MP5: Kernel-Level Thread Scheduling

Ajay Jagannath
UIN: 236002822
CSCE611: Operating System

Assigned Tasks

Part I: FIFO Scheduler - Completed.
Part II: Support for Terminating Threads - Completed.
Bonus Option 1: Correct Interrupt Handling - Completed.
Bonus Option 2: Round-Robin Scheduling - Completed.
Bonus Option 3: Processes - Not Attempted.

Files modified: 'scheduler.H', 'scheduler.C', 'thread.H', 'thread.C', 'simple_timer.H', 'simple_timer.C', 'interupt.H', 'interupt.C', 'kernel.C'(for testing only)

System Design

The goal of this machine problem is to implement a kernel-level thread scheduler, moving away from explicit dispatching. This involves creating a 'Scheduler' class to manage a ready queue, handling thread termination gracefully, and implementing interrupt-driven preemption for a Round-Robin (RR) scheduler.

Part I: FIFO Scheduler

The core task is to implement a First-In-First-Out (FIFO) scheduler. This is achieved by creating a 'Scheduler' class that maintains a ready queue of threads.

- **Ready Queue:** A doubly linked list is implemented using a private 'struct ReadyThread'. The 'Scheduler' class maintains 'head' and 'tail' pointers to manage this queue.
- **'resume(Thread* _thread)':** This function adds a thread to the scheduler. It creates a new 'ReadyThread' node and appends it to the **tail** of the ready queue, ensuring FIFO behavior.
- **'add(Thread* _thread)':** This function is a simple wrapper that calls 'resume()' to make a newly created thread runnable.
- **'yield()':** This function is called by a thread to voluntarily give up the CPU. It removes the next thread to run from the **head** of the ready queue and calls 'Thread::dispatch_to()' to context-switch to it.

Part II: Terminating Threads

To support threads whose functions return, a "zombie" thread mechanism is implemented. This is necessary because a thread cannot deallocate its own stack while it is still running on that stack.

- **'thread_shutdown()':** This function (in 'thread.C') is pushed onto a thread's stack at creation. When the thread function returns, it calls 'thread_shutdown()'.
- This function first calls 'SYSTEM_SCHEDULER->terminate(Thread::CurrentThread())' to mark itself as a zombie, and then calls 'SYSTEM_SCHEDULER->yield()' to stop executing.

- **‘Scheduler::terminate(Thread* _thread)‘:** This function manages the cleanup. It maintains a ‘zombieThread‘ pointer. When called, it first checks if ‘zombieThread‘ is non-null (meaning a *previous* thread is waiting for cleanup). If so, it calls ‘delete zombieThread‘, which invokes the new ‘Thread:: Thread()‘ destructor to free the old zombie’s resources. It then sets the current ‘_thread‘ as the new ‘zombieThread‘, to be cleaned up on the *next* call to ‘terminate‘.
- **‘Thread:: Thread()‘:** A destructor is added to the ‘Thread‘ class to ‘delete[]‘ the thread’s ‘stack‘ and any other allocated resources, ensuring no memory leaks.

Bonus Option 1: Correct Interrupt Handling

The base system runs with interrupts disabled. This bonus option re-enables them and ensures the scheduler is thread-safe.

- **Enabling Interrupts:** The ‘thread_start()‘ function, which runs when a thread is first dispatched, is modified to call ‘Machine::enable_interrupts()‘. This ensures that after the first context switch, interrupts are active.
- **Mutual Exclusion:** The scheduler’s ready queue is a critical-section. The ‘Scheduler::yield()‘ and ‘Scheduler::resume()‘ functions are modified to disable interrupts (‘Machine::disable_interrupts()‘) at the beginning and re-enable them (‘Machine::enable_interrupts()‘) at the end. This prevents a timer interrupt from firing in the middle of a queue operation, which would corrupt the linked list.

Bonus Option 2: Round-Robin Scheduling

This bonus implements a preemptive Round-Robin (RR) scheduler by subclassing ‘Scheduler‘ and using a timer interrupt.

