Operating System
CS 370
Fall 2016

Project: 3
CPU Scheduling

Due Date: Thursday, October 13th 2016
Must be submitted before midnight

Total Points: 170

Teaching Assistant: Pradip Singh Maharjan
Office Hours:
MoWe: 2:00 pm to 4:00 pm
TuTh:10:00 am to 12:00 pm

Course Web Page : `http://osserver.cs.unlv.edu/moodle/`

Requirements:

- This project must be implemented in C++.
- Project must be submitted through the website.
- Project must compile on Linux machines in B361 or on cardiac.cs.unlv.edu
- The first submission of this project must compile as "g++ project3a.cpp". Similarly, second or third submission must compile as "g++ project3b.cpp" or "g++ project3c.cpp" respectively.

Reading Requirements:

Linux Schedulers
`http://www.informit.com/articles/printerfriendly.aspx?p=101760`

Purpose:

The purpose of this project is to provide insight into the Linux Process Scheduling Algorithm presented in class by simulating a simplified version of the CPU scheduler.

Background:

The Linux Process Scheduler divides CPU time into epochs. In a single epoch, every process has a specified time slice. In a single epoch, the CPU's job is to schedule contending processes to use the CPU in a fair and efficient manner.

Time slice, priority, nice value and bonus:

The Linux Process Scheduler uses time slice to prevent a single process from using the CPU for too long. A time slice specifies how long the process can use the CPU. In our simulation, the minimum time slice possible is 5ms and the maximum time slice possible is 450 ms. The scheduler assigns higher time slices to processes that are more interactive and lower time slices to processes that are more CPU intensive. Note that time slice is a positive integer in this range [5, 450]. To calculate the time slice, we use this formula:

$$timeSlice = (int)\left(\left(1 - \frac{priority}{150.0}\right) \times 445 + 0.5\right) + 5$$

Each process is supplied with a priority level that ranks a process based on their worth and need for processor time. The priority levels range from 100 to 150 [100,150]. Processes with a lower priority will run before a process with a higher priority. Process with a lower priority level also receives a longer time slice.

A process's initial priority (sometimes called static priority) is calculated based on its nice value. Nice values range from -20 to +19 [-20, 19] indicating how nice the process is. Larger nice values correspond to a lower priority. CPU intensive processes typically have higher nice values while IO bound processes have lower nice values. Nice values are provided with the input file. Note that a priority value is a positive integer in this range [100,150]. To calculate the initial (or static) priority we use this formula:

$$original\ priority = (int)\left(\frac{nice + 20}{39.0} \times 30 + 0.5\right) + 105$$

After a process exhausts its time slice, it will join the expired queue or go back to the active queue (we will discuss this later). Before doing that, it has to calculate its new priority (sometimes called the dynamic priority). This is the formula used to calculate the dynamic priority:

$$priority = original\ priority + bonus$$

Bonus points are given to processes that either use too much or too little CPU time. Bonus points are integers range from -10 to +10 ([-10, 10]). Here are the guidelines to calculating bonus points:
$if(totalCPU < totalIO)$

$$bonus = (int)\left(\left(\left(1 - \frac{totalCPU}{(double)totalIO}\right) \times -10\right) - 0.5\right)$$

else

$$bonus = (int)\left(\left(\left(1 - \frac{totalIO}{(double)totalCPU}\right) \times 10\right) + 0.5\right)$$

Note : Here totalIO mean the total time spent in the IO queue to this point and totalCPU mean the total time spent in the CPU to this point.

Run queues/Priority Array:

The runqueue is the list of runnable processes on a given processor. There is only one runqueue per processor. Each runqueue contains two priority arrays: Active and Expired. Each priority array contains one queue of runnable processes per priority level. Each array has 140 levels, thus 140 queues. A process with priority level 125 will go to the "125^{th} " queue. In this project, we will not implement all 150 queues. We will only implement a single queue. In the real Linux scheduler, each queue represents one priority level. In our project, we will have a single data structure as discussed later on. The active array contains processes that have yet to exhaust their time slice. Once their time slice is exhausted, it will go to the expired array.
There is also an IO queue. This queue is used to hold
processes that are using IO. There will be no waiting for access to IO devices.
Thus, all the processes in the IO queue can decrease their IO burst in each clock tick.

Data Structure:

- **Process:**
  - pid
  - start time
  - end time
  - priority
  - timeslice
  - list of cpu bursts
  - list of io bursts (the number of io burst will always be 1 less than the number of cpu bursts)
- **Start up Queue:**
  - holds the processes which have not started executing (start time $>$ clock)

    (recommendation: order by start time)
- **Active/Expired Queue:**
  - holds the processes that are waiting to use the cpu

    (recommendation: order by priority)
- **IO Queue:**
  - holds processes that are currently waiting for IO burst to finish.

    (recommendation: order by current io burst left)
- **Finished Queue:**
  - holds the processes that have completed all its cpu bursts

    (recommendation: order by finish time)
- **CPU:**
  - holds the process that is currently executing in the cpu.

