CSS 430 Operating Systems
Program 2B: ThreadOS Scheduler

1. Purpose

This assignment implements the multilevel feedback-queue (MFQ) scheduling algorithm in ThreadOS and compares it with the pre-implemented round-robin (RR) scheduling algorithm, using pre-implemented test thread programs. **You can find a template file named Scheduler_mfq_hw2b.java from Files/code/prog2 on Canvas**. You need to complete this file. The complete version of the file is referred to as Scheduler_mfq.java in the rest of this document.

2. Structure of ThreadOS Scheduler

The algorithm of *ThreadOS Scheduler*, (i.e., Scheduler.java) is based on our lecture slide, while its data structure is extended to manage each user thread using a thread control block (TCB).

2.1. Thread Control Block (TCB.java, see this file from Files/code/prog2)

. The current implementation of TCB includes four private data members: (1) a reference to the corresponding thread object (*thread*), (2) a thread identifier (*tid*), (3) a parent thread identifier (*pid*), and (4) the *terminated* variable to indicate the corresponding thread has been terminated. The *TCB* constructor simply initializes those private data members with arguments passed to it. The *TCB* class provides four public methods to retrieve its private data members: *getThread( )*, *getTid( )*, *getPid( )*, and *getTerminated( )*. In addition, it also has *setTerminated( )* that sets *terminated* true.

2.2. Scheduler.java (=Scheduler_pri.java, these two files are the same, see Scheduler_pri.java from Files/code/prog2)

The following table shows the private data members of Scheduler.java.

Private data members	Descriptions
Vector<TCB> queue;	a list of all active threads, (to be specific, TCBs)
int timeSlice;	a time slice allocated to each user thread execution private
static final int DEFAULT_TIME_SLICE = 1000;	the unit is millisecond. Thus 1000 means 1 second
boolean[] tids;	Each array entry indicates that the corresponding thread ID has been used if the entry value is true.
static final int DEFAULT_MAX_THREADS = 10000;	tids[] has 10000 elements, related to TCB related to TCB

The following shows all the methods of *Scheduler.java* (=Scheduler_pri.java).

Methods	**Descriptions**
private void initTid(int maxThreads)	allocates the *tid[]* array with a *maxThreads* number of elements
private int getNewTid( )	finds a *tid[]* array element whose value is false, and returns its index as a new thread ID.
private boolean returnTid(int tid)	sets the corresponding *tid[]* element, (i.e., *tid[tid]*) false. The return value is false if *tid[tid]* is already false, (i.e., if this *tid* has not been used), otherwise true.
public int getMaxThreads( )	returns the length of the *tid[]* array, (i.e., the maximal # threads to be spawned in the system).
public Scheduler(int quantum, int maxThreads)	finds the current thread's TCB from the active thread queue and returns it
private void schedulerSleep( )	puts the Scheduler to sleep for a given time quantum
public TCB addThread(Thread t)	allocates a new TCB to this thread *t* and adds the TCB to the active thread queue. This new TCB receives the calling thread's id as its parent id.
public boolean deleteThread( )	deletes the current thread's TCB from the active thread queue and marks its TCB as terminated. The actual deletion of a terminated TCB is performed inside the run( ) method, (to prevent race conditions).
public void sleepThread(int milliseconds)	puts the calling thread to sleep for a given time quantum.
public void run( )	This is the heart of the Scheduler. The differences from the lecture slide include: (1) retrieving a next available TCB rather than a thread from the active thread list, (2) deleting it if it has been marked as "terminated", and (3) starting the thread if it has not yet been started. Other than these three differences, the Scheduler repeats retrieving a next available TCB from the list, raising up the corresponding thread's priority, yielding CPU to this thread with *sleep( )*, and thereafter lowering the thread's priority upon the current time quantum (1 sec) expiration.

The scheduler itself is started by *ThreadOS Kernel*. It creates a thread queue that maintains all user threads invoked by the *SysLib.exec(String args[])* system call. Upon receiving this system call, *ThreadOS Kernel* instantiates a user thread and calls the scheduler's *addThread(Thread t)* method. A new *TCB* is allocated to this thread and enqueued in the scheduler's thread list. The scheduler repeats an infinite *while* loop in its *run* method. It picks up the next available *TCB* from the list. If the thread in this *TCB* has not yet been activated (but instantiated), the scheduler starts it first. It thereafter raises up the thread's priority to execute for a given time slice.

