CS3204 Operating Systems - Spring 2001
Instructor: Dr. Craig A. Struble
Multiprogramming Batch OS Simulation

Assigned: Monday, Feb. 26 **Due:** 11:59.59 p.m., Monday, Mar. 26

1 Introduction

Multiprogramming batch operating systems improve overall system performance by allowing one *job* (that is, process) to execute while another job was performing I/O. By supporting multiprogramming, CPU utilization is increased, increasing the job throughput, the number of jobs completed per some unit of time. Multiprogramming batch systems have a *hold queue*, where jobs are maintained until enough memory is available for their execution. Jobs are removed from the queue and allocated memory based on a *long-term scheduling algorithm*. Once jobs have been allocated memory, they moved to a *ready queue* and compete for CPU time. Jobs are scheduled access to the CPU based on a *short-term scheduling algorithm*.

The purpose of this assignment is to simulate a multiprogramming batch operating system that is managing several jobs. Each job interleaves execution on the CPU and performing I/O. The state of the operating system is displayed at specified times, to verify proper operation.

2 Specification

You are to implement a multiprogramming batch operating system simulation that manages several jobs. The executable implementing the simulation must be named `mbos` and accepts two command line arguments: the input file name and output file name. A sample command line for executing your program is

```
% mbos infile outfile
```

where `infile` is the name of the input file and `outfile` is the name of the output file.

2.1 Input File

The input file consists of lines containing the following commands:

- `init <memory> <context> <io>` initializes the operating system to have `<memory>` available units of user memory (a positive integer), the amount of overhead required for a context switch `<context>` (a non-negative integer), and the amount of time required to read or write a single record `<io>` (a positive integer). **This line must be the first line in the input file.**
- `load <time> <job>` loads the job stored in file `<job>` (a string without spaces) at system time `<time>` (a non-negative integer). The job is initially placed in the hold queue and competes for memory allocation. Jobs are assigned a positive integer as an an identifier (e.g., 1, 2, 3, 4, ...) in the order in which they are loaded.

- `print <time>` prints the state of the system at time `<time>` (a non-negative integer) to the output file specified on the command line. The output has the following format (each column is left justified):

```
Sys.    OS          Avail.               Average     Average   Average
Time    State       Memory       Job     Turnaround  Wait      Weighted
---------------------------------------------------------------------------
<time>  <state>     <memory>     <job>   <ttime>     <wtime>   <wetime>

Job     State       Memory      Start    Finish    Remain   Wait     File
---------------------------------------------------------------------------
<id1>   <state1>    <memory1>   <stime1> <ftime1>  <rtime1> <wtime1> <file1>
<id2>   <state2>    <memory2>   <stime2> <ftime2>  <rtime2> <wtime2> <file2>
...

Hold Queue:        <id1> <id2> ...
Ready Queue:       <id1> <id2> ...
I/O Request Queue: <id1> <id2> ...
```

  For the most part, the values to be printed should be self explanatory. Averages are printed with 2 digits following the decimal point in fixed point format.

  The values for `<state>`, the OS state, are `RUNNING` (the OS is in control of the CPU), `CSOS` (context switch to the OS), `CSJOB` (context switch to a JOB), or `IDLE` (a job is executing). The `<job>` value should be empty if the state is `RUNNING`, the id of the job that was executing if the state is `CSOS`, the id of the job about to be executed if the state is `CSJOB`, or the id of the job that is executing if the state is `IDLE`.

  If no processes have completed, the average times should be printed as `0.00`.

  The values for `<state1>` the job state are `HELD` (the job is in the hold queue), `READY` (the job is in the ready queue), `RUNNING` (the job has control of the CPU), `BLOCKED` (the job is waiting for I/O), or `DONE` (the job has completed). Jobs that are not loaded should not be displayed in the job list. The list should be ordered by job id.

  For each of the queues, the job ids should be printed in order from the head to the tail of the queue.

