Due: Sun Nov 1, 2020 11:59pm

Learning outcomes

The primary focus of this project is to learn about scheduling policies and on their impact.

You will design and implement a general-purpose simulator, call we call it the coordinator, of a simple operating system. You will start from scratch; no code will be provided.

To evaluate the utility of your coordinator, and to get a notion of effective CPU scheduling policies, you will compare at least three fundamental scheduling policies: round robin, lottery scheduling and multi-level feedback scheduling.

Required scheduling policies

Multi-level feedback scheduling (aggressive)

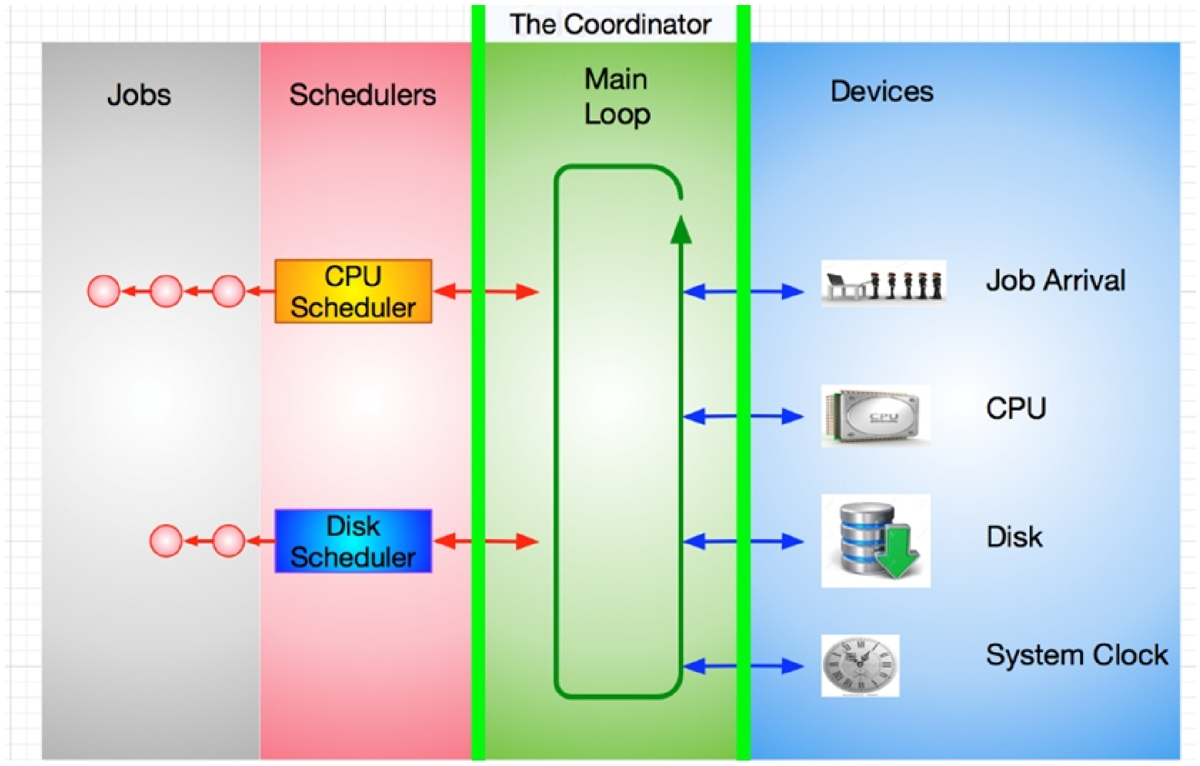
Multi-level feedback scheduling (non-aggressive)

Preemptive shortest job first.

Introduction

You will implement a system coordinator (we will sometimes call the coordinator the simulator).   The coordinator (or simulator) manages:

1. jobs,
2. devices, and
3. schedulers
4. all coordinated by a **single loop** of code (see image below).



A **job** is a 'customer' of services: it is a process that needs to use system resources during its execution.

A **device** represents a resource in the system. In this simulation, the devices available to a job are the CPU and the disk. There is also a clock device and a pseudo-device that interrupts whenever a new job arrives in the system.

