Shane Moore

Multilevel Feedback Queue Discrete Event Simulator

I thought it was important to note my initial drafts and thought processes about this project until I attained a final product.

Draft 1: **Using Multiple Processes:** My initial thoughts from the documentation were that the IODevices and CPU would all run concurrently. I was going to create multiple processes for these and pass the data through pipes. I was going to serialize the objects to pass them through the pipes. Realizing this might be too difficult I tried to do something harder and to use multiple threads

Draft 2: **Multithreading:** I started using pthreads for this project to have a multithreaded program in which there would be 5 threads. A main thread which would pass events and jobs to the other threads. 3 IO device threads and a CPU thread.

Draft 3: After I spoke with Professor Dhall and researched about event simulation I came to realize what was actually needed. Since the first two drafts I had a good template to start with (object wise).

**Main Loop**

The simulator is build upon a main loop which gets the events and schedules the next events in the queue. These events decide which events will be scheduled next. The loop is exited through a break statement when there are no more jobs in the job\_scheduler (long term scheduler). I initially create the 50 jobs and process events from there

The main objects for this simulation include

**CPU** – This handles the compute times of the job

The main part to be discussed within the CPU is the **compute()** method:

This method provides computation for the job and returns an integer on what event should be scheduled next. This will return a 3 if time slice is finished a 9 if we moved a job to the finished queue and 0-2 if we are going to an IO Device.

We cycle through the memory depending on statement we determine what to do.

Since the memory is just an array of numbers we know that when memory%2 ==0 we know that it is an IOdevice since all the memory is 2n+1 same thing with .

When the memory stack is empty the job is finished and put into the finished queue to output data at end of program.

I chose to create a finished queue named **queue<Job\*> finished\_jobs** which when a job finished it is added to this queue. This provides

**Event** – There are 7 main events in this simulation they include the following:

1. Arrival of a new job
2. Schedule a job
3. End of current time slice
4. Completion of job at device 1
5. Completion of Job at device 2
6. Completion of Job at device 3
7. Time to upgrade level of a job

Events are exactly created as described in the documentation for this project.

**EventQueue** – This queue pops and pushes events into it when event is popped it means it is time for that event. When pushed we are scheduling it.

This queue is generic and is mainly just a holder for Events.

**IODevice** - Three IODevices are instantiated all with different times.

There are three different IO devices that can be created

1: With a computation time of 40

2: With a computation time of 30

3: With a expotentially defined compute time.

The main method to be discussed in this object is the **do\_io() method**:

This method mainly just adds the io time for the device to the job that is in its wait queue.

**Job** – The job holds all the timing and compute info regarding a specific job.

The route I took with the job is to store information about the job rather than having a job table.

The Following are all the data held in the Job:

**int** n; **The random expotentially distributed number generated**

**int** id; **The id of the job**

PCB\* pcb; **Process Control Block for this Job**

**int** ts; **Time Slice of this job**

**int** cur\_lev; **Holds which level the job is in ReadyList**

**int** total\_time; **Total time spent in Computation**

**int** io1\_time; **Total time in Iodevice 1**

**int** io2\_time; **Total time in Iodevice 2**

**int** io3\_time; **Total time in Iodevice 3**

**bool** in\_queue; **Tracks if the job is in ReadyList or not**

**int** completion\_time; **Tracks the completion time of job**

**int** arrival\_time; **Holds the arrival time of job into ready**

**int** in\_rdy;

All the methods for this are getter and setter methods which modify all of these attributes depending

**PCB** – Each job has a process control block which manages computation and records times.

The memory for the PCB was implemented using a stack of integers. This stack has 2n+1 integers in it which represents each statement.

**ReadyList** – The readylist is the multilevel feedback queue which schedules jobs for processing

I decided not to go with a table to output the info but rather push all of the finished jobs to a finished queue with –in the CPU. At the end of all the events **print\_finished()** in the CPU is called and prints out all the data.