CPSE 611 – Operating Systems and Applications

**MP4: Design Document**

A description...

Sridhar Mareguddi

UIN: 823000772

Date: 4/08/2014

# **Introduction**

In this machine problem we have implemented a scheduler that schedules multiple kernel-level threads to the code base. The first part of the machine problem includes the code for first-in first-out (FIFO) scheduler using a ready queue. The ready queue consists of a list of threads waiting for the CPU.

The optional 1 part of the Machine Problem implements a interrupt handler code which disables interrupts once a created thread returns from the fake ”exception” when it starts and re-enables interrupts later.

The optional 2 part of the Machine Problem implements a basic Round-Robin Scheduler. The Round-Robin scheduler generates a periodic interrupt at every 50ms that triggers the preemption of the currently running thread and puts the next ready thread from the ready queue onto the CPU.

# **FIFO Scheduler Implementation**

The following section describes the details of the scheduler.H exported functions implemented inside scheduler.C and modifications made to scheduler.H file.

**scheduler.H modifications:**

A Ready\_Queue structure has been defined in the private section of the scheduler class, that contains a pointer to thread, next and prev of the type struct Ready\_Queue as its members. In addition to this , three pointer variables namely first, temp and last have been created of the type Ready\_Queue.

The Ready\_Queue is used to maintain the list of the threads that are ready waiting for the CPU. The Ready\_Queue has been implemented as a first-in first-out (FIFO) doubly linked queue with first pointer pointing to the first thread in the queue and last pointer pointing to the last thread in the queue.

**scheduler constructor in scheduler.C:**

The constructor initializes the next and prev pointers of the first and last Ready\_Queue pointer variables to null.

**Scheduler:: yield()**

The yield function is called by the currently running thread in order to give up the CPU. We retrieve the first thread from the queue, update the first pointer to point to next element in the queue and call the dispatch\_to() to do the context switch.

Error cases such as when the queue is empty or queue has only one thread is also handled here. The function prints the error message for these cases and returns from the yield function without doing the context switch.

**resume(Thread \* \_thread)**

In this function, we add the given thread to the ready queue of the scheduler. This is called

for threads that were waiting for an event to happen, or that have to give up the CPU in response to a preemption.

Currently this function works exactly as the add function described below and hence includes only a call to the add function, to add the thread to the end of the queue.

**add(Thread \* \_thread)**

Add function makes the given thread runnable by the scheduler. This function is typically called after a thread has been created.

The function includes adding the thread to the end of ready queue and updating the last pointer to point to this thread.

**terminate(Thread \* \_thread)**

In this function, we search for the given thread in the Ready\_Queue and remove it from the scheduler in preparation for destruction of the thread. The corresponding pointers are updated.

# **Bonus option 1: Correct handling of interrupts**

In the thread\_start() function of thread.C file, Machine::enable\_interrupts() has been called to enable the interrupts whenever a thread is started.

In the dispatch\_to(Thread \* \_thread) function of thread.C, the following lines have been added o disable interrupts before context switch and re-enabling the interrupt after the context switch

Machine::disable\_interrupts();

threads\_low\_switch\_to(\_thread);

Machine::enable\_interrupts();

Additionally, in the thread\_shutdown() API, the assert(FALSE) has been removed and following lines of code are added to terminate the current thread and delete resources.

SYSTEM\_SCHEDULER->terminate (current\_thread);

delete current\_thread;

SYSTEM\_SCHEDULER->yield();

# **Files included along with this report**

# **scheduler.H**

# **Scheduler.C**

# **Thread.H**

# **Thread.C**

# **Kernerl.C**

# **Note: Option 2 has not been implemented**