Programming Assignment #2 (Lab 2): Scheduler / Dispatcher **Professor Hubertus Franke**
Class CSCI-GA.2250-001 Spring 2020

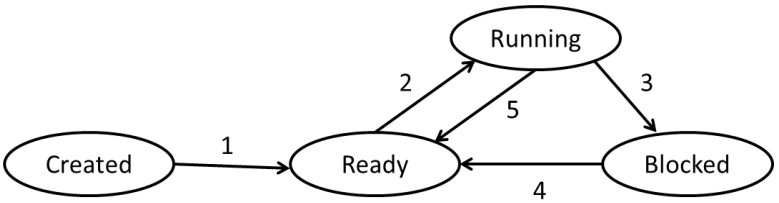
In this lab we explore the implementation and effects of different scheduling policies discussed in class on a set of processes/threads executing on a system. The system is to be implemented using Discrete Event Simulation (DES) **(http://en.wikipedia.org/wiki/Discrete_event_simulation)**. In discrete-event simulation, the operation of a system is represented as a chronological sequence of events. Each event occurs at an instant in time and marks a change of state in the system. This implies that the system progresses in time through defining and executing the events (state transitions) and by progressing time discretely between the events as opposed to incrementing time continuously (**e.g. don't do *"sim_time++"***). Events are removed from the event queue in chronological order, processed and might create new events at the current or future time. Note that DES has nothing to do with OS, it is just an awesome generic way to step through time and simulating system behavior that you can utilize in many system simulation scenarios.

Note that, you are not implementing this as a multi-program or multi-threaded application. By using DES, a process is simply the PCB object that goes through discrete state transitions. In the PCB object you maintain the state and statistics of the process as any OS would do. In reality, the OS doesn't get involved during the execution of the program (other than system calls), only at the scheduling events and that is what we are addressing in this lab.

Any process essentially involves processing some data and then storing / displaying it (on Hard drive, display etc). (A process which doesn't store/display processed information is practically meaningless). For instance: when creating a zip file, a chunk of data is first read, then compressed, and finally written to disk, this is repeated until all of the file is compressed. Hence, an execution timeline of any process will contain discrete periods of time which are either dedicated for processing (computations involving CPU aka cpu_burst) or for doing IO (aka ioburst). For this lab assume that our system has only 1 CPU core without hyperthreading - meaning that only 1 process can run at any given time; and that all processes are single threaded - i,e, they are either in compute/processing mode or IO mode. These discrete periods will therefore be non-overlapping. There could be more than 1 process running (concurrently) on the system at a given time though, and a process could be waiting for the CPU, therefore the execution timeline for any given process can/will contain 3 types of non-overlapping discrete time periods representing (i) Processing / Computation, (ii) Performing IO and (iii) Waiting to get CPU.

The simulation works as follows:
Various processes will arrive / spawn during the simulation. Each process has the following 4 parameters:
1) Arrival Time (AT) - The time at which a process arrives / is spawned / created.
2) Total CPU Time (TC) - Total duration of CPU time this process requires
3) CPU Burst (CB) – A parameter to define the upper limit of compute demand (further described below)
4) IO Burst (IO) - A parameter to define the upper limit of I/O time (further described below)

The processes during its lifecycle will follow the following state diagram :



Initially when a process arrives at the system it is put into CREATED state. The processes' CPU and the IO bursts are statistically defined. When a process is scheduled (becomes RUNNING (transition 2)) the *cpu_burst* is defined as a random number between [1 .. CB]. If the remaining execution time is smaller than the *cpu_burst* compute, reduce it to the remaining execution time. When a process finishes its current *cpu_burst* (assuming it has not yet reached its total CPU time TC), it enters into a period of IO (aka BLOCKED) (transition 3) at which point the *io_burst* is defined as a random number between [1 .. IO]. If the previous CPU burst was not yet exhausted due to preemption (transition 5), then no new *cpu_burst* shall be computed yet in transition 2 and you continue with the remaining cpu burst.