- **‘EOQTimer‘:** A new class ‘EOQTimer‘ is derived from ‘SimpleTimer‘. It’s initialized in ‘kernel.C‘ with a 1000Hz frequency (1ms tick) and a quantum value of 5. This results in a 5ms time quantum.
- **‘EOQTimer::handle_interrupt()‘:** This handler is called every 1ms. It increments a static ‘ticks‘ counter. When ‘ticks‘ reaches the quantum (5), it preempts the current thread by:
  1. Calling ‘SYSTEM_SCHEDULER->resume(Thread::CurrentThread())‘ to put the running thread at the back of the ready queue.
  2. Sending an End-of-Interrupt (‘set_EOI‘) to the PIC.
  3. Calling ‘SYSTEM_SCHEDULER->yield()‘ to switch to the next thread at the head of the queue.
- **‘RRScheduler‘:** A new class ‘RRScheduler‘ is derived from ‘Scheduler‘.
- **‘RRScheduler::yield()‘:** This function overrides the base ‘yield()‘. It first calls ‘EOQTimer::reset_ticks()‘ to reset the quantum timer. This is crucial for voluntary yields, ensuring the *next* thread gets a full 5ms time slice and isn’t penalized by the time remaining from the yielding thread. It then calls the base ‘Scheduler::yield()‘ to perform the context switch.
- **InterruptHandler::set_EOI:** A public function in InterruptHandler to send an End-of-Interrupt (EOI) message to the PIC controller.

Code Description

Scheduler (scheduler.H / scheduler.C)

Data Members (scheduler.H) : The ‘Scheduler‘ class is modified to include a private ‘ReadyThread‘ struct for a doubly linked list. It also adds pointers for the queue (‘head‘, ‘tail‘), the currently running thread (‘currThread‘), and the thread pending deletion (‘zombieThread‘). The ‘RRScheduler‘ subclass is also defined.

```
class Scheduler {

private:
    // Ready Queue with doubly linked list
    struct ReadyThread
    {
        Thread *thread;
        ReadyThread *next;
        ReadyThread *prev;
        ReadyThread():
            thread(nullptr), next(nullptr), prev(nullptr) {}
        ReadyThread(Thread *x) :
            thread(x), next(nullptr), prev(nullptr) {}
        ReadyThread(Thread *x, ReadyThread *next, ReadyThread *prev):
            thread(x), next(next), prev(prev) {}
    };

    // keep track of the top and bottom of queue
    ReadyThread *head, *tail;

    // keep track of currently running thread
    Thread *currThread;

    // zombie thread pointer
    Thread *zombieThread;

public:
    Scheduler();
    virtual void yield();
    virtual void resume(Thread _thread);
    virtual void add(Thread *_thread);
    virtual void terminate(Thread *_thread);
};

class RRScheduler: public Scheduler
{
    /* The RR scheduler fllows the same structure as the FIFO scheduler,
    only change to be done is to reset the EOQ timer in 'yield'. */
public:
    RRScheduler(): Scheduler(){};
    virtual void yield();
};
```

Scheduler::Scheduler (scheduler.C) : Constructor to initialize the data members.

```
Scheduler::Scheduler()
{
  // initially queue is empty.
  head = nullptr;
  tail = nullptr;
  // Use the static function of Thread to get the current thread
  currThread = Thread::CurrentThread();
  zombieThread = nullptr;
  // assert(false);
  Console::puts("Constructed Scheduler.\n");
}
```

Scheduler::resume (scheduler.C) : Implements the FIFO queue-add operation. It disables interrupts to protect the linked list, creates a new 'ReadyThread' node, and appends it to the 'tail' of the list.

```
void Scheduler::resume(Thread *_thread)
{
    // Disable interrupts when adding to ready queue
    if (Machine::interrupts_enabled())
        Machine::disable_interrupts();

    // Add the thread as the tail to the linked list
    ReadyThread *node = new ReadyThread(_thread);

    // Queue is empty
    if (tail == nullptr)
    {
        tail = node;
        head = node;
    }
    else
    {
        tail->next = node;
        node->prev = tail;
        tail = node;
    }

    //enable interrupts after adding to ready queue
    if (!Machine::interrupts_enabled())
        Machine::enable_interrupts();
}
```

Scheduler::add (scheduler.C) : Implements the FIFO queue-add operation for new threads by calling the resume function

```
void Scheduler::add(Thread *_thread)
{
  // Since resume already has the mechanism to add a thread to end of queue, just using that
  resume(_thread);
  // assert(false);
}
```

Scheduler::yield (scheduler.C) : Implements the FIFO queue-remove operation. It pops the 'head' element from the ready queue, dispatches to it, and then deletes the old node. Interrupts are disabled during the queue manipulation.

```
void Scheduler::yield()
{
    //disable interrupts when yielding the CPU (context switch)
    if (Machine::interrupts_enabled())
        Machine::disable_interrupts();

    currThread = Thread::CurrentThread();
    assert(currThread != nullptr && "running thread cant be null and yield\n");
    if (head != nullptr)
    {
        // remove the top element from running queue
        ReadyThread *node = head;
        // if only one element
```

```
        if (head == tail)
        {
            head = nullptr;
            tail = nullptr;
        }
        else
        {
            // make the second element the first element
            head->next->prev = nullptr;
            head = head->next;
        }

        //enable interrupts after context switching
        if (!Machine::interrupts_enabled())
            Machine::enable_interrupts();

        Thread::dispatch_to(node->thread);

        //delete node
        delete node;

        // update running thread
        currThread = Thread::CurrentThread();
    }
}
```

