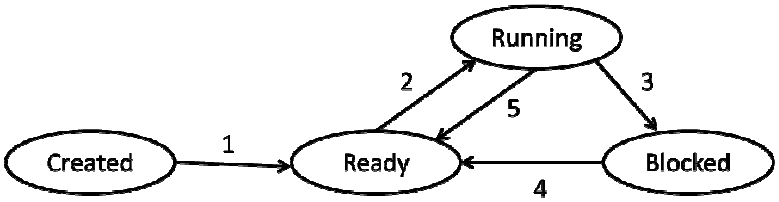
Programming Assignment #2 (Lab 2): Scheduler / Dispatcher **Professor Hubertus Franke**
Class CSCI-GA.2250-001 Spring 2017

You are to implement a Scheduler in C, C++ (only) and submit the **source** code, which we will compile and run. Both are to be delivered to the TA as source code through NYU Classes assignment. The submission must include a *Makefile* for building your code. Please no inputs and outputs (we already got them).

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 as a 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++"***). Timestamped events are removed from the event queue and new events might be created during the simulation.

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 be used to guide you in defining the events (state transitions):



Initially when a process arrives at the system it is put into CREATED mode. Both the 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 random number chosen, reduce the random integer number to the remaining execution time. 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 (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, then no new CPU burst has to be computed yet in transition 5.

Note, you are not implementing this as a multi-program or multi-threaded application. By using DES, a process is simply an object that goes through discrete state transitions. In the object you maintain the state and statistics of the process as the OS would do.

We make a few simplifications:

(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 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:
"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 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 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. 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, create a process start event for all processes in the input file and enter these events into the event queue. Then and only then start the event simulation. Naturally, when the event queue is empty the simulation is completed.

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

You have to implement the scheduler as "objects" without replicating the event simulation infrastructure for each case, i.e. you define one interface to the scheduler/dispatcher (e.g. put_event/get_event) and implement the schedulers using object oriented programming. The proper "scheduler object" is selected at program starttime based on the "-s" parameter. The rest of the simulation must stay the same. The code must be properly documented. When reading in the process specification, assign a static_priority to the process using myrandom(4) (see above) which will select a priority between 1..4. 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 scheduler 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:
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.
PRIO: same as Round Robin plus: the scheduler has exactly 4 priority levels [0..3], 3 the highest. Please use the concept of an active and an expired queue 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. Remember runqueue under PRIO is the combination of active and expired.
All: When a process returns from I/O its dynamic priority is reset (to (static_priority-1).

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).

a) Scheduler information (which scheduler algorithm and if RR 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 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 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++) 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). **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.

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.

```
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%d" .. this is often a source of problems.
```

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 run 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 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.

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 in independent from each other.
Be aware of C++ build in container classes, which often pass arguments by value. When you use lists 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. There should only be one process object per process. To avoid this make lists of process pointers (list<Process*>).

Programming Assignment #2 (Lab 2): Scheduler / Dispatcher **Professor Hubertus Franke**
Class CSCI-GA.2250-001 Spring 2017

Submitting the lab: please submit to your code including a Makefile via NYU Classes.
Please, do **NOT** submit any input or output files (we already have them or will generate them)

Execution and Invocation Format:
Your program should follow the following invocation:
<program> [-v] [-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 on the website.
The scheduler specification is "–s [FLS | R<num> | P<num>]", where F=FCFS, L=LCFS, S=SJF and R10 and P10 are RR and PRIO with quantum 10. (e.g. *"./sched –sR10"*). Supporting this parameter is required.

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. 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 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 <energon1.cims.nyu.edu>** to which you can log in using "ssh <userid>@energon1.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.

Reference Program:

The reference program used for grading is accessible on CIMS account under */home/frankeh/Public/sched* and you can run inputs against it to determine whether your output matches or not if you want to go beyond the provided inputs/outputs.

