Programming Assignment #2 (Lab 2): Scheduler / Dispatcher **Professor Hubertus Franke**
Class CSCI-GA.2250-001 Summer 2012

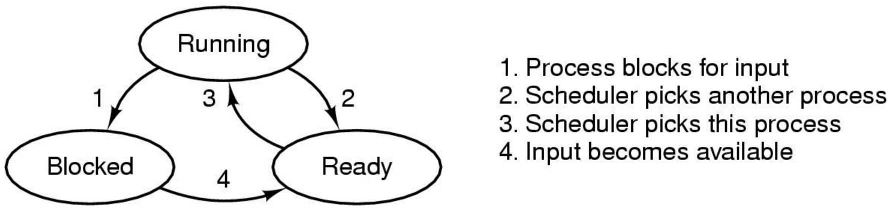
Due July 11th 2012 http://cs.nyu.edu/courses/summer12/CSCI.2250-001/labs/lab2/

You are to implement a Scheduler in C, C++, or Java and submit the **source** code, which we will compile and run. Both are to be delivered to the TA as described in the above link.

In this lab we explore the effects of different scheduling policies discussed in class on a set of processes executing on a system. The system is to be implemented as a Discrete Event Simulation. This implies that the system progresses in time through defining and executing the events (state transitions) and by progressing time discretely between the events.

A process is characterized by 4 parameters:
Arrival Time (AT), Total CPU Time (TC), CPU Burst (CB) and IO Burst (IO).

Each process follows the following state diagram that should also be used to guide you in defining the events (state transitions):


Running
Blocked
Ready
1
2
3
4
1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Both the CPU and the IO bursts are statistically defined. When a process is scheduled (becomes READY (transition 4)) the CPU burst is defined as a random number between (0,CB]. If the remaining execution time is smaller than the random number chosen, reduce the random number to the remaining execution time. Equally when a process finishes its current CPU burst (assuming it has not yet reached its total CPU time), it enters into a period of IO (transition 1) at which point the IO burst is defined as a random number between (0,IO].
We make a few simplification:

(a) all time is based on integers not float, hence nothing will happen or has to be simulated between integer numbers;
(b) to enforce a uniform repeatable behavior, a file with random numbers is provided (see website) that your program must read in and use (note the first line defines the count of random numbers in the file) a random number is then created by using:
“int myrandom(int burst) { return 1 + (randvals[ofs] % burst); }”

You should increase ofs with each invocation and wrap around when you run out of numbers in thefile/array. It is therefore important that you call the random function only when you have, namely for transitions 1 and 4.
(c) IOs are independent from each other, i.e. they can commensurate concurrently without affecting each others IO burst time.

The input file provides a separate process specification (see above) in each line: AT TC CB IO
You can make the assumption that the input file is well formed and that the ATs are not decreasing. It is however possible that multiple processes have the same arrival times.

The scheduling algorithms to be simulated are:
FCFS, RoundRobin, LCFS, SJF. In Round Robin your program should accept the quantum as an input (see below “*Execution and Invocation Format*”). The context switching overhead is “0”.

A few things you need to pay attention to:
Round Robin: you should only regenerate a new CPU burst, when the current one has expired.
SJF: schedule is based on the shortest remaining execution time, not shortest CPU burst.

Output:

At the end of the program you should print the following information and the example outputs on the web provide the proper expected formatting (including precision) (this is necessary to automate the results checking:

a) Scheduler information (which scheduler algorithm and if RR the quantum)
b) Per process information (see below):
for each process (assume process start with pid=0), the correct desired format is shown on the web for various examples:
pid: AT TC CB IO | FT TT IT CW
FT: Finishing time
TT: Turnaround time (finishing time - AT)
IT: I/O Time (time in blocked state)
CW: CPU Waiting time (time in Ready state)
c) Summary Information - Finally print a summary for the simulation:
Finishing time of the last event (i.e. the last process finished execution)
CPU utilization (i.e. percentage (0.0 – 100.0) of time at least one process is running
IO utilization (i.e. percentage (0.0 – 100.0) of time at least one process is performing IO
Average turnaround time among processes
Average cpu waiting time among processes
Throughput of number processes per 100 time units