Scheduler::terminate (scheduler.C) : Implements the zombie cleanup. It deletes the previously stored ‘zombieThread‘ (if one exists) and then stores the new thread (that just terminated) as the *next* zombie to be cleaned up.

```
void Scheduler::terminate(Thread *_thread)
{
    //delete the prior zombie thread if exists
    if (zombieThread != nullptr)
    {
        delete zombieThread; // Custom Destructor in thread. Handles stack cleanup
        zombieThread = nullptr;
    }
    // set the current thread as zombie thread for later deletion
    zombieThread = _thread;
}
```

Thread Management (thread.C)

thread_shutdown (thread.C) : This function is the new entrypoint for when a thread function returns. It notifies the scheduler to ‘terminate‘ the current thread and then ‘yield‘s the CPU, effectively ending the thread’s execution.

```
static void thread_shutdown() {
    /* This function should be called when the thread returns from
    the thread function. */

    // terminate prior zombie thread (only one can exist at maximum)
    SYSTEM_SCHEDULER->terminate(Thread::CurrentThread());

    // yield to next thread
    SYSTEM_SCHEDULER->yield();
```

```
    // assert(false); /* We should never reach this point. */
}
```

Thread:: Thread (Destructor) (thread.C) : A new destructor is added to free the thread's stack and other resources. This is called by 'Scheduler::terminate' to clean up a zombie.

```
//custom destructor to free memory
Thread::~Thread() {
    if (stack) delete[] stack; // Free the stack memory
    if (cargo) delete[] cargo; // Free cargo if allocated
    if (esp) delete esp; // Free esp if allocated
}
```

thread_start (thread.C) : Modified to enable interrupts, kicking off interrupt handling (and thus the RR timer, if used) once the first thread starts.

```
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. */
    if (!Machine::interrupts_enabled())
        Machine::enable_interrupts();
}
```

Round-Robin Scheduler

InterruptHandler::set_EOI (interrupt.H/interrupt.C) : Defines a public function to send an End-of-Interrupt (EOI) message to the slave controller.

```
void InterruptHandler::set_EOI(REGS * _r) {
  /* Check if the interrupt was generated by the slave interrupt controller.
       If so, to send an End-of-Interrupt (EOI) message to the slave controller. */

  // This function needs the interrupt number to determine if it was generated by slave PIC
  unsigned int int_no = _r->int_no - IRQ_BASE;

  if (generated_by_slave_PIC(int_no)) {
    Machine::outportb(0xA0, 0x20);
  }

  // Send an EOI message to the master interrupt controller.
  Machine::outportb(0x20, 0x20);
}
```

EOQTimer Class (simple_timer.H) : Defines the 'EOQTimer' class, inheriting from 'SimpleTimer'. It adds a static 'ticks' counter and a 'quantum' value. The constructor calculates the quantum in ticks from a given millisecond value.

```
class EOQTimer: public SimpleTimer
{
private:
    //count of ticks
    static int ticks;

    //the total time quantum in ticks
```

```
    int quantum;

public:
    EOQTimer(int _hz, int _quantum) :
        SimpleTimer(_hz), quantum((_quantum * _hz) / 1000) {}

    virtual void handle_interrupt(REGS *_r);
    static void reset_ticks();
};
```

EOQTimer::handle_interrupt (simple_timer.C) : The core of the RR scheduler. It's called every 1ms (1000Hz). It counts 5 ticks (for a 5ms quantum) and then forces a context switch by resuming the current thread and yielding to the next.

```
extern Scheduler *SYSTEM_SCHEDULER;

int EOQTimer::ticks = 0;

void EOQTimer::reset_ticks()
{
    EOQTimer::ticks = 0;
}

void EOQTimer::handle_interrupt(REGS *_r)
{
    /* Call the base class handler to maintain time */

    // Interrupt triggered every 1ms, we need to log when ticks reach 5 (5ms)
    // Dont start timer without any threads running
    if (Thread::CurrentThread() != nullptr)
        EOQTimer::ticks++;

    if (EOQTimer::ticks == quantum)
    {
        Console::puts("5ms has passed\n"); // Note: My comment said 50, but code is 5ms
        reset_ticks();

        // Get the next thread from the scheduler and perform context switch
        if (Thread::CurrentThread() != nullptr)
        {
            SYSTEM_SCHEDULER->resume(Thread::CurrentThread());
            InterruptHandler::set_EOI(_r);
            // reset ticks
            SYSTEM_SCHEDULER->yield();
        }
    }
}
```

