IT308: Operating System (Winter 2012)
Programming Assignment #1: Scheduling Algorithm
Due: February 9@17hrs.

Objective: The objective of this assignment is to evaluate CPU scheduling algorithms

Scheduling Algorithm:
There two different scheduling algorithms that should be written within a single module that is distinguishing specific algorithm through the usage of specific switch

1. The program will implement multilevel feedback queues where rise in Ready queue is exponential. As with Round Robin (RR), each process runs until it completes it time slice as per quantum value, blocks for disk I/O, terminates, a disk I/O completes or another job arrives. Any time a process is interrupted it is placed back in the end of the queue, for correct priority. There will be total 8 priority levels with smallest time slice of 10 ms. If process uses its full time slice, program should decrease its priority level and double it time slice. When process does not use its full time slice, program should increase its priority level and half the time slice.
2. Second, CPU scheduling algorithm should have STCF-P – Preemptive Shortest time completion first. This in general this algorithm minimized the average response time. (As an example, two processes, one doing 1 ms computation followed by 10 ms I/O, one doing all computation and finishing in 10 ms. Suppose we use 100 ms time slice: I/O process only runs at 1/10th speed, effective I/O time is 100 ms. Suppose we use 1 ms time slice: then compute-bound process gets interrupted 9 times unnecessarily). For this implementation you have to sort the ready queue according to how much total CPU time remains for each process. A newly arrived process or a disk I/O completing will preempt the running process.

***Important note**:* Each scheduling algorithm will be in a separate file and the functions within the two scheduling algorithm will have identical interfaces. Your scheduler will take an option or switch parameter, processed at start time, to select which scheduling algorithm to use. You will use a table of function pointers, and the values in this table will be filled in at the time you process the algorithm selection parameter.

You must have trace files that contain record of processes that have run on your system. Each line of file is the record of one process and records the name of the program that ran, starting time, amount of CPU time used and number of disk I/O operations performed. As an example each like will have following format where each pieces of information is separated by space:

```
CommandName Starttime CPUTime IOCount
```

- *CommandName* is a character string (maximum length of 10 characters) that contains the name of the program;
- *StartTime* is the time in 10 millisecond increments (100ths of a second) since midnight - this is the time that the program arrived in the system;
- *CPUTime* is the total CPU time, in *seconds*, used by this program;
- *IOCount* records the total number of bytes of disk I/O done by this program. Disk I/O always occurs in full blocks; blocks are 8K (8192 bytes). We will ignore all other types of I/O (such as network or keyboard/display).

Software Design Criteria

:

Program should run in continuous loop that reads trace records and advances in time

Event Generation

Your program will maintain a notion of current time. The "clock" in program will be a variable that holds the value of the current time. The clock tick will be 1 ms. The clock will start at time 0 and advances each time a program runs or while the CPU is idle and waiting.

Hence note that while running program process may be in one of the states – Wait, Run, Ready or Termination.

- For each step (clock) in the program, there are four major operations can occur. These operations should be handled in the following order:
  1. Take the currently running process out of execution (context switch);
  2. Handle the completing disk I/O operation;
  3. Start a new disk I/O operation;
  4. Handle the newly arriving process(es).

Note that not all of these operations may occur at each clock.

- If a trace record has a CPU time field of zero, then the process must be dispatched once, with no running time (but it will have a context switch in and a context switch out).
- If a trace records has zero bytes of disk I/O, then no disk I/O will be performed by that process.

Programming Details

Here are some important details. Please pay careful attention to these items.

1. For both versions, a context switch takes 1 ms: Taking a process out of execution takes 1 ms and starting a process to execute also takes 1 ms.
2. When a process does an I/O operation, it blocks until the operation is completed.
3. Each process will perform a certain number I/O operations based on the *IOCount* field of its trace record. Since, I/O is always done in blocks on 8K, you round **up** IOCount to the next multiple of 8K:

```
IOOperations = trunc ((IOCount + 8191) / 8192)
```

4. You will use the *IOOperations* count and the *CPUTime* field to calculate how often the process will block for I/O. Divide the value *CPUTime* field by the number of I/O operations (round to the near millisecond). Note that we are assuming that I/O operations are evenly distributed throughout the execution of a program. The I/O operation always occurs at the **end** of a CPU burst.

   If the *CPUTime* does not divide evenly by the number of I/O operations, then the last CPU burst will be smaller than the other ones and **not** be followed by a disk I/O.

   If the number of I/O operations is greater than the number of milliseconds of *CPUTime*, than the excess I/O operations will all be done at the end of the process (with no extra context switches between each operation).

