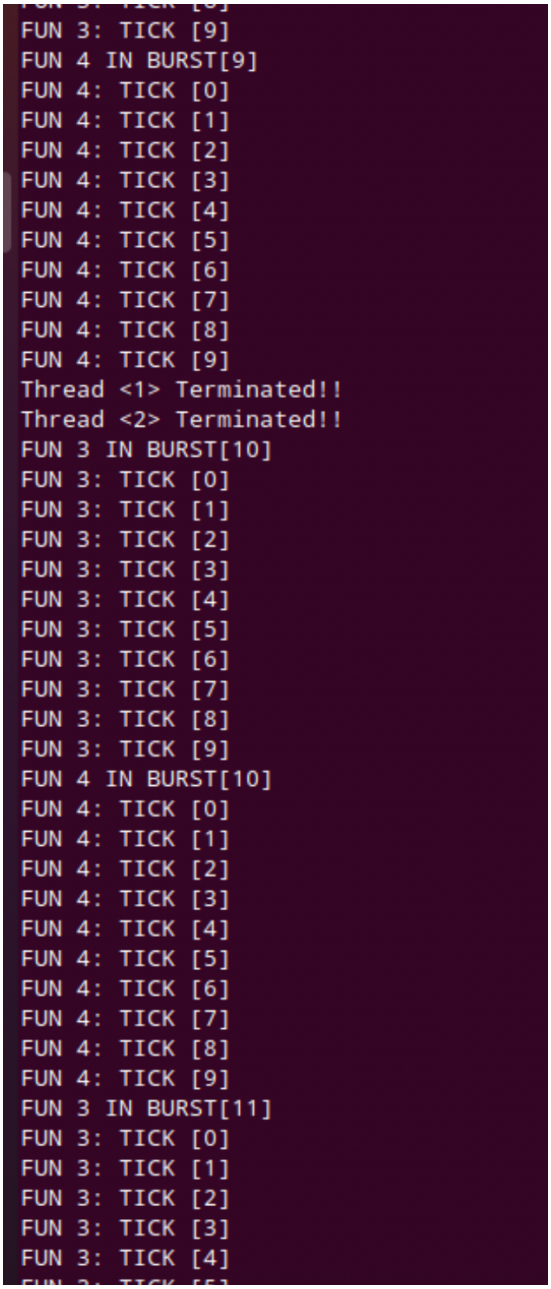
Handle_interrupt(REGS * r): is derived from class Interrupthandler. Here it is checked if the time quantum is reached (50 ms here) and once the quantum is reached, the ticks variable is reset, current thread is resumed and then yield ().
Set_frequency(int time_quantum): is taken from simple timer. It sets the interrupt frequency for Round Robin.
Yield (): is modified here to inform that interrupt is handled by sending the EOI message. It ensures that every thread gets 50ms and they get executed in Round Robin fashion. The ticks count is also set to zero.

Testing and Screenshots:

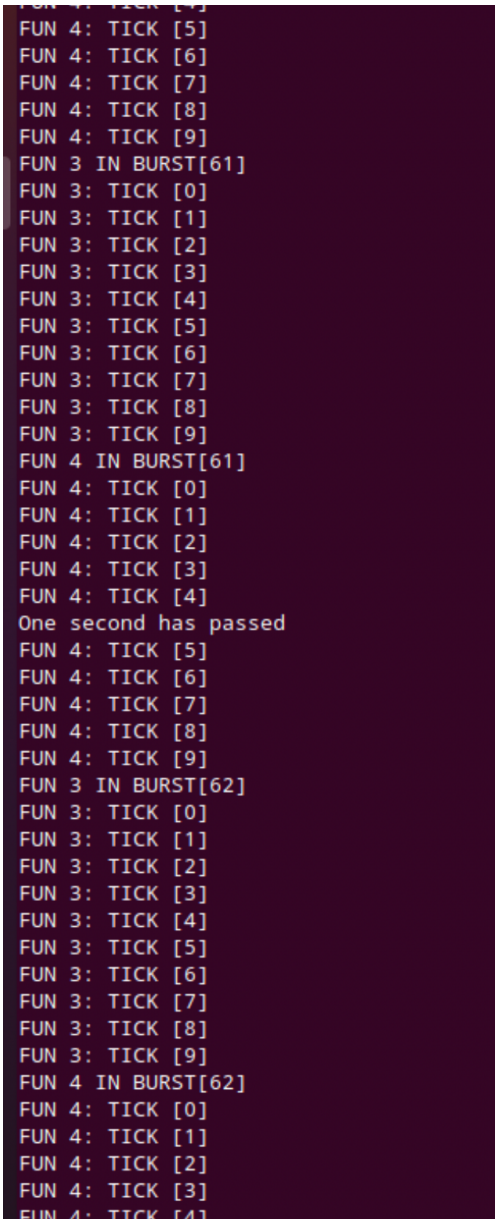
1. **Implement FIFO scheduler with non-terminating threads:** The testing for this implementation is done by uncommenting _USES_SCHEDULER_ macro in kernel.C. Scheduler object is created and Scheduler class functions are invoked for every thread, (Fun1 to Fun4). Each thread ticks 10 times and passes the CPU to next thread (infinite).

```
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
FUN 1 IN BURST[5857]
FUN 1: TICK [0]
FUN 1: TICK [1]
FUN 1: TICK [2]
FUN 1: TICK [3]
FUN 1: TICK [4]
FUN 1: TICK [5]
FUN 1: TICK [6]
FUN 1: TICK [7]
FUN 1: TICK [8]
FUN 1: TICK [9]
FUN 2 IN BURST[5857]
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[5857]
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[5857]
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 1 IN BURST[5858]
FUN 1: TICK [0]
FUN 1: TICK [1]
FUN 1: TICK [2]
```

