**Overview**

*The original distribution of Nachos uses the round-robin scheduling of 100 time-ticks.* In this project you are asked to implement the **multiple level feedback queue scheduling with your own simulated I/O**. You need to implement **at least 4 levels of ready queues** with different amount of time quantum. You need to add new flags such as “-q1” or “-q2” that are used to **set the size of time quantum for each level** of ready queue.

**Task 1 (40pts):** You need to design your own I/O simulating input(read)/output(write) operations. These IO operation routines can be used by Nachos threads.

* Write operation: when a thread calls this routine, a new I/O request event is created and set with random amount of waiting time for blocked state. Then the event will be added to the internal I/O queue. Afterwards this request blocks the calling thread (current thread) until your output message is displayed on the screen.
* Read operation: like Write operation, this also creates and sets a new I/O request event with random amount of waiting time. Then the event will be added to the internal I/O queue. The calling thread will be blocked until the parameter of buffer is filled with your own content, and at this moment, you can display a simple message.
* Write waiting(blocking) time << Read waiting(blocking) time
* I/O request event structure: you can define your own data structure including I/O type, calling thread, buffer holding data between call and completion of the request, parameters used by I/O routines, completion time (decided by random waiting time), and so on.
* I/O request queue: you can keep this as a Kernel’s internal I/O event queue using sorted list by the completion time. You can simulate I/O events using Alarm.
* I/O Simulation can be periodically invoked by Alarm. If completion time of a pending I/O is less than or equal to current time, it raises I/O interrupt and then I/O interrupt handler will be invoked.
* I/O interrupt handler: You need to implement your own simulated I/O interrupt. When a I/O completes (simulated with Alarm), I/O interrupt is raised.
  + Write operation: this interrupt indicates that the request is done and requesting thread should be inserted into a corresponding ready queue because I/O request increments the priority of the thread. You have to make sure that the thread state as Ready-to-Run.
  + Read operation: this interrupt handler will copy your content to the calling thread’s buffer. The thread should be inserted into a corresponding ready queue because I/O request increments the priority of the thread. You have to make sure that the thread state as Ready-to-Run.
  + At the end of interrupt handling, there should be scheduling.
  + You need to display appropriate messages.
* Kernel can have a new simulated I/O component, so thread functions can call the simulated I/O operations and they will be simulated.
* Implement your design of simulated I/O in threads directory and update Makefile.

**Task 2 (40pts):** Make your Nachos system run with **MLFQ** scheduling. Each thread will take the CPU according to its priority. In this assignment, every thread starts with the highest priority.

* **Multi-level Feedback Queue scheduler**
  + Every thread begins with the highest priority.
  + If a thread calls one of simulated I/O operations (read/write) before time quantum, it will be blocked (with random amount of time) and gives up the CPU.
  + After the random amount of time, the simulated I/O interrupt is invoked, the blocked thread will be ready with a higher priority (if it was already in highest priority, it will have the same priority and) and inserted into the ready queue.
  + In order to avoid starvation, you need to combine your scheduler with aging. If a thread has not been on the CPU for a certain amount of time, increase its priority and move it up to a higher level queue. This waiting time is not related to I/O blocking time.
  + If a time quantum expires, the threads gives up the CPU, its priority will be decremented and inserted into a lower level queue. If the thread was already in the lowest level queue, it will stay at the same queue.
* Add a Nachos flag **-q1 xxxx** that sets how long the time quantum will be for each thread at level 1 ready queue.
  + xxxx (q1) < yyyy (q2) < zzzz (q3) < …

**Task 3 (20pts):** Write your own threadtest.cc to prove your implementation working correctly. Quality of your testing cases will be graded.

* You need to set random seed (current time is usually used) for simulation.
* You need to create meaningful number (> 10) of Nachos threads in your testing routine.
  + CPU-bound threads, IO-bound threads, mixed
* You need to display meaningful information to prove your scheduling: context switching from expiring time quantum, simulated I/O requests, simulated I/O interrupt; dynamic priority changes with I/O requests and aging.