   Some examples:

   - If the *CPUTime* is 20 ms and the number of I/O operations is 4, then the process will need to start an I/O operation after each 5 ms of execution. So, the process will execute 5 ms, and then do an I/O, execute another 5 ms, then do an I/O, and so on.
   - If the *CPUTime* is 23 ms and the number of I/O operations is 4, then the process will execute exactly as the above case, with an additional 3 ms CPU burst after the last disk I/O.
   - If the *CPUTime* is 5 ms and the number of I/O operations is 10, then the process will start one I/O operation after each 1 ms of execution, with 6 I/O operations being done together at the end of the last CPU burst.
5. Disk I/O operations take exactly the same amount of time: 10 ms each. Your computer has one disk and can do only one disk operation at a time. As soon as one operation is completed, the next can start (with no time in between).
6. If the last thing a process does is a disk I/O, it does not have to go back into execution before it terminates. After the last I/O completes, the process is done. There are no extra context switches.
7. If a process has multiple disk I/O's to do at the end, they get do not happen all together. For example, if there are three remaining at the end of the process' execution, the process goes into the queue for 10 ms, as usual. When this first I/O completes, it will go to the end of the queue and wait to do the second I/O. This sequence continues until all the I/O operations are completed.
8. For each step (clock) in the program, there are four major operations can occur. These operations should be handled in the following order:
   1. Take the currently running process out of execution (context switch);
   2. Handle the completing disk I/O operation;
   3. Start a new disk I/O operation;
   4. Handle the newly arriving process(es).

   Note that not all of these operations may occur at each clock.

9. If a trace record has a CPU time field of zero, then the process must be dispatched once, with no running time (but it will have a context switch in and a context switch out).
10. If a trace records has zero bytes of disk I/O, then no disk I/O will be performed by that process.

Performance Evaluation

As per discussed in lecture session your program must provide performance evaluation. Hence your program will keep trace of several performance statistics and print out these results when it completes. These statistics are:

1. Average Completion Time (ACT): For each job, you will calculate how it much time it took to run. The time is the difference between its completion time and arrival time. The ACT is the average of this value for all jobs in the trace file.
2. Minimum and Maximum Completion Time: You will also compute the minimum and maximum completion times over all the jobs.
3. Throughput: This is the number of jobs per second. Divide the number of jobs that were executed by total running time of the program.
4. Utilization: This is the amount time spent doing useful computation. It does not include idle time or time spent doing context switches. Print out the total and as a percentage of the running time of the program.

Hint: Good design on this assignment will save you, literally thousands of lines of code. A crucial module in each version of your program will be the one that does the queuing. In one version of the program, it is a priority queue sorted by one of 8 priority levels. In the other version, it is a priority queue sorted by remaining CPU time.

All other parts of your program should be the same, so you can re-use them for the different versions scheduler program or you can write such part as generic sub-module that is used by both versions of your schedulers.

You have plenty of time for this assignment, but don't delay in getting started! Start work early on a design and initial structure for your modules and functions.

Preliminary Review

Before you start coding, you should develop a design for your program. This design should include a description of each of the major modules (which should each be separate source code files). The description of a module should include:

1. A brief text description of what the module does.
2. A first-pass at the declarations of the major data structures implemented by the module.
3. The external functions (the C equivalent of the public methods). For each method, you should include:

    - The function name.
    - A description of the parameters and return value.
    - A description of what the function does.
    - If the algorithm in the function is at all complicated, then a description of how the algorithm works.
4. A box diagram of how the modules fit together (how they interact with each other.

You will meet with TA to present your design and to get feedback. This is an essential step in your design and **must** be done for your program to be acceptable.

Deliverable:

You should hand your program, README, and Makefile as a part of this assignment electronically. You should include a copy of the code for **each** scheduling algorithm, and one copy of the code for the rest of the program. These should be clearly labeled (commented) as to what they do.

Your program output should print out the statistics described above. You should also hand in a text file (HARDCOPY) for the output from each required run of the program. The README file should describe each of these files.

There are test files provided in lab folder that you have to consider for testing.

Test files for this assignment can be found in the lab directory under the folder - IT308_2012

Currently, there three files in this directory, `pa1data.small`, `pa1data.medium`, and `pa1data.big`.

You can use the two smaller files for initial testing, but do not turn in results for these two files. You should use file `p4data.big` for results that you turn in.

You will upload your deliverable with to FTP server. Details of ftp server are given below. You can start to upload deliverable 2 days in prior to deadline, and not early to make ftp server usage more consistent.

There is four folder in IT_308_2012 (This is share folder) batch1, batch2, batch3, batch4 each folder 32 student. Must have your folder under relevant batch with name *your.sid_pa1* for final evaluation.

\\10.100.56.24\IT_308_2012