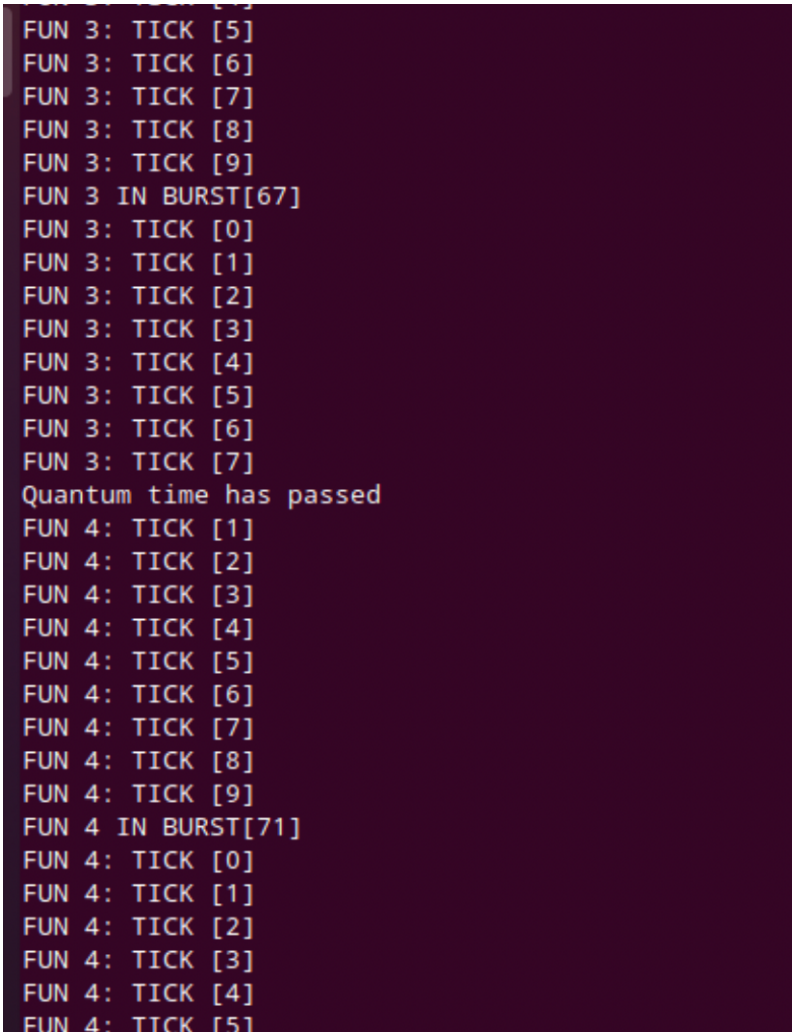
2. **Implement FIFO scheduler with the terminating threads:** This implementation is tested by uncommenting _TERMINATING_THREADS_ macro in the kernel.C file. Threads 0 and 1, terminate after 10 Bursts and context switches happen between threads 2 and 3. For simplicity in testing, a macro-DEFAULT_NB_ITER is created and set to 100. This is used in the for loop to give the upper bound. As a result, threads 0 and 1 terminate after 10 bursts and thread 2, 3 context switches among themselves and finally terminate after 100 bursts.



3. **Enabling/Disabling interrupts (bonus option 1):** When the interrupts are enabled and disabled a message “one second passed” is shown whenever the thread is interrupted.



4. **Implement Round Robin Scheduler (bonus option 2):** To test this implementation, a macro _USES_RRSCHEDULER is created and used to override the FIFO scheduler when defined. It creates a RR scheduler object passing the time quantum instead of using simple timer.
   In the output screenshot, threads 2 and 3 are context switching. Quantum time has passed for thread 2 at Tick [7], and CPU is given to thread 3 at tick [1], which would have been interrupted earlier. In the 2nd screenshot, thread 2, which was in the ready queue gets the CPU back when thread 3 gets interrupted at Tick [8]. Note that, thread 2, starts again at tick [8].



```
FUN 4: TICK [9]
FUN 4 IN BURST[93]
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]
Quantum time has passed
FUN 3: TICK [8]
FUN 3: TICK [9]
FUN 3 IN BURST[68]
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 3 IN BURST[69]
FUN 3: TICK [0]
FUN 3: TICK [1]
FUN 3: TICK [2]
FUN 3: TICK [3]
FUN 3: TICK [4]
```