When a user thread calls *SysLib.exit( )* to terminate itself, the *Kernel* calls the scheduler's *deleteThread( )* in order to mark this thread's *TCB* as terminated. When the scheduler dequeues this *TCB* from the circular queue and finds out that it has been marked as terminated, it deletes this *TCB*.

2.3. Scheduler_rr.java (see this file from Files/code/prog2)

This version of ThreadOS Scheduler is based on a rigid RR strategy, using Thread.suspend( ) and Thread.resuem( ). Scheduler_rr.java's code made the following five modifications onto Scheduler_pri.java.

1. Removed *setPriority(2)* (line 96) from the *addThread( )* method,
2. Add *this.interrupt( )* between lines 111 and 112 in the *deleteThread( )* method,
3. Removed *setPriority(6)* (line 127) from the *run( )* method,
4. Replaced *current.setPriority(4)* (line 143) with *current.resume( )*,
5. Removed *current.setPriority(4)* (line 148) from the *run( )* method, and finally
6. Repalced *current.setPriority(2)* (line 157) with *current.suspend( )*.

2.4. Test Programs: Test2.java and Test2b.java (see these files from Files/code/prog2)

Test2 and *Test2b* spawn five child threads from *TestThread2* and *TestThread2b*, respectively, each named *Thread[a]*, *Thread[b]*, *Thread[c]*, *Thread[d]*, and *Thread[e]*. Threads spawned from TestThread2b print out a heartbeat message every 0.1 second: "*Thread[name]* is running". If the scheduler's time slide is 1 second, each thread should print out this message 10 times consecutively until it is switched to another thread or gets terminated. At the end, TestThread2b prints out its execution stats, including the response time, turnaround time, and execution time. TestThread2 is the same as TestThread2b except printing out only its execution stats (thus no heartbeat messages). Test2b, (i.e., spawning TestThread2b threads) will be used in Task2: Verification, whereas Test2, (i.e., spawning TestThread2 thread) will be used in Tast3: Performance Comparison.

These five child threads have the following running time:

Thread Names	Running Time (in Milliseconds)
Thread[a]	5000
Thread[b]	1000
Thread[c]	3000
Thread[d]	6000
Thread[e]	500

3. Use of Thread.suspend() and Thread.resume() methods

We use *Thread.suspend( )* and *Thread.resume( )* only in this assignment, in particular only inside *Scheduler.java*. **Note that, in general, you should avoid using *Thread.suspend( )* and *Thread.resume( )* in your future thread programs (i.e., assignment 3B)**.

The *suspend( )* method suspends a target thread, whereas the *resume( )* method resumes a suspended thread. These methods are system independent. No matter what operating systems you use, a target thread is suspended and resumed immediately. In order to implement a rigid round-robin CPU scheduling, we could modify *ThreadOS Scheduler* to dequeue a front user thread from its circular list, to resume it with the *resume* method, and to suspend it with the *suspend* method after an execution quantum has expired. However, *suspend* and *resume* may cause a deadlock if a suspended thread holds a lock, and a runnable thread tries to acquire this lock. **To avoid any deadlocks, we must pay our closest attention when using them with *synchronized*, *wait( )* and *notify( )* keywords**. (They will be exercised in assignment 3B to realize inter-threads synchronization.) When you peek at Scheduler.java, you see some *synchronized* keywords in it. **Don't remove them or put additional *synchronized* keywords**, otherwise your Scheduler.java will easily fall into a deadlock.

When you compile Java programs that use deprecated methods such as *suspend( )* and *resume()*, you must compile them with a *-deprecation* option. Although the *javac* compiler will print out some warning messages, just ignore them in assignment 2.

```
[cssmpi1h]$ javac -deprecation Scheduler.java
./Scheduler.java:128: warning: resume() in java.lang.Thread has been deprecated
                        currentThread.resume( );
                                     ^
./Scheduler.java:136: warning: suspend() in java.lang.Thread has been deprecated
                        currentThread.suspend( );
                                     ^
2 warnings
[cssmpi1h]$
```