RRScheduler::yield (scheduler.C) : The overridden 'yield' for voluntary-yields. It resets the timer before calling the base 'yield'.

```
void RRScheduler::yield()
{
    // Reset the EOQ timer
    EOQTimer::reset_ticks();

    // Call base class yield to perform context switch
```

```
    this->Scheduler::yield();
}
```

Testing

Testing was performed by modifying the macros in ‘kernel.C‘.

- **_USES_SCHEDULER_**: This macro was defined to enable the scheduler. This activates the ‘else‘ block in ‘pass_on_CPU‘, redirecting all context switches through the ‘Scheduler::yield()‘ and ‘Scheduler::resume()‘ functions. All the four threads will be non terminating.
- **_TERMINATING_FUNCTIONS_**: This macro was defined to test the ”zombie” thread mechanism. It changes the loops in ‘fun1()‘ and ‘fun2()‘ from infinite loops to loops that run 10 times and then return. This triggers the ‘thread_shutdown()‘ function and tests the ‘Scheduler::terminate()‘ logic, including the deferred deletion of thread stacks.
- **_RR_SCHEDULER_**: This macro was defined to test the Round-Robin scheduler. In ‘main()‘, this macro causes an ‘RRScheduler‘ and ‘EOQTimer‘ to be instantiated instead of the default ‘Scheduler‘ and ‘SimpleTimer‘.

With just the _USES_SCHEDULER_ defined, we get 4 non-terminating functions, as evidenced with no preemption, as seen that thread 1 and 2 go beyond 10 bursts.

```
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[53]
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[53]
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[54]
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[54]
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[54]
FUN 3: TICK [0]
FUN 3: TICK [1]
FUN 3: TICK [2]
```

Figure 1: Non-terminating functions

With the _TERMINATING_FUNCTIONS_ also defined, thread 1 and 2 terminate safely after 10 bursts. This is seen when after the 10th burst (index 9) of thread 4, the 11th burst (index 10) of thred 3 starts, instead of thread 1.

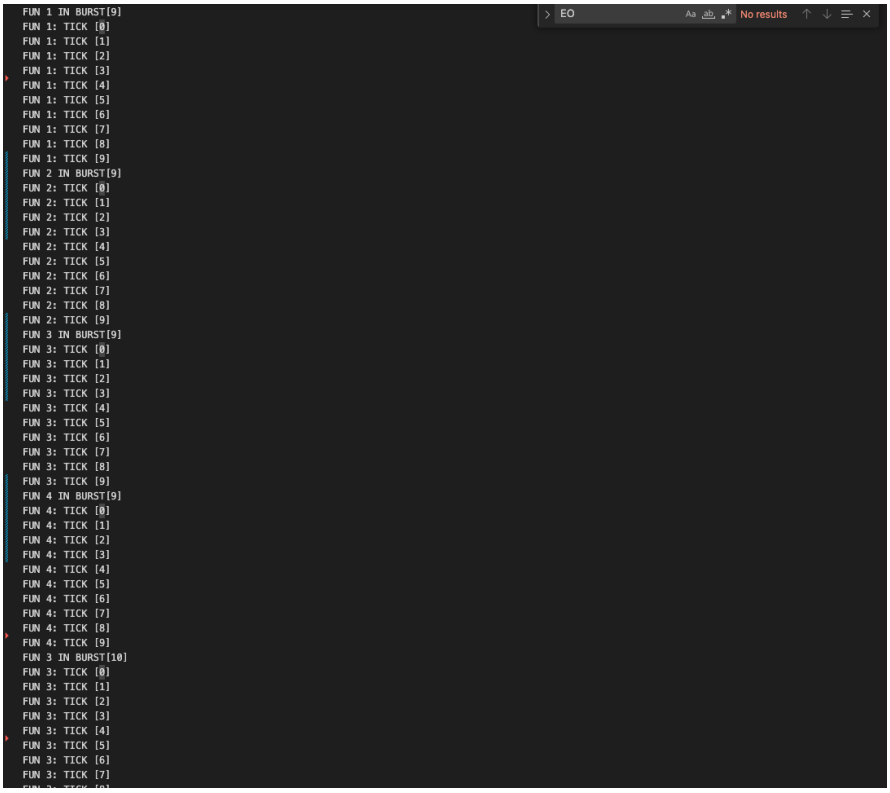

Figure 2: Terminating functions

Bonus Option 1: With interrupts enabled, we can see that every one second, there is an interrupt which is registered through a console print.