CPU / IO utilizations and throughput are computed from time=0 till the finishing time.

```
Example:
FCFS
0000:    0  100   10   10 |  223  223  123    0
0001:   500  100   20   10 |  638  138   38    0
SUM:    638 31.35 25.24 180.50 0.00 0.313
```

You must strictly adhere to this format. The programs results will be graded by a testing harness that uses “diff –b”. In particular you must pay attention to separate the tokens and to the rounding. In the past we have noticed that different runtimes (C vs. C++ vs. Java) use different rounding. For instance 1/3 was rounded to 0.334 in one environment vs. 0.333 in the other (similar 0.666 should be rounded to 0.667).
My reference program uses: printf("SUM: %d %.2f %.2f %.2f %.2f %.3f\n", float_variables …).
In C++ and Java you must specify the precision and the rounding behavior.

If in doubt, here is a small C program (gcc) to test your behavior (you can transfer to Java and C++) and verify:

```
#include <stdio.h>

main()
{
   float a,b;
   a = 1.0/3.0;
   b = 2.0/3.0;
   printf("%.2f %.2f\n",  a, b);
   printf("%.3f %.3f\n",  a, b);
}
```

Should produce the following output

```
0.33 0.67
0.333 0.667
```

Deterministic Behavior:

There will be scenarios where events will have the same time stamp and you must follow these rules to break the ties in order to create consistent behavior:

(a) On the same process: termination takes precedence over scheduling the next IO burst over preempting the process on quantum expiration
(b) Processes with the same arrival time should be entered into the ready queue in the order of their occurrence in the input file.
(c) Events with the same time stamp (e.g. IO completing at time X for process 1 and cpu burst expiring at time X for process 2) should be processed in the order they were generate, i.e. if the IO start event (process 1 blocked event) occurred before the event that made process 2 running) then the IO event should be processed first.
(d) You must process all events at a given time stamp before invoking the scheduler/dispatcher.

Submitting the lab: please submit to the TA as described on the web and done in Lab1
Do **NOT** submit any input or output files

Execution and Invocation Format:
Your program should follow the following invocation:
<program> [-v] [-s<schedspec>] inputfile randfile

The test input files and the sample file with random numbers are on the website.
The scheduler specification is "–s [FLS | R<num>]" where F=FCFS), L=LCFS, S=SJF and R10 is RR with quantum 10. (e.g. *"./sched –v –sR10"*). Supporting this parameter is required. If not provided to the program execution, the default shall be FCFS.

The –v option stands for verbose and should print out some tracing information that allows one to follow the state transition. Though this is **not** mandatory, this is highly suggested you build this into your program to allow you to follow the state transition and to verify the program as will be discussed in class. I include samples from my tracing. Matching my format will allow you to run diffs and identify why results and where the results don't match up.

Two scripts "runit.sh" and "diffit.sh" are provided that will allow you simulate the grading process. "runit.sh" will generate the entire output files and "diffit.sh" will compare with the outputs supplied.

Please ensure the following:

(a) The input and randfile must accept any path and should not assume a specific location relative to the code ore executable.

(b) **All code/grading will be executed on machine <linax1.cims.nyu.edu>** to which you can log in using "ssh <userid>@linax1.cims.nyu.edu". You should have an account by default. Please try that immediately.
This is the only way we can guarantee that we execute the right version of compiler (C/C++) and virtual machines (JVM). It is your responsibility to test your code on linax1.

If the code doesn't compile or does not run or does not allow any path (relative or absolute), the code will be returned with appropriate score reduction and additional reduction for the resulting delays.

As always, if you detect errors in the sample inputs and outputs, let me know immediately so I can verify and correct if necessary.