2.2 Job Files

Job files are used to specify the behavior of jobs. Each job file consists of lines containing the following commands:

- `memory <mem>` specifies the amount of memory `<mem>` (a positive integer) needed to load the job. **This line must appear as the first line in the job file.**

- `compute <time>` specifies an amount of time `<time>` (a positive integer) the job spends computing on the CPU. Several of these lines may appear in the job file. The sum of all the `<time>` parameters appearing in the job file is the total amount of CPU time needed by the job to complete its execution.

- `read <records>` is a request to read `<records>` records (a positive integer) from the disk. The process remains blocked until all of the records are read.
- `write <records>` is a request to write `<records>` records (a positive integer) to the disk. The process remains blocked until all records are written.

The job files are processed from top to bottom, carrying out the commands in order. For example, a job file with the contents

```
memory 10
compute 50
read 2
compute 20
```

states that the job requires 10 units of memory, will compute for 50 units of time, then read 2 records from the disk, and then computes for another 20 units of time. The total amount of CPU time needed by the process is 70 time units.

2.3 Errors

Your implementation should verify that the arguments passed to the program are valid. If the number of arguments is incorrect, or there are problems opening the input or output files specified on the command line, your simulation should print out an appropriate indication of the error that occurred followed by a message describing how your program should be executed.

If an error occurs after the input and output files are opened successfully, then each subsequent error message shall be printed to the output file. Error messages have the following format:

```
(<file>, <line>) ERROR: <Error message>
```

where `<file>` is the file containing the error (input or job file), `<line>` is the line number in the file containing the error, and `<Error message>` is a message describing the error that occurred.

You may assume that each line of the input and job files corresponds to a command, however, the commands may have missing or extra parameters. Also, the parameters may not be valid (e.g., a negative integer for the amount of memory). Invalid commands may also appear. An appropriate error message should be printed when an error in the input file occurs. In general, a line with an error is ignored and the simulation continues with the next line.

If the first line of an input file contains an error or does not initialize the operating system, an error message is printed and the simulation stops immediately.

The time parameter for an input file command should be greater than or equal to the time parameter of the previous command. If the time parameters decrease, the error should be detected and an informative message printed.

Jobs may fail to load for several reasons:

- the first line of the job file is not the `memory` command,
- the first line contains an error,

- or the memory required by the job is more than the maximum provided by the system.

An error message should be printed for each of these cases. **In all cases, the prospective job is assigned an identifier even if it cannot be loaded by the operating system.**

2.4 System Time

System time starts at 0 and is incremented in integral time units.

2.5 Long-Term Scheduling

Recall that a long-term scheduler is used to allocate memory to a job in the hold queue and places that job into the ready queue. In this assignment, the long-term scheduling algorithm is strictly *first come first served*. This may mean that jobs with smaller memory requirements will have to wait until a job with larger memory requirements can be allocated memory. Whenever the long-term scheduler is executed, as many jobs as possible should be moved from the hold queue to the ready queue.

2.6 Short-Term Scheduling

Recall that a short-term scheduler dispatches a job in the ready queue to the CPU. In this assignment, the short-term scheduling algorithm is *shortest job next*. To determine how much time a job has remaining, consider only the CPU time needed by the job. **The amount of time needed for I/O is not including in the CPU time needed.**

2.7 Input and Output

Only one process at a time may be actively performing I/O in the system. I/O is handled in *first come first served* order. An I/O request completes when enough time has elapsed to read or write the number of requested records.

2.8 Interrupts

The currently executing job, if one exists, is interrupted whenever one of the following happens:

- the executing job requests I/O,
- an I/O request completes.

2.9 Context Switches

A context switch occurs whenever one of the following happens:

- an interrupt occurs,
- a job completes its execution,
- a job is dispatched to the CPU.

The overhead for a job's final context switch **should not** be included in the turnaround time of the job.

2.10 Operating System Behavior

The behavior of a multiprogramming batch operating system will be covered in class. Some of the following notes may be helpful.