A **scheduler** coordinates access to a device. It queues jobs that are waiting to use a device and will choose which job is the next to access that device.

The overall execution of the coordinator occurs like this: Jobs arrive at the job arrival device and are entered into the system. A job's lifetime consists of alternating periods of using the CPU (often called a **burst**) and performing I/O.

The Main Loop is responsible for moving jobs around the system. It sends them to a scheduler, takes the next job from a scheduler, and starts and stops jobs running on a device.

The Disk Scheduler and the CPU Scheduler decide which job should be the next to run on their respective devices. They also buffer jobs that are waiting to run but have not yet been given access. The clock device is used to enable **preemption**.

Requirements

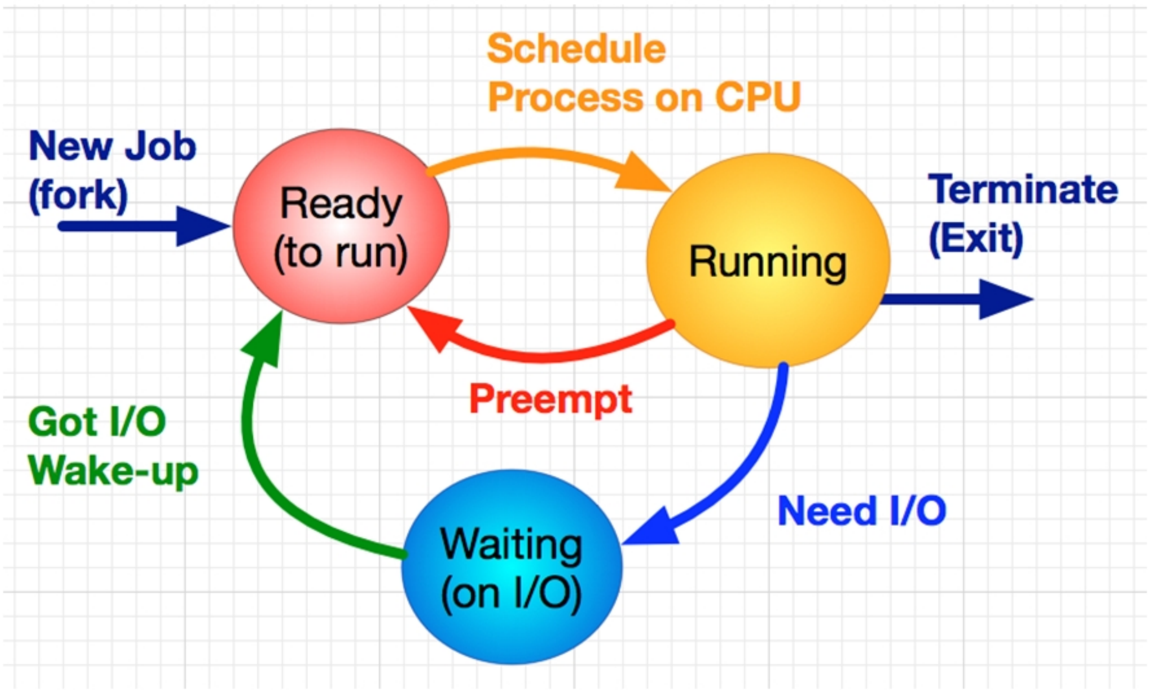
Your project is to implement a simplified process management scheme using different scheduling techniques.

You need to keep track of various statistical data for each scheduling technique and job:

- total time each process was in each state,
- number of jobs run,
- total elapsed time,
- total elapsed time per job (throughput),
- total time each process is in a scheduling state: 1) waiting- ready to run, 2) waiting for I/O, and 3) running on the CPU.
- longest and shortest time taken for any job in the system.

The coordinator works in a loop, a while loop, checking on events, specified below. At each iteration a clock is incremented.

At least seven things should be done in the coordinator loop during each cycle, the first six pretty much map directly to the short-term transitions of a process in a standard/classic operating system. See figure below (recall preemption occurs at the end of a time quantum or time slice).



