OS Group (2 or 3 each group) Project 1 Process Scheduling

Due: 11:59 pm Oct 9th

The **short-term process scheduler** is the core of OS. It is responsible for choosing a new process to run whenever the CPU is available.

Overview

In this project, you will implement a single processor OS simulator. You will build a simulator of a simple operating system to learn more about OS in general and process schedulers in specific. Your task is to implement the CPU scheduler, using three different scheduling algorithms: FCFS, Round-Robin and Static Priority.

Scheduling Algorithms

For your simulator, you will implement the following three CPU scheduling algorithms:

- *First-Come, First Served (FCFS)* - Runnable processes are kept in a first-in, first-out ready queue. FCFS is non-preemptive; once a process begins running on a CPU, it will continue running until it either completes or blocks for I/O.
- *Round-Robin* - Similar to FCFS, except preemptive. Each process is assigned a time slice when it is scheduled. At the end of the time slice, if the process is still running, the process is preempted, and moved to the tail of the ready queue.
- *Static Priority* - The processes with the highest priorities always get the CPU. Lower-priority processes may be preempted if a process with a higher priority becomes runnable.

Process States

In this OS simulation, you will use the five-state model for a process:

- NEW - The process is being created, and has not yet begun executing.
- READY - The process is ready to execute, and is waiting to be scheduled on a CPU.
- RUNNING - The process is currently executing on a CPU.
- WAITING - The process has temporarily stopped executing, and is waiting on an I/O request to complete.
- TERMINATED - The process has completed.

Queues of your simulator:

1. New Queue: To keep track of the processes in the NEW state.

2. Ready Queue: On systems, there are a large number of processes, but one CPU on which to execute them. When there are more processes ready to execute than CPUs, processes must wait in the READY state until the CPU becomes available. To keep track of the processes waiting to execute, we keep a ready queue of the processes in the READY state.

3. I/O Queue: To keep track of the processes waiting to I/O, we keep an I/O queue of the processes in the WAIT state.

4. Terminated Queue: To keep track of the processes that already terminate

Input file:

The txt file of processes will consist of a number of lines, each describing a single process. Each line will consist of four fields: Process ID, Arrival order, Priority, CPU-I/O burst sequence. In this sequence, a black-color **number [0-9]** is a CPU burst and a red-color **number [0-9]** is an I/O burst.

Here, a CPU burst is the period when it is executing instructions; an I/O system burst is the period when it services requests to fetch information. A process execution consists of a cycle of CPU execution and I/O wait. Processes alternate between these two states. Process execution begins with a CPU burst. That is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution

ID	Arrival order	Priority	CPU-I/O burst Sequence
1001	0	10	4256346346341
1002	1	7	31425331624331 1
1003	2	12	6346321
1004	3	9	346846341

Output:

Your simulator needs to compute average, minimum, and maximum values as well as their standard deviations for the following measurements:

- Latency - How long does it take between when a task is created and when it completes
- Response Time - How long does it take between when a task is created and when it first responds (first IO burst finishes)
- Throughput - How many tasks / unit time can the system process.

ID	Priority	Latency (ms)	Response (ms)
1001	10	2.8662466456	0.154676573456
1002	7	5.2544443667452	1.3232453655452
1003	12	11.549435498741	3.532654754571
1004	9	2.5846135798	0.54543525798

For example, the throughput of your CPU is 7 per ms.

Report:

Your report contain the average, minimum, and maximum values as well as their standard deviations of Latency and Response Time.

Java Project Structure:

```
Class PCB{

//To do: PCB data structure of a process

        //for example: Process_id, Arrive_time, state,
        PositionOfNextInstructionToExecute(PC value)
         and so on
}

//Process image class of a process including PCB, code, data and stack

Class ProcessImage {

      {
        public PCB Pcb_data;
        public string code= CPU-I/O burst Sequence;
       //to do: other variables help you computing the latency, response
      }
```

```
        public Process(String process) {
                    //set PCB data, code and others
                     //set state as "NEW";
          }

}

//CPU class

Class CPU{

        public boolean BusyOrNot;
        public int PC; //Your CPU only has one register PC
        Public int timeslice;
        public CPU(int settimeslice)
        {
              Timeslice= settimeslice;
              BusyOrNot=false;
        }

        Public Pair<int PC, String state> execute(Process P){
              BusyOrNot=true;
             /* read the CPU burst number, say #, from the position
        PositionOfNextInstructionToExecute of P.
              Repeat calling Bubble Sort() for # times and then continue.
              If the code runs out, return (PositionOfNextInstructionToExecute,
        “terminated”), then OS put it back to the terminated queue.

              If the slice of time (restricted number of calling Bubble sort() for a
        process each time) runs out, return (PositionOfNextInstructionToExecute+1,
        “ready”), then OS put it back to the ready queue.

            Otherwise, return (PositionOfNextInstructionToExecute+1, “wait”)
        (namely, P has an I/O request and then OS remove it from the ready queue
        and sent it to I/O queue)
        */
        }
        Public BubbleSort(){
        }
```

```
        Public Boolean CPUisBusy?()

        {       return BusyOrNot;}

}

//Operating system class

Class OS{

            public CPU cpu;
            public IOdevice io;
            public boolean isCPUAvailable;
            public ProcessTable process_Table;
            public ArrayList<Process> New_Queue;
            public ArrayList<Process> Ready_Queue;
            public ArrayList<Process> Wait_Queue;
            public ArrayList<Process> Terminated_Queue;
            //Read the txt input file, for each line, create a process and record its arrival
time
            //Put each process in New_Q queue initially then put them in Ready_Q
            //Always check whether the CPU is idle or not; if yes, use your scheduler
algorithm to select a process from the Ready_Queue for CPU execution\
          // According to the return value of CPU execute(), put the process into the
corresponding queue.
          //Record the time of every operation for computing your latency and
response
}

Class IOdevice{

        Public IOdevice(ArrayList<Process> Wait_Queue)

        public boolean BusyOrNot;

        //Always pick one process from Wait_Queue to exeute

        public String execute(int IO_burst){

              BusyOrNot=true;
             //Call Bubble Sort() for IO_burst times and then return “ready”;
```

```
        }
        Public BubbleSort(){}
        }

}
```

Java techniques:

Thread: https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

Synchronized Methods

https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html