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

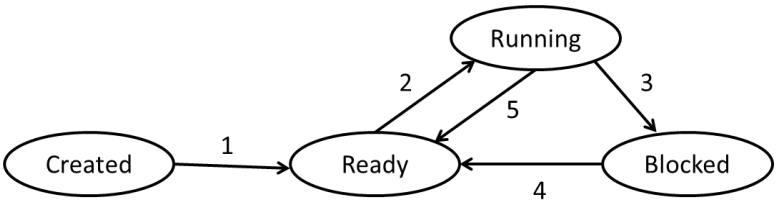
In this lab we explore the implementation and effects of different scheduling policies discussed in class on a set of processes/threads executing on a system. The system is to be implemented using Discrete Event Simulation (DES) **(http://en.wikipedia.org/wiki/Discrete_event_simulation)**. In discrete-event 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.

What to submit, scoring and deductions:

Submit only your source code (C/C++) along with the makefile and a readme if compilation is not straightforward.

We score this lab as 100pts. You will receive 40 pts for a submission that attempts to solve the problem. The rest you get 60/N points for each successful test that passes the "*diff*". Due to the difference of complexity, F,R,S scheduler are 1/13 each, RR is 3/13 and PRIO is 4/13 and PREPRIO is 3/13 (of the 60 points). 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 please 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 (you will have to fix this)	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.

Additional Useful Stuff

Reference Program:

The reference program used for grading is accessible on my 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.

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).

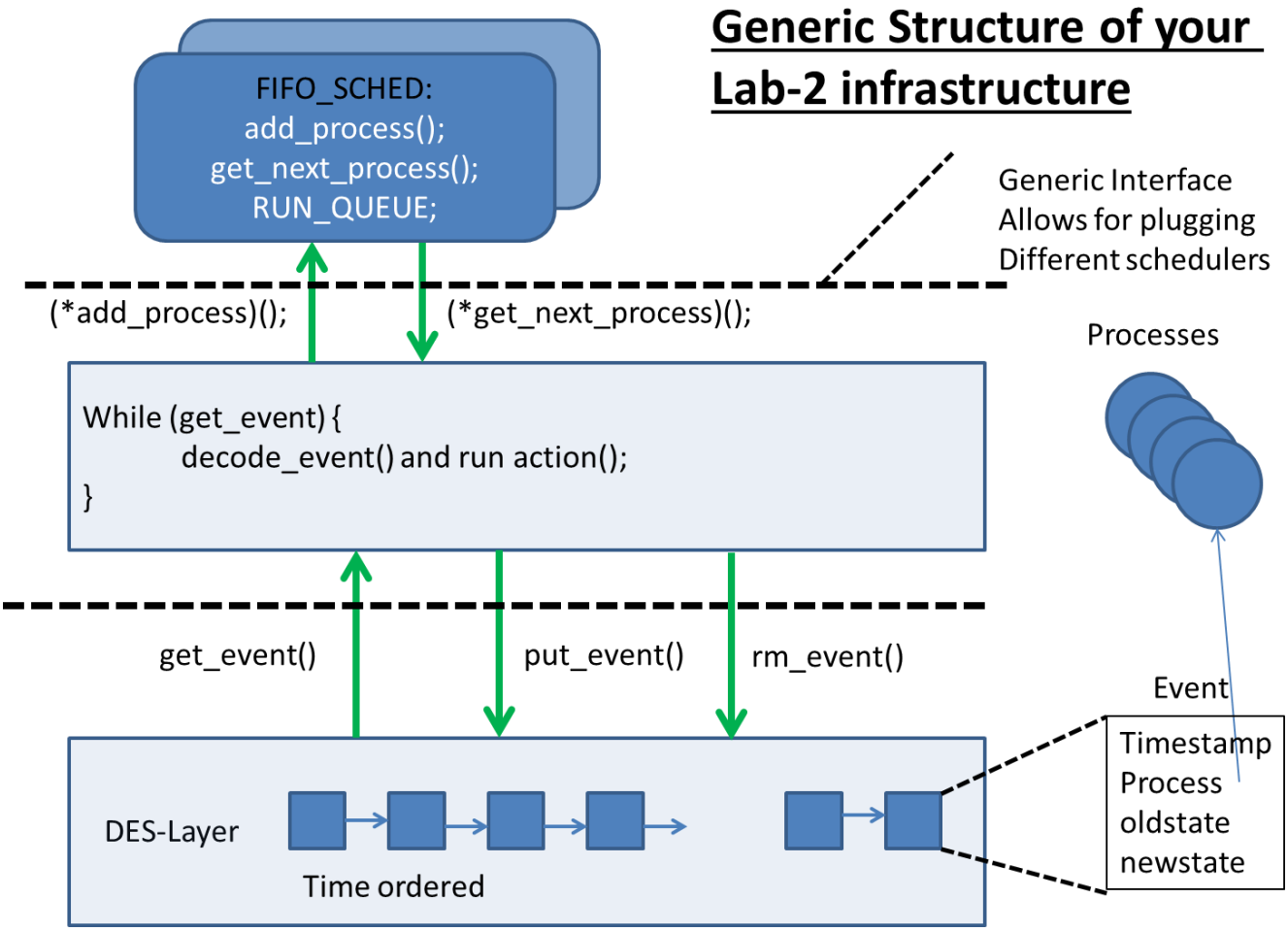
Example 2: 42 2 7: RUNNG -> BLOCK ib=3 rem=77

