CSCE - 611

MP5: Kernel Level Thread Scheduling

DESIGN DOCUMENT

Submitted by: RAJENDRA SAHU (731008796)
github-repo : CSCE-410-611-fall-2021/rpsahu_CSCE611

OPTIONS ATTEMPTED

- **Option 1: Fixing the interrupt management:** Interrupts have been enabled by default in the code.
- **Option 2: RR Scheduling:** Comment the macro _FIFO_SCHDEULING in *kernel.C* to select the RR scheduler.

I couldn't establish critical sections in all the parts of the RR scheduler since it was causing some race conditions. However, I could almost implement the RR scheduler apart from some corner cases in scheduling, probably because of the above lacking support in interrupt management. But the current RR scheduler doesn't cause any data/thread loss. So, the hazardous race conditions have been handled.
Comment the macro _FIFO_SCHDEULING in *kernel.C* and assess accordingly from the console output.

INTRODUCTION:

This MP deals with thread creation & handling them in specific scheduling policies (FIFO & RR in this MP).

SOURCE CODES MODIFIED/ADDED:

1. *kernel.C: Commenting the _TEST_PAGE_TABLE to test the virtual memory allocator.*
2. *scheduler.C: Core Implementation*
3. *scheduler.H: Core Implementation*
4. *thread.C: pushing 1 in 9th bit of EFLAGS register so that interrupt flag is enabled*
5. *simple_timer.C: Core Implementation*
6. *simple_timer.H: Core Implementation*
7. *console.C: modifications as per the instructor provided solution for the console redirection*
8. *console.H: modifications as per the instructor provided solution for the console redirection*
9. *bochsrc.bxrc: modifications as per the instructor provided solution for the console redirection*

ABSTRACTION INTRODUCED: A structure *tcb_node_s* as the datatype of the node of the ready queue.

CORE IMPLEMENTATION (Only the incremental additions/modifications have been mentioned, the rest of the code stays same as provided):

1. ***scheduler.C constructors:***
   We maintain 3 classes here.
   a. Scheduler () (Base class)
   b. FIFOScheduler() (Deriving from base class)
   c. RRScheduler()(Deriving from base class)

- The *Scheduler()* constructor initializes the head & tail nodes. The FIFOScheduler() does nothing extra. The RRScheduler() initializes a rr_yield_flag

2. ***Scheduler::add():*** This is the non-virtual function in the base class that basically adds a thread to the ready queue using the head & tail pointer. We create a new node using the new operator here.

3. ***Scheduler::yield():*** This is the function where the current running thread yields the CPU. Please take note in my implementation the head node of the queue is the next thread that should be dispatched. The current thread is never in the queue. So in yield(), I have just updated the head pointer & deleted the node that just got dispatched. In the *FIFOShceduler::yield()* I have added a critical section to make the node deletion mutually exclusive. In *RRScheduler::yield(),* I check the *rr_yield_flag* whether it was a voluntary yield by the thread or the quantum timer pre-empted the thread. If it's a quantum timer yield, then I reset the timer if almost the time quantum has passed in order not to penalize the next thread.
4. ***Scheduler::resume():*** This function gets usually called when a thread has to be pre-empted & added at the end of the ready queue. It simply calls the add function which adds the node at the tail. This is a virtual function although the functionalities are the same for both of the derived class.

5. ***Scheduler::terminate():*** This function is called when we have to terminate the thread. This gets called internally when the thread function ends. It can also be called with a specific thread to be deleted from the queue. When it's a running thread that is about to end we simply call *yield()* which will delete the node without adding it to the queue again. If it's a specific thread, then we traverse the queue & delete that specific node.

6. ***RRScheduler::handle_rr_quantum():*** The interrupt handler of the quantum timer calls this function. The goal is to dispatch the next thread. We call our established flow of resume & yield. We also set the *rr_yield_flag* here to differentiate this scenario.

7. ***thread.C:*** Two things have been modified here. One is I have added the *scheduler::terminate()* call in the *thread_shutdown()* function to terminate the thread. We also delete the *current_thread* here. Since *thread_shutdown()* function ptr gets pushed to the thread's context, it gets popped & called when the *thread_function* has ended. Second thing is to push nonzero ELAG, specifically setting 9th bit which is the IE flag to enable interrupts for the option 1. Also added the extern scheduler variable here.

8. ***kernel.C:*** Only mentioning the modifications
   Uncommented the _USES_SCHEDULER & _TERMINATING_FUCNTIONS macros to test the code.
   Added macro _FIFO_SHCEUDLING to select between FIFO scheduler & RR scheduler.
   Instantiated the *EOQTimer*
   Moved the *enable_interrupts* call after the threads gets added to ensure mutual exclusion.

9. ***simple_timer():*** I have created the *EOQTimer()* here. This has been derived from SimpleTimer. The handler of this new timer is modified to create a 50mS time quantum. Some helper functions have also been added.

EXTERNS:

scheduler.C: EOQTimer * timer, MemPool * MEMORY_POOL

thread.C : MemPool * MEMORY_POOL, Scheduler* SYSTEM_SCHEDULER

TESTING:

Uncommented the _USES_SCHEDULER & _TERMINATING_FUCNTIONS macros to test the code.

GENERIC INSTRUCTIONS:

This code has been developed on the new code environment.
The TA is expected to unzip the file and the run *make.* If *make clean* command is used, make sure to copy paste.

CODE MAINTENANCE:

The entire source code base has been pushed to the student's github repository. This have been done progressively over the course of development. This should help the TA in grading the assignment. Apart from the code base provided in the zip file submitted on canvas, the code can also be compiled from the github repo code base. solution to make the code complete.