

CSCE 611: MP4: DESIGN DOCUMENT
AUTHOR: SRIVIDHYA BALAJI
UIN: 827007169

PROBLEM STATEMENT:

To perform scheduling of multiple Kernel-level threads.

- 1) To implement a FIFO scheduler to schedule non-terminating threads
- 2) To modify the FIFO scheduler mechanism to schedule and terminate the Terminating threads
- 3) [OPTION 1 : BONUS] Disable and Enable Interrupts
- 4) [OPTION 2 : BONUS] To implement a Round Robin Scheduler

PART:1 IMPLEMENTATION OF A FIFO SCHEDULER

PRINCIPLE:

A ready Queue is maintained which contains a list of all the threads that are waiting to execute. When the running thread calls yield(), the scheduler finds the next thread in the queue ,ie dequeues the head thread, and sends it to the dispatcher for context switch. When the thread which transferred control is ready to execute it calls the resume function, which adds the thread back to the queue.

DESIGN:

(i) QUEUE IMPLEMENTATION USING CLASS QUEUE

A class named Queue is created in Scheduler. H. This class implements Queue as a linked list with data as thread object type and a next point which points to the next thread object. It has public functions, enqueue() : which adds the data to the queue, dequeue() : which removes the data from the queue.

List of variables or data structures of Class Queue are :

- 1) Thread *threadObj
- 2) Queue* next

List of functions used in class Queue:

- 1) **void enqueue(Thread* t) :** This function is responsible for adding the thread to the queue. It adds the incoming elements to the last in the queue. It initially iterates through the queue till the last, creates a new queue object and then assigns the incoming thread to the threadObject of the newly created thread.

PSEUDOCODE:

```
void enqueue(Thread *t)
{
    If(!threadObj)
        threadObj = t // if object assigned NULL, assign the incoming thread as thread type
    else
        if(!next)
            next = new Queue(t) // next is NULL, adding element; adding element at last
```

```

        else
            next->enqueue( t ) //otherwise iterating till the last element
    }

```

- 2) **Thread* dequeue():** This function is responsible for removing the threads from the queue. Since both the queue and FIFO scheduler follow a FIFO operation, the thread at the beginning is always removed from the queue and dispatched. If empty queue, it returns NULL, otherwise it maps the first object of the thread pointed to by its next and deletes itself.

PSEUDOCODE:

```

Thread *dequeuer()
{
    If(!threadObj) // denotes empty threadObject, ie queue empty situation
        return NULL;
    if(next) // if there are more than one nodes
    {
        Thread* removeThread = threadObj // first thread to be removed
        threadObj = threadObj->next // the next object is made the head
        /*delete memory allocated*/
        Queue *consumed = next;
        next = next->next
        delete consumed;
        return removeThread; // This would be the thread to be dispatched
    }
    /*if only one node*/
    Thread* temp = threadObj;
    threadObj = NULL;
    return temp;
}

```

(ii) FIFO SCHEDULER IMPLEMENTATION :

The Scheduler class, consists of a ready Queue of class type Queue, which maintains a queue of the incoming threads, and dispatches them in a FIFO fashion.

List of variables in Scheduler(default FIFO) class :

- 1) Queue readQueue
- 2) Int queueSize

List of functions in Scheduler class:

- 1) **void Scheduler :: add(Thread * _thread) :** This function is responsible for add the _thread to the ReadyQueue.

Pseudocode:

```
void Scheduler :: add(Thread *_thread)
{
    readyQueue.enqueue(_thread);
    ++queueSize
}
```

- 2) **void Scheduler :: yield()** : This function is responsible for yielding the threads in the queue. It dequeues the thread at the beginning and then sends that to the thread dispatcher for execution.

Pseudocode:

```
void Scheduler :: yield()
{
    If(queueSize != 0)
    {
        Thread *currentThread = readQueue.dequeue() ; // returns the first object in Q
        Thread :: dispatch_to(currentThread); // sends it to the dispatcher function
    }
}
```

- 3) **void Scheduler :: resume()** : When a thread that is dispatched is ready to be executed again, it is added back to the queue at the end, similar to the add function.

Pseudocode:

```
void Scheduler :: resume(Thread *_thread)
{
    readyQueue.enqueue(_thread);
    ++queueSize
}
```

TESTCASES :

The above FIFO implementation was tested using the non-termination thread loops func1() to func4() in Kernel. C, where the CPU is passed from one thread to another after every burst. This happens continuously in a FIFO fashion using the above implementation.