Your coordinator's main loop will resemble the following (with some variation depending on scheduling policy):

1. if there are new jobs, put these jobs in the "**ready to run**" state
2. if the job currently running on the CPU has used up its time-slice, then put it back in the appropriate queue (could be multiple queues) in the "**ready to run**" state.
3. if the job currently running job requests I/O, then move it to "**waiting** (on I/O)" state.
4. if a job is swapped **out**, then choose a new job to swap **in**.
5. for all jobs waiting for I/O, check if their requests have been satisfied.
6. if a job's I/O has completed move it to "**ready to run**" state
7. keep track of statistics and update as appropriate.

For this simulation jobs will do no "real" work; i.e., your coordinator will implement what happens when the CPU invokes the operating system process scheduler. You can think of this project as a function call inside the operating system. A function call that updates all job states and sets up the next job to run, so that when the function call completes, the CPU has the next job on hand and can begin to run it.

Since our simulated jobs are doing no "real work", we have a little problem. If they are not actually executing code, they cannot request I/O. If they cannot request I/O, we can't transition from running to sleeping on I/O. And since there is no real I/O going on, I/O can never complete, so we cannot transition from sleeping on I/O to a ready to run.

This is a problem.

But we have a solution.

When a job is in the running state you will have a function that "randomly" decides whether the job has requested I/O. When a job is in the "waiting for I/O" state, you will have a function that randomly decides whether I/O has completed for that job. This I/O complete function is applied to each of the jobs that are waiting for I/O in FCFS order. Pseudocode for these functions are listed below.

```
const int CHANCE_OF_IO_REQUEST = 10;
const int CHANCE_OF_IO_COMPLETE = 4;
int IO_Request()
{
    if ( rand() % CHANCE_OF_IO_REQUEST == 0 ) return 1;
    else return 0;
}

 int IO_Complete()
{
   if ( rand() % CHANCE_OF_IO_COMPLETE == 0 ) return 1;
   else return 0;
}
```

While writing your program use rand(), Random, or any other generator you like.

Implementation

The coordinator should never be "idle" if there is an existing job. When jobs are completed, your scheduler will need to update any of its statistics and then process the next job, immediately. If a job is swapped back to "ready to run" state, it should be put on the queue assigned to its priority. For each job that the scheduler works on, that job's time remaining should decrease by one.

Your project's main function could look something like this (in pseudocode):

```
clock = 0;
seed the random number generator
add new incoming jobs to the ready to run state

while (there are jobs in the system )
{
    current_job = choose job to execute
    while (1) /* this loop is the main process management loop */
    {
        Add new incoming jobs to the ready to run state
        while there are jobs waiting for I/O
        {
            /* check if this job's I/O is complete */
            status = IO_complete()
            if status == 1 then
            {
                put the job whose I/O has completed on the
                    ready to run queue
            }
        }

        if this job has finished its work (if time remaining is 1 ) then
        {
            mark current_job as swapped out (job complete, exit)
        }
        else if there are now higher priority jobs on the ready to run state then
        {
            mark current_job as swapped out (preempted)
        }
        /* check for I/O request from current_job */
        else if the job is not complete then
        {
            status = IO_request()
            if status == 1 then /* need to do I/O */
            {
                mark current_job as swapped out (sleeping on I/O)
            }
            else if this job has been on the CPU for an entire timeslice then
            {
                mark the job as swapped out (end of timeslice )
                and move it to the ready to run state
            }
        }

        do bookkeeping and statistics
        clock++

        if current_job was swapped out then
        {
            move the job to the appropriate queue and break from this loop
        }
    }
}
```

*Note: The choose job to execute"line will depend on your different scheduling functions. (If the inner loop above is well thought out, it should not have to change when your scheduling algorithm changes.) Also, make sure that you account for the case that no processes will run. In this case, run an **idle process** (no statistics need to be generated for this pseudo process).*

Ties

Break all "ties" according to PID number. This includes completed jobs on the IO Queue. You should *traverse* the IO Queue FIFO but note if two jobs end up completing at the same time, since that would constitute a tie. Another example: if two processes with the same arrival time arrives at the same time, then the PID with the lowest ID should win:

- process PID 0 arrives at time 55
- process PID 1 arrives at time 55

then PID 0 should be inserted into priority queue 0 before pid 1.

