MP5: Kernel-Level Thread Scheduling

Chonglin Zhang
UIN: 833003072
CSCE611: Operating System

Assigned Task

The objective of this machine problem is to incorporate the scheduling of multiple kernel-level threads through a series of steps:

1. Implement a non-terminating FIFO scheduler.
2. Adapt the scheduler to allow for thread termination.
3. OPTION 1: Manage interrupt (enabling or disabling).

NOTE: Some changing on kernel file which will be mention below.

FIFO Scheduler: Non-Terminating Threads

Design

I utilized a list with two pointers for the FIFO scheduler's ready queue; one points to the list's head and the other to the tail. The scheduler removes the head thread upon the thread's yield() call and sends it to the dispatcher for context switching. The thread re-enters the queue when resume() is invoked once it's ready.

Implementation

The FIFO scheduler uses a ready queue with pointers to the first and last nodes to organize and dispatch threads sequentially and includes several functions.

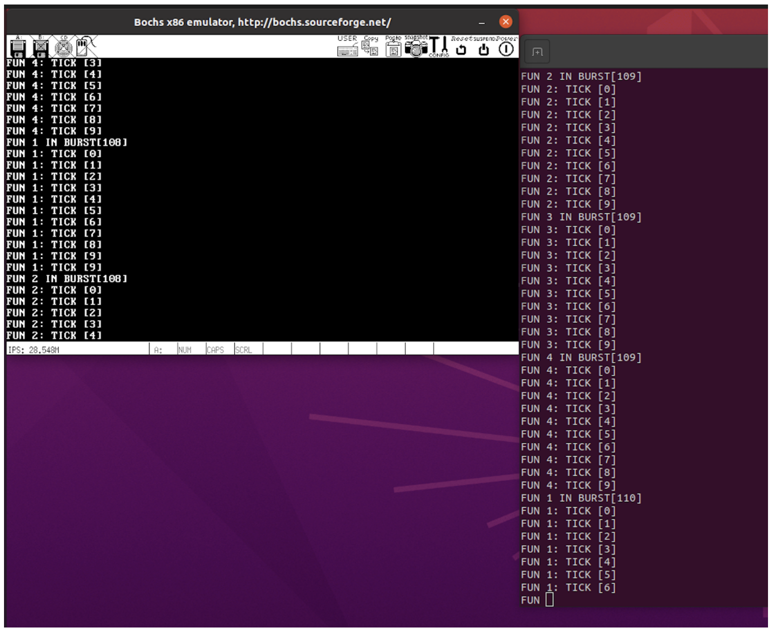
scheduler.c: yield(): The function dispatches the queue's first thread for execution and then updates the head pointer to the next thread.

```
void Scheduler::yield() {

    ThreadNode *tempQueue = head;
    if (tempQueue) {
        Thread *schedThread = tempQueue->thread;
        head = tempQueue->next;
        tail = (!head) ? nullptr : tail;

        delete tempQueue;
        Thread::dispatch_to(schedThread);
    }
  //assert(false);
}
```

scheduler.c: resume(): When a thread is ready to resume, it's placed at the list's end, and the tail pointer is adjusted to this thread.

```
void Scheduler::resume(Thread * _thread) {

    ThreadNode *newEntry = new ThreadNode;
    newEntry->thread = _thread;
    newEntry->next = nullptr;

    if (!head) {
        head = newEntry;
    } else {
        tail->next = newEntry;
    }
    tail = newEntry;
  //assert(false);
}
```

Testing: To test the FIFO implementation, I employed the continuous non-terminating thread loops, func1 and func4, from Kernel.C, facilitating seamless CPU handoff between threads in a FIFO sequence after each burst. I enabled the FIFO scheduler by uncommenting the USE_SCHEDULER macro in the kernel file, thereby invoking Scheduler class functions for thread management. The result figure below:



```
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[108]
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 1: TICK [9]
FUN 2 IN BURST[108]
FUN 2: TICK [0]
FUN 2: TICK [1]
FUN 2: TICK [2]
FUN 2: TICK [3]
FUN 2: TICK [4]
```

```
FUN 2 IN BURST[109]
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[109]
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[109]
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[110]
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
```

FIFO Scheduler: Terminating Threads

We enhanced the FIFO scheduler to handle thread termination by removing the thread from the queue, deallocating its memory, and then yielding to the next thread.

Implementation

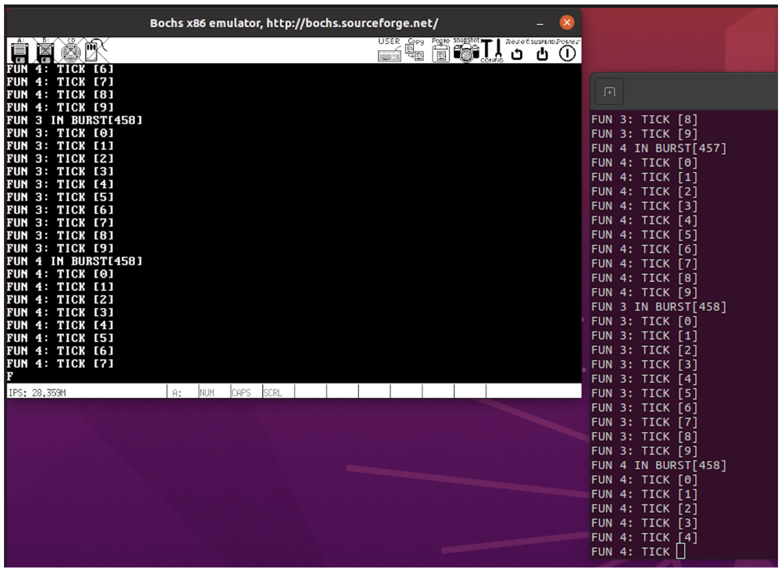
thread.c: thread_shutdown(): This function is invoked upon the thread's exit from its function, terminating the thread and freeing its memory and resources.

```
static void thread_shutdown() {
    /* This function should be called when the thread returns from the thread
        function.
       It terminates the thread by releasing memory and any other resources held by
          the thread.
       This is a bit complicated because the thread termination interacts with the
          scheduler.
     */
    Machine::disable_interrupts();
    if (SYSTEM_SCHEDULER) {
        SYSTEM_SCHEDULER->terminate(current_thread);
    }
    if (current_thread) {
        delete current_thread;
        current_thread = nullptr;
    }
    SYSTEM_SCHEDULER->yield();
    //assert(false);
    /* Let's not worry about it for now.
       This means that we should have non-terminating thread functions.
    */
}
```