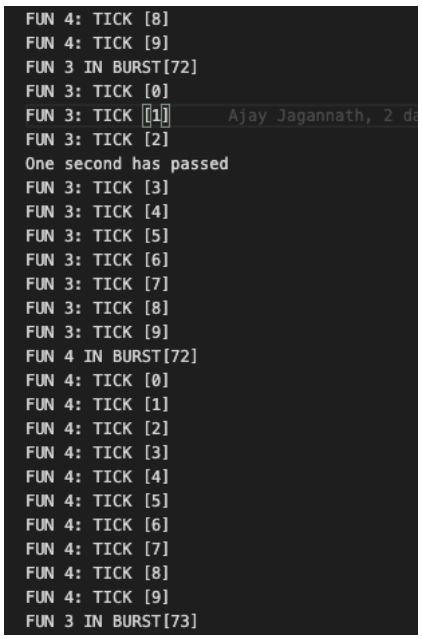

Figure 3: Interrupts Enabled

Bonus Option 2 With all three macros enabled (including newly defined **_RR_SCHEDULER_**), the resulting behavior is a preemptive, round-robin system. The output shows 'fun1' and 'fun2' running

and being preempted (as evidenced by the "5ms has passed" messages from the 'EOQTimer' handler) until they complete their 10 iterations and terminate. The system then continues to run, scheduling the infinite loops of 'fun3' and 'fun4', which are continuously preempted by the 5ms timer, demonstrating that all three components (FIFO queue, thread termination, and RR preemption) are working correctly. Here, voluntary yielding resets timer to maintain time quantum per job (5ms).

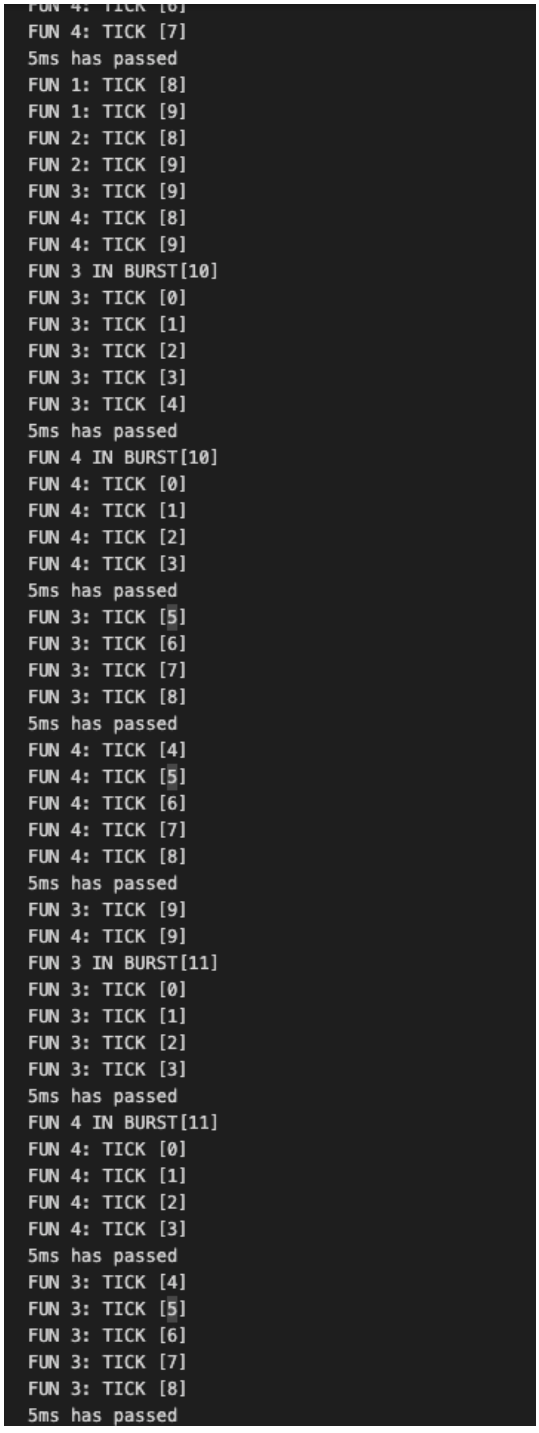

Figure 4: RR Scheduler