Input and Output (I/O):

Your program takes as input, which is assumed to be regular text.

This input specifies the jobs that are to be scheduled and your coordinator will produce output statistics of the performance of your coordinator.

```
>> my_coordinator input_file.txt
```

Input format

Jobs in this simulation will be simulated using a job data file that contains 4 columns. The format is :

Process Id : Arrival time : Service time : Priority

NOTE: Priority needs to be provided, but depending on the scheduler, may never be used.

For example the data file, input_file.txt, is:

```
123:0:10:1

124:1:20:0
```

Describes a simulation that has two jobs.

The first has a pid of 123, an arrival time of 0 (when the simulation starts), it needs to do 10 clock ticks of work, and has an priority of 1 (which may not be used by your program depending on your scheduler policy).

The second job has a pid of 124, arrives at clock tick 1, needs to do 20 clock ticks of work, and has an initial priority of 0 (again may or may not be used by your program).

The behavior of all the jobs your simulator will handle is described in this data file. Associated with every job is a unique process id (pid), the arrival time for the given job, a service time (how long a job has until completion, i.e., how long the jobs "real work" will take), and a priority rating.

Output details

The final output should be standard and follow the format given below:

Verbose output [NOT REQUIRED, BUT HELPFUL]

Your program is required to produce verbose output if given the "-v" flag on the command line. The verbose output will be written to stdout at every clock tick and consist of the following ONE line format:

```
# while the systems has jobs

<clock tick int>

    :<pid id int>

    :<remaining time for this job>

    :<io_request_boolean>

    :<io_requests_completed_list>

    :<job state at end of loop>
```

<pid id int>	the process id scheduled, an int.
<remaining time for this job>	remaining time of job
<io_request_boolean>	true or false; if there was an I/O request
<io requests>	list of pids requesting I/O, or "none" if none
<job state at end of loop>	"preempted", "still running", "sleeping",

"idling", or * exited

For example, the job file, input_file.txt, above (consisting of PID's 123 and 124) would produce the following verbose output to stdout:

```
0:123:9:false:none:preempted

1:124:19:false:none:still running

2:124:18:false:none:still running

3:124:17:false:none:still running

4:124:16:true:none:sleeping

5:123:8:false:none:still running

6:123:7:false:0:preempted

7:124:15:true:none:sleeping

...

...

...

...
```

As another example, assume the following happens during clock tick 45: job 123 is running, it does not request I/O, it has 15 clock ticks until it exits, it has not used its entire time-slice, and the I/O completes for jobs 234, 345 and 456. Then the verbose output for that cycle would be:

```
45:123:15:false:234,345,456:still running
```

If no I/O had completed it would be:

```
45:123:15:false:none:still running
```

If no process is running on the CPU the output would be:

```
13: *: x:false: 4 :idling
```

PID is denoted by a '*' since no process is running and the remaining time is denoted by an 'x'

If the process you ran completely terminated, the output would be:

```
4: 5: 0:False:none :* exited
```

Command line options

Your program needs to support several command line options.

Your program must be able specify the algorithm that you are using on the command line. This will be done by the flags.

"-A" is for the multi-level aggressive scheduling,

"-N" is for the multi-level non-aggressive scheduling (default), or

"-S" is for preemptive SJF.

If none of the flags are set on the command line your simulator will run the non-aggressive version. An example command line illustrating the specification to run the aggressive version of your algorithm is listed below:

```
>> my_coordinator -A
```

You will also need to be able to set the various time slices for each of the seven priority queues on the command line. The defaults time slices are 1, 2, 4, 8, 16, 32, 64, 128 for each of the priority queues 0 to 7 respectively. You will set the time slices by the flags: -0, to -7. Where -0 denotes setting the time slice for priority queue with priority "0", and so on.

```
>> my_coordinator -0 4 -3 7
```

Would set the time slices of priority queue 0 to 4 and priority queue 2 to 7. All other priority queues would use their default values.

