Design Report

This is the implementation details for Machine Problem 5 for thread management and schedule.

Here I have implemented both FIFO and Round-Robin in the `schedule.C` class. To use the FIFO scheduler, you just need to follow the handout instructions, as for the Round-Robin scheduler, you have to make some modifications with the following steps because I added some methods in Scheduler class and Simple Timer class.

1. Comment out the `pass_on_CPU(thread3)` function in all functions, it is not required here because timer will switch threads.
2. Add new commands after the `SYSTEM_SCHEDULER = new Scheduler()`, so now the scheduler definition becomes:

```
#ifdef _USES_SCHEDULER_
/* -- SCHEDULER -- IF YOU HAVE ONE -- */
SYSTEM_SCHEDULER = new Scheduler();
SYSTEM_SCHEDULER->set_round_robin();
SYSTEM_SCHEDULER->addTimer(&timer);
#endif
```

Implementation Details

In this implementation, I have made changes to the following files: `scheduler.H, scheduler.C, simple_timer.H, simple_timer.C, thread.H, thread.C`.

Scheduler class

In order to make the scheduler first-in and first-out, I have defined a queue class inside the scheduler. The queue class is able to push, pop and remove threads.

```
struct Node {
    Thread* thread;
    Node* next;
};
class Queue {
 private:
    Node *head;
    int _size;
 public:
    Queue();
    void enqueue(Thread *t);
    Thread* dequeue();
    void remove(Thread * thread);
    bool isEmpty() { return _size == 0;}
    int size() {return _size;}
};
```

Now the scheduler operation becomes very easy, we can simply use the queue class to operate schedule add, schedule resume, schedule terminate.

```
Scheduler::Scheduler() {
  ready_queue = Queue();
  ...
}
void Scheduler::yield() {
  if (ready_queue.isEmpty()) return;
  Thread* thread = ready_queue.dequeue();
  ....
  Thread::dispatch_to(thread);
}
void Scheduler::resume(Thread * _thread) {
  ...
  ready_queue.enqueue(_thread);
}
void Scheduler::add(Thread * _thread) {
  ready_queue.enqueue(_thread);
}
void Scheduler::terminate(Thread * _thread) {
   ...
   ready_queue.remove(_thread);
}
```

Handling Interrupts and Thread Shutdown

To shutdown of thread, the below changes need to be made to the file `thread.C`, here I defined an extern variable `extern SYSTEM_SCHEDULER` to perform operation on the thread shutdown by scheduler and delete thread if thread shutdown is called.

```
static void thread_shutdown() {
    SYSTEM_SCHEDULER->terminate(current_thread);
    delete current_thread;
    SYSTEM_SCHEDULER->yield();
}
```

Now, to handle interrupts, I first need to enabled the interrupts in the `thread.C` file and then disable or enable interrupts in the scheduler files.

```
static void thread_start() {
     Machine::enable_interrupts();
}
```

Now in the scheduler files, I also need to make modification to make sure mutual exclusion in some critical operation, so I disable the interrupt in thread yield, and enable interrupt in thread resume, and now time interrupt works fine.

```
void Scheduler::yield() {
....
  if (Machine::interrupts_enabled())
    Machine::disable_interrupts();
....
}
void Scheduler::resume(Thread * _thread) {
....
  if (!Machine::interrupts_enabled())
    Machine::enable_interrupts();
...
}
```

Round Robin

The round robin scheduler can be implemented with the same FIFO class, and some modifications needs to be made to the Simple Timer class. In the Simple Timer class, every 10ms the ticks will trigger timer interrupt, so we can handle the 50ms time quantum here whenever the time quantum expires, the scheduler will execute resume and yield command. So, first we define an extern variable in the Simpler timer class. Here, we need also to tell FIFO scheduler that we are going to use Round Robin now and we need a time interrupt handler function too, so I defined a Simple Timer object and a round robin enabling variable in FIFO:

```
Scheduler::Scheduler() {
  ready_queue = Queue();
  round_robin = false;
  timer = NULL;
```

In the simple timer class, when the time quantum expires, it is handled in the `handle_interrupt` method. Here, I used the `resume` and `yield` methods to give CPU to next thread in the queue. Also, here I need to tell the interrupt controller that the interrupt is over, so I use the `Machine::outportb(0x20, 0x20)` command to tell interrupt controller that the interrupt has been handled, so next interrupt can be triggered. Otherwise, the next interrupt will not be able to be detected.

```
void SimpleTimer::handle_interrupt(REGS *_r)
    ...
   ticks++;
    if (ticks >= 5)
    {
        Machine::outportb(0x20, 0x20);
        ticks = 0;
        SYSTEM_SCHEDULER->resume(Thread::CurrentThread());
        SYSTEM_SCHEDULER->yield();
    }
    ....
}
```

Now, in the Scheduler class, we have to deal with cases when the yield occurs before the time quantum expires. To make this one happen, we can simply reset the ticks in the Simple Timer object to 0, so ticks start from 0 again and next thread will not be penalized due to the unfinished quantum from previous thread. So we can do this to reset ticks in yield method:

```
void Scheduler::yield() {
...
  if (round_robin && timer)
        timer->reset_Ticks();
...
}
```

Now, if the yield method is called outside of the interrupt handling method, it will also reset the ticks to zero to make counter count time from zero again.

In order to make it work in the main test files, we have to modify the scheduler creation method as:

```
SYSTEM_SCHEDULER = new Scheduler();
SYSTEM_SCHEDULER->set_round_robin();
SYSTEM_SCHEDULER->addTimer(&timer);
```

In the above commands, the round robin variable is set to be true, so the Simple Timer class knows to handle CPU transfer whenever 50ms quantum reaches. And the scheduler class calls timer object to reset ticks whenever yield occurs outside of the quantum interrupt handler, namely, yield occurs earlier than quantum reaches.

The above is the implementation details for this assignment.