When the OS gains control of the CPU, the following occur

- The currently executing process has its state changed from running to whatever is appropriate;
- Any pending interrupts are handled;
- The long-term scheduler is executed;
- The short-term scheduler is executed;
- The selected job is dispatched to the CPU.

You will have to consider the following questions:

- What happens when there are no processes in the hold queue?
- What happens when there are no processes in the ready queue?
- When is the I/O request queue updated?
- When do processes move from the hold queue to the ready queue?
- What is the state diagram for operating system execution?

3 Executing the Simulation

Your simulation reads the input and job files and carries out the instructions in each file. When your simulation starts, the following message is printed to the output file specified on the command line:

```
MBOS Simulation CS3204 Spring 2001
Programmer: <your name>

MBOS Initialized
Maximum Memory:          <memory>
Context Switch Overhead: <switch>
I/O Overhead Per Record: <io>
```

where `<your name>` should be replaced with your name, and `<memory>`, `<switch>`, and `<io>` are replaced with the values specified in the input file.

Each line of the input file is processed from top to bottom, carrying out the commands at their specified times and in the order in which they appear should two commands be carried out at the same time. The simulation should run until every job completes, the operating system has regained control of the CPU, and the entire input file has been processed. At that time, the final state of the system should be printed as if the `print` command were executed.

4 Design and Implementation Requirements

- You may assume that no line of input is longer than 255 characters.
- **You must use separate compilation for this program.** Programs implemented in a single source file will not be accepted.
- Your program is expected to be well designed and implemented. This means you must adhere to the principles of appropriate information hiding and encapsulation. You may use either object-oriented or procedural techniques for design and implementation, as long as you adhere to these design principles.
- Your design and implementation should reflect the major operating system components we've discussed in class. You should incorporate ideas such as interrupt handling, system calls, context switching, process descriptors, process and resource managers, dispatchers, etc. as appropriate for the project.

5 Test Files

Sample input and output files will be available on the course web site. These files should not be considered a comprehensive test of your program. You should include several of your own tests in your program submission to demonstrate that your program works properly.

6 Submission

We will use the Curator, `http://ei.cs.vt.edu/~eags/Curator.html` to collect program submissions. The URL for submission is `http://spasm.cs.vt.edu:8080/curator/`. Only the servlet interface to the Curator is supported. No grading will be done by the Curator.

You are to submit a single tarred (`man tar`) and gzipped (`man gzip`) archive containing

- A text file named `README` describing the program, describing the contents of the archive, providing building instructions (including the platform you used for development), a user's guide (including how to start the program), and a description of any outstanding bugs;
- The source code for your program;
- The test files you used for your program;
- A script named `build` or a suitable `Makefile` for building your program.

Your files must be in directory named `project3` of the archive. Be sure to include only the files listed above. Do not include extra files from an integrated development environment such as `configure` scripts, automake related files, etc. This is primarily an issue if you are using KDevelop.

Be sure to include your name in all files submitted. **DO NOT** include executables or object files of any type in the archive. **Submissions that do not gunzip and/or untar will not be graded. Be careful to FTP in binary mode if you are transferring your file to a Windows machine before submitting to the Curator.**

Failure to follow the submission rules will result in a grade of zero (0) for this assignment. There will be no exceptions.

7 Programming Environment

As stated in the syllabus, you must use Linux and `gcc/g++` to implement this project. Your program must compile and run properly in the lab on RedHat 6.2. **All data structures used in your program must be student implemented. Using the standard template library (STL) or other third party libraries for data structure implementations is strictly prohibited.** Using C++ input and output streams and C++ strings is OK.

While it is feasible that you can develop the project under Windows using Visual C++, I recommend against it. There are subtle differences with the template implementations and memory management that can cause a program that appears to run properly with Visual C++ to fail miserably under Linux. The best recipe for success with this project is to start early, build incrementally, test frequently, and make several snapshots as you progress. While you should spend most of your early time designing, you can build and test the common data structures, like queues, early as well. I can almost guarantee that waiting until the last week to start this project will result in disaster.