CSCE6111
MP5
Siru Li
222007338

Design Document

Note: The basic assignment is fully functional. In addition, I implemented bonus option 1 (interrupt) and option 2 (RR scheduling). However, my RR scheduling is only partially functional. So to run FIFO scheduling, simply uncomment //pass_on_CPU(threadX); in kernel.C.

In this machine problem, we implement a scheduler that allows context switches between threads. The bare minimum requires us to implement a simple FIFO scheduler, where each thread preempts themselves explicitly. Later on, as an option, we handle interrupts correctly, and use the system timer to implement a Round-Robin scheduling policy. Lastly, if time permits, we implement Processes, which is much trickier because each process will have their own address space.

1. **FIFO Scheduler**

   In order to implement the FIFO scheduler, we need to support basic scheduling functions, mainly yield() and terminate(). In addition, we will need to setup a FIFO queue. I decided to use a Ring-Buffer to implement the queue, and let the queue sit on the stack, since the queue only holds a bunch of pointers to threads, therefore, it won't be big. The code for Ring-Buffer is included on the next page.

   Next, we implement the void Scheduler::yield() function, which pops the next waiting thread from the queue and dispatch to that thread. Void Scheduler::resume(Thread* _thread) adds a thread to the queue, quite straight forward. Void Scheduler::add(Thread* _thread) simply calls resume().

   Lastly, we implement the termination function of the thread. This gets a little more complicated. We observe that the pointer to the function thread_shutdown() is placed onto the stack during the context setup of the thread. When the thread is terminating, the stack gets popped correspondingly. During the process, thread_shutdown() gets called. In order for a graceful shutdown of a thread, we need to yield the CPU to the next waiting thread. Clean up the memory for this thread object.

   We do not have access to the scheduler object in thread.C. The solution I chose is to use the "extern" keyword, which specifies external linkage. Since the Scheduler * SYSTEM_SCHEDULER; object is defined in main, the linker will figure out the scheduler object through the "extern" keyword.

```
class FIFOQueue{
    private:
        Thread* buffer[100]; //statically allocated pointers to thread
objects
        int head;
        int tail;
        int size;
    public:
        FIFOQueue(){
            head = 0;
            tail = 0;
            size = 0;
        }


        bool push(Thread* t){
            int newTail = (tail + 1) % 100;
            if(newTail == head)
                return false; //full


            buffer[tail] = t;
            tail = newTail;
            size++;
            return true;
        }

        Thread* pop(){
            if(head == tail)
                return 0; //empty


            Thread* item = buffer[head];
            head = (head + 1) % 100;
            size--;
            return item;
        }


        int q_size(){
            return size;
        }
};
```

Fig. 1 Ring-Buffer implementation of FIFO queue

2. **OPTION 1: Correct handling of interrupts.**
   By default, interrupt is disabled. In order to prepare for RR scheduling, we need to re-enable interrupts, so that when the timer fires, we can preempt the current thread in the interrupt handler.

   To do that, we simply enable interrupt in static void thread_start() function. In addition, we need to temporarily disable interrupt when yield() is called, and re-enable interrupt() when yield() returns. This ensures mutual exclusion.

   After the above modifications, interrupts work correctly, and the timer correctly triggers interrupts and prints interrupt message.

3. **OPTION 2: Round-Robin Scheduling.**
   In order to implement Round-Robin scheduling, we need to handle context switches in simple_timer's interrupt handler. To handle context switch this way, we do not return from interrupt. So the PIC does not know whether the interrupt has been successfully handled, and therefore, will not accepted new interrupts. Thus, we need to correctly inform the PIC to allow future interrupts from being triggered correctly.