The scheduling algorithms to be simulated are:
FCFS, LCFS, SRTF, RR (RoundRobin), PRIO (PriorityScheduler) and PREemptive PRIO (PREPRIO). In RR, PRIO and PREPRIO your program should accept the *time quantum* and for PRIO/PREPRIO optionally the number of priority levels *maxprio* as an input (see below "*Execution and Invocation Format*"). We will test with multiple *time quantums* and *maxprios*, so do not make any assumption that it is a fixed number. The context switching overhead is "0".

You have to implement the scheduler as "objects" without replicating the event simulation infrastructure (event mgmt or simulation loop) for each case, i.e. you define one interface to the scheduler (e.g. *add process(), get next process()*) and

implement the schedulers using object oriented programming (inheritance). The proper "scheduler object" is selected at program starttime based on the "-s" parameter. The rest of the simulation must stay the same (e.g. event handling mechanism and *Simulation()*). The code must be properly documented. When reading the process specification at program start, always assign a static_priority to the process using myrandom(*maxprio*) (see above) which will select a priority between 1..*maxprio*. A process's dynamic_priority is defined between [0 .. (static_priority-1)]. With every quantum expiration the dynamic_priority decreases by one. When "-1" is reached the prio is reset to (static_priority-1). Please do this for all schedulers though it only has implications for the PRIO/PREPRIO schedulers as all other schedulers do not take priority into account. However uniformly calculating this will enable a simpler and scheduler independent state transition implementation.

A few things you need to pay attention to:
All: When a process returns from I/O its dynamic_priority is reset (to (static_priority-1).
Round Robin: you should only regenerate a new CPU burst, when the current one has expired.
SRTF: schedule is based on the shortest remaining execution time, not shortest CPU burst and is non-preemptive
PRIO/PREPRIO: same as Round Robin plus: the scheduler has exactly *maxprio* priority levels [0..*maxprio-1*], *maxprio-1* being the highest. Please use the concept of an active and an expired runqueue and utilize independent process lists at each prio level as discussed in class. When "-1" is reached the process's dynamic_priority is reset to (static_priority-1) and it is enqueued into the expired queue. When the active queue is empty, active and expired are switched.
Preemptive Prio (E) refers to a variant of PRIO where processes that become active will preempt a process of lower priority.
Remember, runqueue under PRIO is the combination of active and expired.

Input Specification

The input file provides a separate process specification 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. So no fancy parsing is required. It is possible that multiple processes have the same arrival times. Then the order at which they are presented to the system is based on the order they appear in the file.
Simply, for each input line (process spec) create a process object, create a process-create event and enter this event into the event queue. Then and only then start the event simulation. Naturally, when the event queue is empty the simulation is completed.

We make a few simplifications:

(a) all time is based on integers not floats, 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 NYU classes attachment) 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 (don't make assumptions about the number of random numbers):
`"int myrandom(int burst) { return 1 + (randvals[ofs] % burst); }"` // yes you can copy the code

You should increase ofs with each invocation and ***wrap around*** when you run out of numbers in the file/array. It is therefore important that you call the random function only when you have to, namely for transitions 2 and 3 (with noted exceptions) and the initial assignment of the static priority.

(c) IOs are independent from each other, i.e. they can commensurate concurrently without affecting each other's IO burst time.

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

Options should be able to be passed in any order. This is the way a good programmer will do that.
http://www.gnu.org/software/libc/manual/html_node/Example-of-Getopt.html

Test input files and the sample file with random numbers are available as a NYU classes attachment.
The scheduler specification is "–s [FLS | R<num> | P<num>[:<maxprio>] | E<num>[:<maxprios>]]", where F=FCFS, L=LCFS, S=SRTF and R10 and P10 are RR and PRIO with quantum 10. (e.g. *"./sched –sR10"*) and E10 is the preemptive prio scheduler. Supporting this parameter is required and the quantum is a positive number. In addition the number of priority levels is specified in PRIO and PREPRIO with an optional ":num" addition. E.g. "-sE10:5" implies quantum=10 and numprios=5. If the addition is omitted then maxprios=4 by default (lookup : `sscanf(optarg, "%d:%d",&quantum,&maxprio)`)

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, it is highly suggested you build this into your program to allow you to follow the state transition and to verify the program. I include samples from my tracing for some inputs (not all). Matching my format will allow you to run diffs and identify why results and where the results don't match up. You can always use /home/frankeh/Public/sched to create your own detailed output for not provided samples. Also use -t and -e options.