scheduler.c: terminate():This action eliminates the Thread from the ready queue, deallocates its memory, and then executes yield() to transition to the next Thread in the sequence.

```
void Scheduler::terminate(Thread * _thread) {
    delete _thread;
    yield ();
  //assert(false);
}
```

Testing:
I activated on the TERMINATING_THREADS macro, allowing only threads 3 and 4 to engage in context switching by terminating threads 1 and 2. The result figure below:



```
FUN 4: TICK [6]
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
FUN 3 IN BURST[458]
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[458]
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]
F
```

```
FUN 3: TICK [8]
FUN 3: TICK [9]
FUN 4 IN BURST[457]
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 3 IN BURST[458]
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[458]
FUN 4: TICK [0]
FUN 4: TICK [1]
FUN 4: TICK [2]
FUN 4: TICK [3]
FUN 4: TICK [4]
FUN 4: TICK
```

OPTION 1: Interrupt Handling
I updated the scheduler.c and thread.c files to control interrupt settings, enabling interrupts at thread startup and disabling them during queuing to prevent the time quantum interrupt from interfering with the process. We re-enable interrupts after completing queue operations.

Implementation

thread.c: thread_start():Activate interrupts at each thread's initiation.

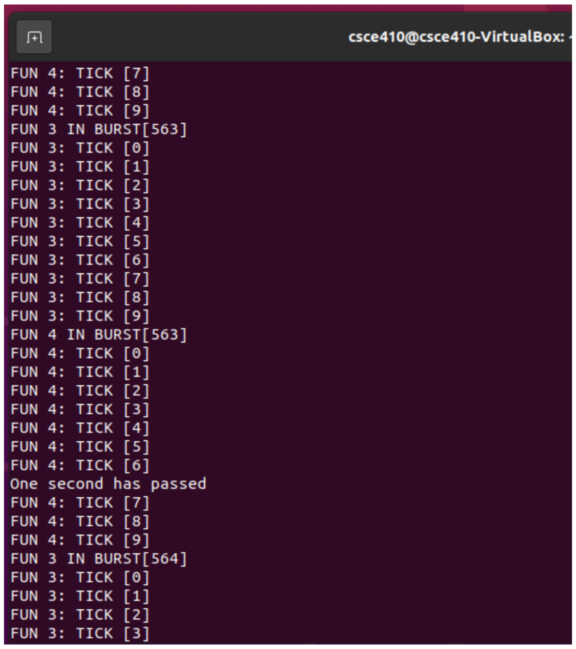
```
static void thread_start() {
     /* This function is used to release the thread for execution in the ready
         queue. */

     /* We need to add code, but it is probably nothing more than enabling
         interrupts. */
    Machine::enable_interrupts();
}
```

scheduler.c: yield, resume : For this two functions/methods, Interrupts are disabled during these functions to protect queuing operations, and re-enabled upon their completion.

```
void Scheduler::yield() {

    bool areInterruptsEnabled = Machine::interrupts_enabled();
    if (areInterruptsEnabled) {
        Machine::disable_interrupts();
    }

    ThreadNode *tempQueue = head;
    if (tempQueue) {
        Thread *schedThread = tempQueue->thread;
        head = tempQueue->next;
        tail = (!head) ? nullptr : tail;

        delete tempQueue;
        Thread::dispatch_to(schedThread);
    }

    if (!areInterruptsEnabled) {
        Machine::enable_interrupts();
    }
  //assert(false);
}

void Scheduler::resume(Thread * _thread) {

    bool areInterruptsEnabled = Machine::interrupts_enabled();
    if (areInterruptsEnabled) {
        Machine::disable_interrupts();
    }

    ThreadNode *newEntry = new ThreadNode;
    newEntry->thread = _thread;
    newEntry->next = nullptr;

    if (!head) {
        head = newEntry;
    } else {
        tail->next = newEntry;
    }
    tail = newEntry;

    if (!areInterruptsEnabled) {
        Machine::enable_interrupts();
    }
  //assert(false);
}
```

Testing:

Managing interrupts on and off guarantees the display of the "One second has passed" message in the console. The result figure below:



csce410@csce410-VirtualBox:

```
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
FUN 3 IN BURST[563]
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[563]
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]
One second has passed
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
FUN 3 IN BURST[564]
FUN 3: TICK [0]
FUN 3: TICK [1]
FUN 3: TICK [2]
FUN 3: TICK [3]
```