Lab Exercise 2 Revision 1

The size of a team was slightly modified. The description of the input was moved to before the section "What You Are to Do". The need to check input is listed. When to block has also been clarified.

Due: June 2, 2022
Points: 100

Please form a team of 2 or 3. Include the names of all team members in the header comment. One person needs to submit the program; the other team members must submit a short note identifying who turned in the program.

Introduction

This assignment has three goals: first, to write code that is functionally identical to the low-level blocking/unblocking code and dispatching code of any operating system; to measure the performance of the system using many of the important measures used to evaluate operating systems performance; and finally, to practice using C structures and pointers.

The Problem

The canonical model of a process in an operating system is:

```
do
        compute for a bit
        do input or output for a bit
until done
```

Here, we see that a job computes for some amount of time and then does some I/O, which also takes a varied amount of time. In this assignment, we shall build that part of an operating system that maintains the ready queue, and that part which maintains the I/O queue. We shall then simulate the time various processes would spend executing and doing I/O, and gather statistics for a variety of job scheduling algorithms.

Input Data

Each job has four attributes associated with it: its name, run time, and the probability of blocking. The name is at most 10 characters long. The run time is an integer; the probability of blocking is a floating point number between 0 and 1 inclusive.

The input file is a series of lines with three columns, the first being the name (and no more than 10 characters), the second the run time, and the third the probability of blocking. For example,

editor	5	0.87
compiler	40	0.53
adventure	30	0.72

Notice that the columns are separated by one or more blanks and/or tabs. Remember to do error checking; specifically, check the following:

- The name is no more than 10 characters. (You may assume no characters other than letters and digits make up the name, in particular no whitespace will appear in the name.)
- The run time ise an integer 1 or more.
- The probability is a decimal number between 0 and 1 with 2 decimal places.

What You Are To Do

After reading in the job information (see below), your program will then simulate the execution of those jobs. There are two scheduling policies involved. For the first part of the programming assignment, you are to use First Come First Serve as both the CPU and I/O scheduling policies; for the second part, you are to use Round Robin as the CPU scheduling policy and First Come First Serve as the I/O policy.

When a process is dispatched to the CPU, you must determine whether it is to block for I/O (and is to be transferred to the I/O queue). To do this, generate a random number between 0 and 1. Then compare it to the probability that the process will block (obtained from the input, as above). If the number you generated is less than the input probability, the process blocks; otherwise not.

For the First Come First Serve CPU scheduling policy, if the process does not block, it will run until it finishes. If the process is to block, you must then decide how long it will run before blocking (again, using a random number generator). This value must be an integer between 1 and its remaining run time inclusive.

For the Round Robin CPU scheduling policy, you have to consider the same issues. Here, though, you also have to handle the quantum of 5. So to determine when to block, generate a random integer between 1 and 5 inclusive to determine how long the process will run before blocking.

For the I/O queue, the next request to execute is the first one in the queue. The amount of time needed to service this request is to be generated randomly and be an integer between 1 and 30 inclusive. Once the I/O request is started, the entire time needed to service it is dedicated to that request. So, when dispatching the next I/O request, you only need decide how long it will take to complete. When the request completes, the job moves to the end of the ready queue

Generating Random Numbers

For this program, you need to use the function *random*(3). This is a random (well, pseudorandom\footnote{Since they are generated by a mathematical formula, they are not truly random. However, to all appearances, the sequence of numbers appears to be randomly selected, hence the term “pseudorandom.” For brevity, however, they are called “random” and with the above understanding we shall use that term here.}) number generator that generates an integer between 0 and **RAND_MAX** (defined in the header file *stdlib.h*) inclusive.

If you simply call *random*, each time you run your program, you will get a different series of numbers. While correct, this will make gradescope unusable because it compares your program’s output against reference output. So we need to have everyone use the same starting point for *random* to begin its generation of random numbers. To do this, call the following function ***exactly*** as below. In particular, make sure you enter the number correctly!

```
(void) srandom(12345);
```

This will initialize *random* just as in the program used to generate the output files.

WARNING: there are other functions designed to generate random numbers. Do not use them.

User Interface

Your program should be written to allow the user to specify which scheduling policy (FCFS or Round Robin) they want to simulate using a command line option. Use `-r`:

```
procsim -r
```

to produce output for the Round Robin scheduling policy and the option `-f`:

```
procsim -f
```

to produce output for the FCFS policy.