- 1) Uncomment `_USES_SCHEDULER_` macro in Kernel. C . This would ensure an object of type Scheduler object is created which then invokes the Scheduler class functions for each thread.
- 2) The screenshots of the test can be seen below , where each thread executes for 10 ticks and passes the CPU to the next thread.

```

FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
FUN 1: IN BURST[23]
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[23]
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[23]
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[23]
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[24]
FUN 1: TICK [0]
FUN 1: TICK [1]

```

POSSIBLE IMPROVEMENT : Here dynamic scheduling of thread objects are done in the class Queue. Possible improvement wrt memory would be to create a static HEAD pointer pointing to the beginning of the queue and hence allocate memory statically as pointers wouldn't occupy much of memory while being stored in the stack.

PART 2: IMPLEMENTATION OF FIFO SCHEDULER FOR TERMINATING THREADS :

Terminating threads involves removing the thread from the readyQueue, freeing memory allocated to it. The Thread_shutdown() function is called whenever a thread returns from the thread function. In this we interact with the Scheduler, in order to remove the current Thread from the ReadyQueue and then we remove the memory allotted to this thread and finally call yield, which dispatches the next thread in the Queue.

The following function was modified in Thread.C :

PSUEDOCODE:

```
static void thread_shutdown()
{
    SYSTEM_SCHEDULER->terminate(Thread::CurrentThread())
    delete currentThread
    SYSTEM_SCHEDULER->yield() }
```

New function added in Scheduler.C :

- 1) void terminate(Thread * _thread) :** In this function, the _thread is searched for in the ready Queue using its threadID and once a match is found it is removed from the readyQueue. Here,)since each object is a threadobject, to find the matching thread, a technique where every thread was dequeued , checked for ID and enqueued back if didn't match. This way the matching thread ID thread is removed from the ready Queue.

PSEUDOCODE:

```
void terminate(Thread *_thread)
{
    For I from 0 to QueueSize
        Thread*temp = readyQueue.dequeue()
        If temp.threadID == _thread
            " Thread removed"
        Else
            readyQueue.enqueue(temp)
    }
```

POSSIBLE IMPROVEMENT : Since dequeuing and enqueueing repeatedly might cost an additional overhead, another previous pointer could be maintained. Then the entire queue could be traversed to find the thread and set the previous pointer point to the next pointer of the currently being removed thread.

TESTCASES:

- 1) In Kernel. C the `_TERMINATING_THREADS_` macro was enabled. Based on this thread 1 and 2 were initially terminated and thus there was a context switch happening only between thread 3 and thread 4. Screenshot below in the first figure.
- 2) Now, terminate all three, 1, 2, 3 threads. Only thread 4 context switches continuously. Screenshot below in the second figure.

```
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]
TERMINATE
TERMINATE
FUN 3 IN BURST[10]
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[10]
FUN 4: TICK [0]
FUN 4: TICK [1]
FUN 4: TICK [2]
FUN 4: TICK [3]
FUN 4: TICK [4]
```

```
FUN 4: TICK [6]
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
TERMINATE
TERMINATE
TERMINATE
FUN 4 IN BURST[10]
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 4 IN BURST[11]
FUN 4: TICK [0]
FUN 4: TICK [1]
FUN 4: TICK [2]
FUN 4: TICK [3]
```

PART 3: [BONUS] INTERRUPT HANDLING :

The Scheduler.C and Thread.C were modified in order to appropriately enable and disable the interrupts. It is necessary to enable the interrupts at the start of thread and disable it when there is any ongoing operations in the Queue. As , in case of a scheduler like RR that we would discuss in Part4, if the interrupt is not disabled then, there is a possibility of the Time quantum interrupt to disrupt the enqueue and dequeue operations. So whenever a queue operation is performed, we disable the interrupts and enable it once the queue operation is complete.