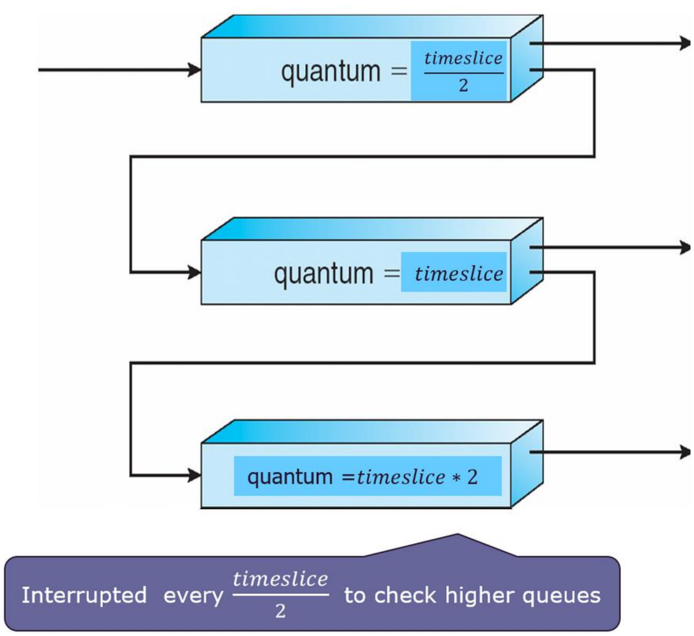
4. Statement of Work

Task1: MFQ Implementation. Implement an MFQ algorithm in Scheduler_mfq.java. While the file name is Scheduler_mfq.java, the actual class name should be Scheduler, so that you can distinguish it from the other two versions (i.e., Scheduler_pri.java and Scheduler_rr.java) but compile it with other ThreadOS classes by just copying it into Scheduler.java. The generic algorithm is described in the textbook. Your multilevel feedback-queue scheduler must have the following specification:

1. It has three queues, numbered from 0 to 2.
2. A new thread's TCB is always enqueued into queue 0.
3. Your scheduler first executes all threads in queue 0. The queue 0's time quantum is half of the one in Part 1's round-robin scheduler, (i.e., timeSlice / 2).
4. If a thread in the queue 0 does not complete its execution for queue 0's time slice, (i.e., timeSlice / 2), the scheduler moves the corresponding TCB to queue 1.
5. If queue 0 is empty, it will execute threads in queue 1. The queue 1's time quantum is the same as the one in Part 1's round-robin scheduler, (i.e., timeSlice). **However, in order to react new threads in queue 0, your scheduler should execute a thread in queue 1 for timeSlice / 2 and then check if queue 0 has new TCBs.** If so, it will execute all threads in queue 0 first, and thereafter resume the execution of the same thread in queue 1 for another timeSlice / 2.
6. If a thread in queue 1 does not complete its execution for queue 1's time quantum, (i.e., timeSlice), the scheduler then moves the TCB to queue 2.
7. If both queue 0 and queue 1 are empty, it can execute threads in queue 2. The queue 2's time quantum is a double of queue 1's time quantum, (i.e., timeSlice * 2). However, in order to react threads with higher priority in queue 0 and queue 1, your scheduler should execute a thread in queue 2 for timeSlice / 2 and then check if queue 0 and queue 1 have new TCBs. The rest of the behavior is the same as that for queue 1.
8. **If a thread in queue 2 does not complete its execution for queue 2's time slice, (i.e., timeSlice * 2), the scheduler puts it back to the tail of queue 2. (This is different from the textbook example that executes threads in queue 2 with FCFS.)**

To focus on the essence of the MFQ algorithm, use Scheduler_mfg_hw2b.java that has already implemented the logic except Scheduler.run( ) method. Your focus should be placed on this run( ) method.

The following figure illustrates the quantum value of each queue and how to check each queue.



