CSCE 613: Project 5 Design

Student: Caio Duarte Diniz Monteiro

Date: 04/5/16

Scheduler class

```
class Scheduler {

   /* The scheduler may need private members... */
    int queue_size; //store how many elements are on the queue;
    TaskNode* queue; //store the queue
    void enqueue(Thread* _thread);
```

The Scheduler class is responsible to manage the tasks allocation on the CPU. There are two scheduling policies implemented, first in first out (FIFO), and Round Robin. The desired policy is chosen by manipulating the defined value **SCHE_ROUND_ROBIN**, when it is defined, round robin is used, otherwise it uses FIFO.

To maintain the queue of tasks, a list of TaskNode is used. Each task node contains the thread it represents and a pointer to the next element on the queue.

```
struct TaskNode{ //data structure to hold the queue of tasks ready
    Thread *thread;
    TaskNode *next;
    TaskNode(Thread* t) : thread(t){}
};
```

Yield function

```
virtual void yield();
```

Yield function is responsible to dispatch the execution to the first task on the queue. It will remove the task from the ready queue, update the queue size and dispatch it to execution.

Resume and Add functions

```
virtual void resume(Thread * _thread); virtual void add(Thread * _thread);
```

For this assignment, both implementations are the same, which is simply pass the thread pointer received as a parameter to the enqueue function. The enqueue function is responsible to add the thread to the end of the ready queue and increase the queue size.

Enqueue function

```
void enqueue(Thread* _thread);
```

Enqueue function adds the thread passed as a parameter on the end of the ready tasks queue. It creates a TaskNode with the thread, and set the next pointer of the last element on the list to point to it. It will also increment the queue_size variable.

Terminate function

```
virtual void terminate(Thread * _thread);
```

The terminate function is responsible to free the resources used by a finished thread and yielding the execution to the next ready thread on the queue.

Interrupts class

Interrupt class had to change its dispatch_interrupt function to deal with the round robin scheduling. When using round robin scheduling the timer will trigger a context switch through an interrupt, so we need to let the PIC know that the interrupt has been taken care of before switching contexts on the handle_interrupt function.

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

Thread class

Thread class maintains the TCP of the existing threads on the system. To allow thread termination, an extern pointer to the system scheduler was included, so, upon thread_shutdown, the system scheduling terminate function is called. To properly work with the round robin scheduling, interruptions were disabled when setting the context of the thread, enabled upon thread start, disabled right before the context switch and enabled right after.

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

SimpleTimer class

Behavior of the SimpleTimer is dependent on the defined variable SCHE_ROUND_ROBIN. If the OS is using round robin, then it needs to perform a context switch every 50ms, otherwise it keeps the previous behavior of simply displaying a message every one second. The changes are made on the handle_interrupt function and include divide the frequency of the timer by 20, to obtain the 50 ms, put the current thread at the end of the scheduling queue, and yield execution to the next available thread.