Statistics To Gather

As you perform the simulation, gather the following statistics.

For Each Process

- Its name;
- Its total CPU time (as read);
- The wall clock time at which it was completed (this is the value of your own counter, not the time of day);
- How many times the process was given the CPU;
- How many times the process blocked for I/O; and
- How much time the process spent doing I/O.

For the System

- The wall clock time at which the simulation finished.

For the CPU

- Total time spent busy;
- Total time spent idle;
- CPU utilization (= busy time / total time);
- Number of dispatches (number of times a process is moved onto the CPU); and
- Overall throughput (= number of processes / total time).

For the I/O device

- Total time spent busy;
- Total time spent idle;
- I/O device utilization (= busy time / total time);
- Number of times I/O was started; and
- Overall throughput (= number of jobs / total time).

Output Format

Your output is to look like this:

```
Processes:

 Name  CPU time When done # Dispatches # Blocked for I/O I/O time
 name        n         n            n                 n        n
   ...
```

```
System:
The wall clock time at which the simulation finished: n

CPU:
Total time spent busy: n
Total time spent idle: n
CPU utilization: nn.nn
Number of dispatches: n
Overall throughput: nn.nn

I/O device:
Total time spent busy: n
Total time spent idle: n
I/O device utilization: nn.nn
Number of times I/O was started: n
Overall throughput: nn.nn
```

Text that should appear as shown is in `typewriter font`; text that is to be computed and inserted is in *italics*; and all numbers should either be integers (shown by *n*) or have 2 decimal places (shown by *nn.nn*). In the section of output listing the fields, use single tabs to separate the headings, line up the process names at the beginning of the line, and line up the values at the end of each header. One blank line separates each section of output.

This web page may be showing a blank line between the `Process:` header and the table. Your output should ***not*** have that line; the `Processes:` is to be followed by the `Name` line. That blank line is an artifact of how the browser draws the web page.

Some Hints

Here are some suggestions that you should feel free to either take or ignore.

In my solution, I have five major data structures:

- An integer variable to record the current (wall clock) time; initially it is 0.
- A structure to record information about the CPU. This includes the relevant statistics, the CPU status (busy or idle), and a time field that gives the time at which the currently running job should stop running. Of course, that field is relevant only when the CPU status is busy!
- A structure to record information about the I/O device. The structure is analogous to that for the CPU.
- A structure to store information about each job. This consists of the job's name, its priority, its probability of blocking, the time it is to run, and how much run time remains. There is also space for two link fields.
- A linked list for the ready queue and another for the I/O queue. These consist of structures for the jobs linked together.

The main routine does some initializations, runs the simulation, and prints out the statistics. The initialization loads the process description information and puts the process in the ready queue, and then the CPU dispatch routine is called to start things rolling. The I/O device is initially idle.

The major portion of the program is the simulation. The simulator loops until the ready queue and the I/O queue are both empty, at which point the simulation is done. During each loop, the CPU and I/O structures are checked to see what is to be done next. The one which finishes earlier is honored (but if one of the two is idle, it is of course ignored.)

There is a preempt routine for the CPU and I/O device to deal with the currently running process. This moves the process from the head of the queue to the end of the appropriate queue. The next process is then dispatched.

Note that the CPU and I/O routines can call each other. If a process moves to the end of the I/O device to block, the CPU preempt routine should call the I/O dispatcher if the I/O device is idle. Similarly, if a process moves to the end of the ready queue, the I/O preempt routine should call the CPU dispatcher if the CPU is idle. Don't forget that at some point you have to check to see if the currently running process has completed.

What to Turn In

Turn in your program using gradescope in the area for Lab Exercise 2a.

Turn in a short (at most, 1 page) writeup in the area for Lab Exercise 2b. In this writeup, compare and contrast the results of your simulation of Round Robin with those of FCFS. In particular, if these are a representative set of processes for a site, and the site is trying to decide between a Round Robin and a FCFS process scheduling policy, which one should it adopt, or should it combine them in some way? Why?

Matt Bishop
Office: 2209 Watershed Sciences
Phone: +1 (530) 752-8060
Email: mabishop@ucdavis.edu

ECS 150, Operating Systems
Version of May 12, 2022 at 10:49PM

You can also obtain a PDF version of this.