Task2: Verification. Run Scheduler_pri.java, Scheduler_rr.java, and Scheduler_mfq.java (this is the one you implemented) with Test2b that spawns 5 child threads, each printing out "Thread[..] is running" every 1 second.

```
[cssmpi1h]$ cp Scheduler_pri.java Scheduler.java
[cssmpi1h]$ javac *.java
[cssmpi1h]$ java boot
threadOS ver 1.0:
Type ? for help
threadOS: a new thread (thread=Thread[Thread-3,2,main] tid=0 pid=-1)
-->l Test2b
l Test2b
threadOS: a new thread (thread=Thread[Thread-6,2,main] tid=1 pid=0)
threadOS: a new thread (thread=Thread[Thread-8,2,main] tid=2 pid=1)
threadOS: a new thread (thread=Thread[Thread-10,2,main] tid=3 pid=1)
threadOS: a new thread (thread=Thread[Thread-12,2,main] tid=4 pid=1)
threadOS: a new thread (thread=Thread[Thread-14,2,main] tid=5 pid=1)
threadOS: a new thread (thread=Thread[Thread-16,2,main] tid=6 pid=1)
Thread[a] is running
....
Thread[b] is running
....
-->q
[cssmpi1h]$ cp Scheduler_rr.java Scheduler.java
[cssmpi1h]$ javac *.java
[cssmpi1h]$ java boot
threadOS ver 1.0:
Type ? for help
threadOS: a new thread (thread=Thread[Thread-3,2,main] tid=0 pid=-1)
-->l Test2b
l Test2b
threadOS: a new thread (thread=Thread[Thread-6,2,main] tid=1 pid=0)
....
Thread[a] is running
....
-->q
[cssmpi1h]$ cp Scheduler_mfq.java Scheduler.java
[cssmpi1h]$ javac *.java
[cssmpi1h]$ java boot
threadOS ver 1.0:
Type ? for help
threadOS: a new thread (thread=Thread[Thread-3,2,main] tid=0 pid=-1)
-->l Test2b
l Test2b
threadOS: a new thread (thread=Thread[Thread-6,2,main] tid=1 pid=0)
....
Thread[a] is running
....
-->q
```

Can you see the difference between the three schedulers? Save all the outputs as what your turn in.
Note that Scheduler_rr.java and Scheduler_mfq.java repeatedly print out no heartbeat messages. This means that Loader.java receives a CPU time quantum but does nothing.

Task3: Performance Comparison. Instead of running Test2b.java from ThreadOS loader, run **Test2.java**. Focus on only Scheduler_rr.java and Scheduler_mfq.java.

```
[cssmpi1h]$ cp Scheduler_rr.java Scheduler.java
[cssmpi1h]$ javac *.java
[cssmpi1h]$ java boot
threadOS ver 1.0:
Type ? for help
threadOS: a new thread (thread=Thread[Thread-3,2,main] tid=0 pid=-1)
-->l Test2
....
-->q
[cssmpi1h]$ cp Scheduler_mfq.java Scheduler.java
[cssmpi1h]$ javac *.java
[cssmpi1h]$ java boot
threadOS ver 1.0:
Type ? for help
threadOS: a new thread (thread=Thread[Thread-3,2,main] tid=0 pid=-1)
-->l Test2
....
-->q
```

Did your Scheduler_mfq.java demonstrate better stats than Scheduler_rr.java? Save all the outputs as what your turn in.

5. What to Turn in

Total 20pts.

	Materials	Points	Note
1	Task 1: Scheduler_mfq.java a. Code organization: **+2pts** • Well organized: 2pts, • Poor comments or bad organization: 1pt, or • No comments and horrible code: 0pts b. Correctness: **+8pts** • queue[0], [1], and [2] guarantee 1 slice, 2 slices, and 4 slices of 500msec for running each thread, respectively: **+3pts** • At each slice of 500msec, Scheduler checks if any threads exist in higher-priority queues: **+3pts**, and • If the current thread used up the slices of the current queue, it would go to the next lower queue or stay in the lowest queue: **+2pts**	10	Submit this java file individually. When you submit this file, don't rename it to Sheduler_mfq.java.
2	Task 2: one or more execution snapshots a. The snapshots show longer threads go down from queue[0] to queue[2]: 5pts, b. The snapshots show little difference between shorter and longer threads: 3pts, or c. No results: 0pts	5	Include the snapshots in a pdf file named: FirstNameLastName_prog2B.pdf. Submit this pdf file individually. Please clearly label each snapshot.
3	Task 3: one or more execution snapshots a. The snapshots show Scheduler_mfq performs better than Scheduler_rr.: 5pts, b. The snapshots show Scheduler_mfq does not outperform Scheduler_rr: 3pts, or c. No results: 0pts	5	