At time unit 42 the transition for process 2 to BLOCKED state is executed and it was in RUNNING state for 7 units.
It was in RUNNING state since time timeunit 35 (derived from42 – 7)
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 suggestions on approaching the problem and on structuring your program:

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



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 to 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()). Its best to create the DES layer first in isolation and use <integers> instead of Processes. Then write a small program and insert different <integers> with same and different timestamps in different orders. Print out the sequence of events to ensure you really process events in chronological order following the specification. If this is wrong you will be debugging to no end.

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 for C see *~frankeh/Public/ProgExamples.tz* subdir VirtFunC). 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-preemptive scheduler as preemptive with very large quantum that will never fire (10K is good for our simulation). 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 all processes's arrival into the event queue, the simulation can start. The simulation code structure will look something like below (very sketchy, after all you are supposed to write the code). Note (again) that runqueue/readyqueue has nothing to do with the event queue, they are completely different entities. One interesting thing that is different in the 'E' scheduler is that the process waking up (new/end-block) might preempt the running process if its priority is higher. In this case the future event on the running processes must be cancelled (rm_event()) and a preemption event for the current time must be generated for that process. If preempted that way, the next time the process runs it gets a full quantum again (see more details next page).

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 processes to the runqueue (transitions 1,4,5) or asks the scheduler for the next process to run (there might not be one, at which point the scheduler returns NULL). Note, this is incomplete pseudo code to serve as a framework.

```
void Simulation() {
        EVENT* evt;
        while( (evt = get_event()) ) {
                Process *proc = evt->evtProcess;    // this is the process the event works on
                CURRENT_TIME = evt->evtTimeStamp;
                timeInPrevState = CURRENT_TIME - proc->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; evt = nullptr;

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

Some other useful suggestions (read this how things are done in real programming to be readable and efficient), since learning proper programming is a side goal/directive of this class:

- Do not represent process states or transitions as strings or integers. Use enumerations. Let the compiler do the hard work. Why? Assume you want to represent states using strings (e.g. "STATE_RUNNING", "STATE_BLOCKED"). First, to store the current state in a process requires memory to hold the string which above would be 16 bytes or if you are clever could be stored in a pointer (last one not too bad). At some point you have to check a processes state and you have to code something like

  ```
  if (proc->state == "STATE_RUNNING")
  ```

  which has two problems. (a) your compiler might convert that into a string comparison OR a pointer comparison .. do you know the difference ? → debugging nightmare. (b) string compares can-be/are inefficient.

  Integers are unreadable and code can't be maintained. Have mercy on those that potentially have to deal with your code in the real world.

  ```
  if (proc->state == 1)
  ```

  What does that mean?

  Instead you would have in C/C++

```
typedef enum { STATE_RUNNING , STATE_BLOCKED }   process_state_t;
if (proc->state == STATE_RUNNING)
```

This is efficient and readable. An enumeration takes at most an integer worth (4 bytes) and the compiler converts names to integers starting with 0.

- Don't use vectors/arrays for runqueue/readyqueue. You can not efficiently add/remove at different locations, use lists or queues. (exceptions is the priority levels which should be implemented as vectors/arrays of lists/queues).

- Don't create lists or queues of processes. Queue<Process> implies that when you add a Process to a list you will make a full copy of a process. This is incorrect and will ~~potentially~~ lead to wrong implementations. There can only be ONE process object per running process. It also is inefficient, in a real OS the process object is ~4K. Instead create a process object while you are reading the input and then create Queue<Process*> for the ready/runqueue and passing pointers to that object. As a result you always point to that one process which is important for accouting and efficiency, let alone for correctness. Note Queue is just one type of collection here.

- Implement the priority scheduler as `Queue<Process*> *activeQ, *expiredQ` (really pointers to arrays of queues) and dynamically allocate the array for the required priority levels so you can add at the end of the queue and and pop from the front efficiently. This will force you to do the classical priority decay approach. Please don't use a single `PrioQueue<Process*>` for active or expired.

- The scheduler classes really have to provide only three functions:

```
void add_process(Process *p);
Process* get_next_process();
void test_preempt(Process *p, int curtime ); // typically NULL but for 'E'
```

- To make the implementation of the state transitions uniform, you can pretend that non-preemptive schedulers have a very large quantum (e.g. 10000) which essentially means that no preemption will ever occur for those schedulers.

- The 'E' scheduler is a bit tricky. When a process becomes READY from creation or unblocking it might preempt the currently running process. Preemption in this case happens if the unblocking process's dynamic priority is higher than the currently running processes dynamic priority AND the currently running process does not have an event pending for the same time stamp. The reason is that such an event can only be a BLOCK or a PREEMPT event. So we do not force a preemption at this point as the pending event will be picked up before the scheduler is called.
  If preemption does happen, you have to remove the future event for the currently running process and add a preemption event for the current time stamp (ensure that the event is properly ordered in the eventQ).

Good Luck