Changes made for INTERRUPT HANDLING :

- 1) In Thread.C, in thread_start() , Machine::enable_interrupts(); is used to enable the interrupts at the start of each thread.
- 2) In Scheduler.C at the beginning of each of yield(), add(), resume(), terminate() we include,

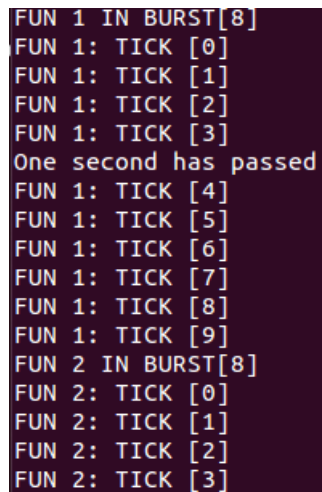
```
if (Machine::interrupts_enabled())  
    Machine::disable_interrupts();
```

and at the end of corresponding Queue operations in each of these functions, we include,

```
if(!Machine::interrupts_enabled())  
    Machine::enable_interrupts();
```

TESTCASES :

Enabling and disabling the interrupts would ensure us seeing the message that “One second has passed” in the console. Screenshot is as below ,



```
FUN 1 IN BURST[8]  
FUN 1: TICK [0]  
FUN 1: TICK [1]  
FUN 1: TICK [2]  
FUN 1: TICK [3]  
One second has passed  
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[8]  
FUN 2: TICK [0]  
FUN 2: TICK [1]  
FUN 2: TICK [2]  
FUN 2: TICK [3]
```

PART 4: [BONUS] ROUND ROBIN SCHEDULING:

In this, we implemented a RR Scheduler based on the functions as defined in PART1 with few additional functions responsible for the time quantum management. In RR Scheduling here, each thread gets the CPU for 50 ms, after which we raise an interrupt, where we add the current thread to the end of the queue and yield the next thread. The RRScheduler, inherits from the Scheduler class and Interrupt Handler class and creates a timer like implementation within it in order to keep track of the time quanta. The add(), resume(), terminate() functions are the same as FIFO Scheduler. The tick frequency is set to 5, which corresponds to 50 ms and the interrupt Handler is registered, all this is done in the constructor.

Class definition for RR Scheduler:

```
class RRScheduler : public Scheduler, public InterruptHandler
{
    Queue readyRRQueue;
    int queueSize;
public:
    /*other functions*/
}
```

Additional functions in RR Scheduler :

- 1) **void RRScheduler :: handle_interrupt(REGS *_r)** : In this function, we check if the 50 ms is reached. If so, we reset the ticks , resume the current Thread by adding it to the end of the Queue and then yields the next thread.

PSEUDOCODE:

```
void RRScheduler :: handle_interrupt(REGS *_r)
{
    If ticks >= hz
        ticks = 0
        resume(current_thread)
        yield()
}
```

- 2) **[MODIFICATIONS] void yield()** : The yield is modified to inform the master interrupt controller that the interrupt is handled at the start of the function . This ensures, that the 50 ms is given to each thread and the threads execute in a RR fashion. It is also important to reset the ticks count.

Machine::outportb(0x20, 0x20); //informing that interrupt is handled by sending EOI message

The above line is added in the yield function ensuring proper RR scheduling.

TESTCASES:

- 1) A macro called `_USES_RRSCHEDULER_` was defined and enabled whenever RRScheduler is used. Object was created corresponding to it and used to execute the threads. Further the `Pass_on_CPU` was also commented out.
- 2) **NON-TERMINATING THREADS** : In this case, each thread from thread 1 to thread 4 context switch for every 50 ms. Screenshot is as below. We can see that once 50ms is passed, thread 2 is pre-empted and thread 3 which previously was pre-empted at `TICK[2]` continues its execution from `TICK[3]`

```
FUN 2: TICK [6]
FUN 2: TICK [7]
FUN 2: TICK [8]
FUN 2: TICK [9]
FUN 2 IN BURST[13]
FUN 2: TICK [0]
FUN 2: TICK [1]
FUN 2: TICK [2]
FUN 2: TICK [3]
50 ms second has passed
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]
```

- 3) **TERMINATING THREADS**: In this case, thread 1 and 2 were terminated and thread 3 and 4 context switched for every 50 ms. Screenshot is as below, it can be seen that thread 3 and 4 context switch for every 50 ms.

```
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]
50 ms second has passed
FUN 3: TICK [4]
FUN 3: TICK [5]
FUN 3: TICK [6]
FUN 3: TICK [7]
FUN 3: TICK [8]
FUN 3: TICK [9]
```

NOTE: The macros are defined for RRScheduler in `Kernel.C`, hence including the file too as part of `MP5.zip`