Scoring and deductions:

We score this lab as 100pts. You will receive 30 pts for a submission that attempts to solve the problem. The rest you get 70/N points for each successful test that passes the "*diff*". In order to institute a certain software engineering discipline, i.e. following a specification and avoiding unintended releases of code and data in real life, we account for the following additional deductions:

Reason	Deduction	How to avoid
Makefile not working on CIMS or missing.	2pts	Just follow instructions above or see lab1.
Late submission	2pts/day	Upto 7 days. After which reach out to me or TA but work on next lab (don't fall behind).
Inputs/Outputs or *.o files in the submission	1pt	Go through your intended submission and clean it up.
Output not going to the screen but to a file	1pt	We utilize the output to <stdout> during the runit.sh and gradeit.sh so just use printf or cout.
Replicating Event and Simulation per scheduler	6 pts	Use object oriented coding style and code fragments at the end for the simulation.

Explanation of the verbose output:

Two examples of an event in my trace

Example 1:

57 0 12: BLOCK -> READY

At timestamp 57 process 0 is going from BLOCKED into READY state. The process has been in its current state for 12 units (hence it must have been BLOCKED at time 45).

Example 2:

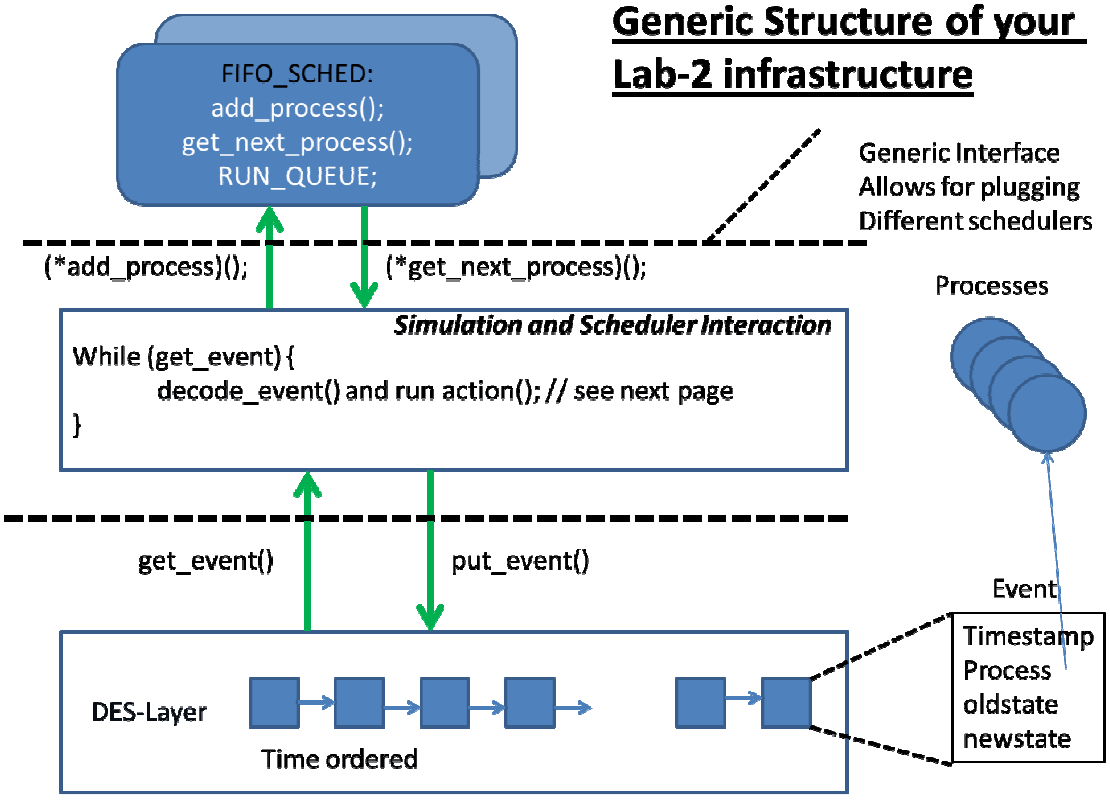
42 2 7: RUNNG -> BLOCK ib=3 rem=77

At time unit 42 the transition for process 2 to BLOCKED state is executed.
It was in RUNNING state since time timeunit 35.
The IO burst created is ib=3 and there remains 77 time units (rem=77) left for executing this job.