You will also need to set the initial random seed on the command line, as well as the chance of I/O completion and I/O request. An example of a command line that sets these parameters is below:

```
>> my_coordinator -0 3 -s 1 -r 10 -c 4 -v -A input.txt
```

Where:

-0 is the time slice of the highest priority queue, above it is set to 3 (the default of -0 is 1).

-1 is the time slice of the next highest priority queue

-2 , and so on... all the way to -7.

-s is seed value (where 1 is the default),

-r is I/O request chance (where 10 is the default)

-c is I/O complete chance (where 4 is the default), and

-v verbose output is on. [NOT REQUIRED, BUT HELPFUL]

-A, -N, and -S flag allow you to specify the scheduling polices (see above).

Other requirements

The program should be written using object-oriented principles in an OO language such as Python, C++, C#, or Java, etc.

Scheduling policies

For the multi-Level feedback scheduler, you will implement one aggressive and one non-aggressive multi-level feedback scheduler, you will also implement a pre-emptive scheduler.

The algorithm for each scheduler is described next.

Non-aggressive pre-emptive scheduler

1. Jobs are scheduled according to priority, which ranges from 0 (highest priority) to 7 (lowest priority)
2. When a job starts a burst (that is, when it becomes ready either because it has just started or because it has finished doing I/O), it is assigned priority 0 .
3. The scheduler maintains a (FIFO) queue of jobs for each priority level. The scheduler will always run the first job of the highest priority level available (i.e. lowest-numbered non-empty queue). For example, if queues 0 and 1 are empty but queue 2 is not, the scheduler will run the first job in queue 2.
4. When a job is run, it is assigned a slice, which is a number of quanta based on the priority of the job. A job at priority level 0 has a time slice length of 1 quantum, a job at level 1 has a time slice of 2 quanta, a job at level 2 has a time slice of 4 quanta, and so on. In general, a job with priority i has a time slice of 2i quanta.
5. If a job with priority i uses up its time slice without blocking for I/O or terminating, the scheduler stops it, lowers its priority to i+1, and adds it at the tail of queue i+1, and selects a job as in rule (3). However, if the job is already in the lowest priority queue, its priority is unchanged and it returns to the end of the same queue. While it is possible that the same job will be selected again--for example, if it is the only ready job--normally a different job will be given the opportunity to run.
6. This policy is non-aggressive in the following sense: If a job becomes ready while another job is running, it is added to the tail of queue 0, but the running job is not stopped until it terminates, blocks for I/O, or uses up its time slice.

Aggressive pre-emptive scheduler

This version is a modified version of your first version. In this version jobs arriving at the CPU scheduler can preempt running jobs, and the priority of a job is ``remembered'' from one burst until the next. In more detail, rules (2) and (5) are modified as follows:

- (2)' When a job becomes ready because it has finished doing I/O, it is given priority i-1, where i is the priority it had when it blocked for I/O. There is no level -1, so if a job finishes a burst at priority 0, it stays at priority 0. Newly created jobs are assigned priority 0.

- (5)' This policy is aggressively preemptive in the following sense: If a job becomes ready while another job is running, it is added to the tail of the appropriate queue as defined by rule (2'), the running job is stopped and has 1 subtracted from its priority (unless it is already at priority 0), it is added to the tail of the appropriate queue, and another job is selected to run as in rule (2).

Preemptive shortest job first

In this strategy the ready queue will consist of one queue ordered according to the time that the scheduler 'thinks' the job needs on the CPU. You will need to calculate this "guess" using exponential averaging (research this). The weight of the most current value is w and the default weight is 1/2.

Deliverables

1. The artifacts you used for a mandatory run using the provided input data file:
2. Verbose output file for Scheduler 1 [OPTIONAL]
3. Non-verbose output file for Scheduler 1
4. Verbose output file for Scheduler 2 [OPTIONAL]
5. Non-verbose output file for Scheduler 2
6. Your code committed and pushed to master with a tag/release submitted to Canvas.