Algorithm:

Use this algorithm for the order of which you perform each operation. If you do not use this order you have incorrect output.

```
read input
clock = 0
While (true) {
 insert processes to the active queue if they are to start at this
     clock tick. (calculate priority and timeslice).
 if the cpu is empty the lowest priority process in the active queue
     is put into the cpu. If more than two processes have the same
     priority then FIFO strategy can be used.
 Check if the lowest priority process in the active queue has a lower
     priority then the process in the cpu.  If so then preempt.
     Preempted process goes back to the active queue.
 Perform cpu (decrement the timeslice of the process in the cpu)
 Perform IO(decrement the IO burst for all processes in the IO queue)
 if there is a process in the cpu (call this process p)
  if p's current cpu burst is exhausted
   if p is done with all cpu bursts send to the finished queue
   if p is not done with all cpu bursts (which means there is an IO
       burst) send to the IO queue.
  if p's timeslice is exhausted send to the expired queue and
      recalculate priority and timeslice.
 If there is any process in the IO queue that is finished with its IO
     burst (there can be more than one, call this process p)
  If p's timeslice is exhausted move to the expired queue and
      recalculate the priority and timeslice.
  If p still has timeslice left insert p into the active queue.
 If the startup queue, ready queue, expired queue, IO queue and the
     cpu are all empty then break out of the while loop (the
     simulation is complete)
 if the ready queue and cpu are empty and the expired queue is not
     empty then switch the expired and active queues, this can be done
      by just swapping the pointers of both the queues.
 Increment the clock (clock++)
}
print ending report
```

Input Specification:

The input file will be read using Unix redirection.
i.e. ./a.out < InputFile.txt
The input of the program will be a text file that contains a finite number of jobs (or processes), one per line. The order of the process can be arbitrary (i.e. not in the order of arrival time). You should read each number until the end of the line. All numbers are integers.
All input will be assumed to be correct.

- First column is a process's nice value.
- Second column is a process's arrival time.
- Third column is the number of CPU bursts. (remember number of IO bursts is CPU bursts -1)
- Fourth column is the CPU burst time.

- Subsequent column pairs represent I/O bursts and CPU bursts, respectively.
- The last input of the line will always be a CPU burst time.
- *** on its own line to designate the end of input

```
Example Input 1:
11 200 2 300 321 450
1  100 1 4000
-14 200 2 250 350 200
10  200 1 700
***
```

```
Example Input 2:
0  0  2 1000 100 1000
1  200 2 300  300 300
-20 200 2 300  300 300
-3 300 1 300
***
```

```
Example Input 3:
0 100 4 300 300 300 300 300 300 300
1 150 4 300 300 300 300 300 300 300
2 200 4 300 300 300 300 300 300 300
3 250 1 300
***
```

The order of the input will determine the process identification (PID). The process described in the first row will be considered to have a PID value of 0. The process described in the second row will be considered to have a PID value of 1, and so on.

Output Specification:

For Each Process:
During the processing of each process, your simulation will output significant events. The following events must be in the output of the simulation:

- Arrival of a process in the active.

```
1.[clock] <pid> Enters ready queue (Priority: _, TimeSlice: _)
2.[25] <2> Enters ready queue (Priority: 120, TimeSlice: 51)
```

- When a process enters the CPU.

```
1.[clock] <pid> Enters the CPU
2.[137] <0> Enters the CPU
```

- When a process is preempted

```
1.[clock] <pid> Preempts Process pid
2.[200] <2> Preempts Process 0
```

- When a process is finished with all cpu bursts.

```
1.[clock] <pid> Finishes and moves to the Finished Queue
2.[3599] <0> Finishes and moves to the Finished Queue
```

- When a process finishes a cpu burst and moves to the IO queue.

```
1.[clock] <pid> Moves to the IO Queue
2.[1552] <1> Moves to the IO Queue
```

- When a process finishes its timeslice and moves to the expired queue.

```
1.[clock] <pid> Finishes its time slice and moves to the Expired
    Queue (Priority: _, Timeslice: _)
2.[1630] <2> Finishes its time slice and moves to the Expired
    Queue (Priority: 107, Timeslice: 78)
```

- When a process finishes its IO burst and moves to the expired queue.

```
1.[clock] <pid> Finishes IO and moves to the Expired Queue (
    Priority: _, TimeSlice: _)
2.[895] <2> Finishes IO and moves to the Expired Queue (Priority:
     112, TimeSlice: 80)
```

- When a process finishes its IO burst and moves to the active queue.

```
1.[clock] <pid> Finishes IO and moves to the Ready Queue
2.[1294] <2> Finishes IO and moves to the Ready Queue
```

- When active and expired queues are switched.

```
1.[clock] *** Queue Swap
2.[1119] *** Queue Swap
```

Overall Performance Output:

This output is to be computed after all processes have completed executing. For each Process, print individual statistics (in order of termination):

$$Turn\ around\ time(TAT) = time\ completed - time\ arrived$$

$$Total\ CPU\ time(TCT) = sum\ of\ all\ CPU\ bursts$$

$$Waiting\ Time(WT) = TAT - TCT - TIT(total\ IO\ time).$$

$$Percentage\ of\ CPU\ utilization\ time(CUT) = \frac{TCT}{TAT}$$

Give value in percentage rounded to the nearest 10th. i.e. 97.8

After all individual statistics are displayed, print overall statistics. Using P_i to denote the process used for the calculation in the following formulas:

$$AVG\ \ TAT = \frac{\sum_{i=0}^{n-1} TTofP_i}{n}$$

$$AVG\ \ WT = \frac{\sum_{i=0}^{n-1} WTofP_i}{n}$$

$$AVG\ \ CPU\ \ Utilization = \frac{\sum_{i=0}^{n-1} CPUUtilizationP_i}{n}$$

Display the results to the thousandths (three places after the decimal).
Average Waiting Time: ...
Average Turnaround Time: ...
Average CPU Utilization: ...

Point Breakdown:

- Reading input and doing output format correctly. [20 pts]
- Correct calculations of the 7 required performance outputs. [30 pts]
- Correct calculations of time slice and priority. [30 pts]
- Correctly printing the time of significant events as specified in the output specifications. [60 pts]
- Good coding style, your main method should be short(around 100 lines with comments) [15 pts]
- Good use of Object Oriented Programming [15 pts]