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

Please ensure the following:

(a) The input and randfile must accept any path and should not assume a specific location relative to the code or executable.
(b) All output must go to the console (due to the harness testing)
(c) **All code/grading will be executed on machine <linserv1.cims.nyu.edu>** to which you can log in using "ssh <userid>@linserv1.cims.nyu.edu". You should have an account by default, but you might have to tunnel through access.cims.nyu.edu.

As always, if you detect errors in the sample inputs and outputs, let me know immediately so I can verify and correct if necessary. Please refer the input/output file number and the line number.

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) Processes with the same arrival time should be entered into the run queue in the order of their occurrence in the input file.
(b) On the same process: termination takes precedence over scheduling the next IO burst over preempting the process on quantum expiration.
(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 generated, i.e. if the IO start event (process 1 blocked event) occurred before the event that made process 2 running (naturally has to be) then the IO event should be processed first. If two IO bursts expire at the same time, then first process the one that was generated earlier.
(d) You must process all events at a given time stamp before invoking the scheduler/dispatcher (See *Simulation()* at end).

Not following these rules implies that fetching the next random number will be out of order and a different event sequence will be generated. The net is that such situations are very difficult to debug (see for relieve further down).

ALSO:

Do not keep events in separate queues and then every time stamp figure which of the events might have fired. E.g. keeping different queues for when ~~various I/O will complete~~ vs a ~~queue for when new processes will arrive~~ etc. will result in incorrect behavior. There should be effectively two logical queues:

1. An event queue that drives the simulation and
2. the run queue/ready queue(s) [same thing] which are implemented inside the scheduler object classes.

These queues are independent from each other. In reality there can be at most one event pending per process and a process cannot be simultaneously referred to by an event in the event queue and be referred to in a runqueue (I leave this for you to think about why that is the case). Be aware of C++ build-in container classes, which often pass arguments by value. When you use queues or similar containers from C++ for process object lists, the object will most likely be passed by value and hence you will create a new object. As a result you will get wrong accounting and that is just plain wrong. There should only be one process object per process in the system. To avoid this, make queues of process pointers (queue<Process*>).

Output

At the end of the program you should print the following information and the example outputs provide the proper expected formatting (including precision); this is necessary to automate the results checking; all required output should go to the console (stdout / cout).

a) Scheduler information (which scheduler algorithm and in case of RR/PRIO/PREPRIO also the quantum)
b) Per process information (see below):
   for each process (assume processes start with pid=0), the correct desired format is shown below:
   pid: AT TC CB IO PRIO | FT TT IT CW
   FT: Finishing time
   TT: Turnaround time (finishing time - AT)
   IT: I/O Time (time in blocked state)
   PRIO: static priority assigned to the process (note this only has meaning in PRIO/PREPRIO case)
c) CW: CPU Waiting time (time in Ready state)
d) 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 0 |  223  223  123    0
0001:    500  100   20   10 0 |  638  138   38    0
SUM:     638 31.35 25.24 180.50 0.00 0.313
```

You must **strictly** adhere to this format. The program's 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++) 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). **Always use double** (instead of float) variables where non-integer computation occurs. See *outformat.c* in assignment file. In C++ you must specify the precision and the rounding behavior. See examples in */home/frankeh/Public/ProgExamples/Format/format.cpp* as discussed in extra session.

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

```
#include <stdio.h>
main()
{
    double a,b;
    a = 1.0/3.0;
    b = 2.0/3.0;
    printf("%.2lf %.2lf\n",  a, b);
    printf("%.3lf %.3lf\n",  a, b);
}
```

Should produce the following output
0.33 0.67
0.333 0.667

Use the following printf's (or design your equivalents for C++) to print out the per-process and summary report.
See C++ examples in ~frankeh/Public/ProgExamples.tz (Format subdirectory for C and C++).

```
printf("%04d: %4d %4d %4d %4d %1d | %5d %5d %5d %5d\n",
printf("SUM: %d %.2lf %.2lf %.2lf %.2lf %.3lf\n",
```

note " %4d %4d" is not equivalent to "%5d%5d" .. this is often a source of problems.