By providing this extended output you will be able to create a detailed trace for your execution and compare it to the reference and identify where you start to differ. At a point of difference you should see which rule potentially was deployed that choose a different job/event in the reference vs. your program.

Some hints on structuring your program.

The structure / modules of your program should look something like the following.



Programming Assignment #2 (Lab 2): Scheduler / Dispatcher **Professor Hubertus Franke**
Class CSCI-GA.2250-001 Spring 2017

Start by reading in the input file and creating Process objects.
Then program a generic DES Layer, which basically means you need to be able to create events, that take the timestamp when it is supposed to fire, a pointer to the Process (don't pass by value, as there can only be ONE object related a process, otherwise your accounting will be incorrect) and the state you want to transitions to (see diagram). Make sure when you enter the event it is inserted based on the prior description. Don't use sort() functions as they are inefficient in this case, often don't fit the problem (know your stable vs instable sort behavior otherwise) and are simply overkill (e.g. use insert_sort()).

Next implement ONE scheduler (suggest you start with FIFO as we might not have covered all schedulers by the time of handing out the problem). Implement the schedulers as a class hierarchy (C++ or see me how to do this in C). Note from the code fragments below, the Simulation() should not know any details about the specific scheduler itself, so all has to be accomplished through *virtual* functions. One trick to deal with schedulers is to treat non-preemtive scheduler as preemptive with very large timeout (10K is good for our simulation) that will never fire. This way the *TRANS_TO_RUN* transition is implemented generically.

After you have created the process objects and after you have put initial events for processes's arrival into the even queue, the simulation can start. The simulation code structure will look something like below (very sketchy). Note that runqueue/readyqueue has nothing todo with event queue, they are completely different entities.

Also based on the sketchy code, note that the simulation knows no details about the run/ready queue or other details from the scheduler. It simply adds process to the runqueue (transitions 1,4,5) or ask the scheduler for the next process to run (there might not be one, at which point the scheduler returns NULL).

```
void Simulation() {
        EVENT* evt;
        while( (evt = get_event()) ) {
                CURRENT_TIME = evt->evtTimeStamp;
                evt->evtProcess->timeInPrevState = CURRENT_TIME - evt->evtProcess->state_ts;

                switch(evt->transition) {  // which state to transition to?
                case TRANS_TO_READY:
                         // must come from BLOCKED or from PREEMPTION
                         // must add to run queue
                        CALL_SCHEDULER = true; // conditional on whether something is run
                        break;
                case TRANS_TO_RUN:
                        // create event for either preemption or blocking
                        break;
                case TRANS_TO_BLOCK:
                        //create an event for when process becomes READY again
                        CALL_SCHEDULER = true;
                        break;
                case TRANS_TO_PREEMPT:
                        // add to runqueue (no event is generated)
                        CALL_SCHEDULER = true;
                        break;
                }
                //remove current event object from Memory
                delete evt;

                if(CALL_SCHEDULER) {
                        if (get_next_event_time() == CURRENT_TIME) {
                                continue;//process next event from Event queue
                        }
                        CALL_SCHEDULER = false;
                        if (CURRENT_RUNNING_PROCESS == nullptr) {
                                CURRENT_RUNNING_PROCESS = THE_SCHEDULER->get_next_process();
                                if (CURRENT_RUNNING_PROCESS == nullptr) {
                                        continue;
                                }
                                // create event to make process runnable for same time.
                        }
                }
        }
}
```