CS3204 Operating Systems - Fall 2000
Instructor: Dr. Craig A. Struble
Simulating Process Scheduling and States

Assigned: Tuesday, Sep. 26; *Revised: Oct. 5* **Due:** 11:59.59 p.m., Monday, Oct. 23

1 Introduction

We have discussed process and resource management in the course. Processes may be one of three different states during their lifetime: ready, running, blocked. The operating system controls the transitions between each of the states using different mechanisms: scheduling algorithms, waiting for children, and resource allocation, for example.

The purpose of this assignment is to write a simulation of an operating system that manages several processes and resources. You will implement a process scheduling algorithm and a resource allocation algorithm. You will need to make heavy use of data structures, including ready lists, resource queues, and a process hierarchy. As part of the simulation, you will calculate estimated turnaround times for each process.

2 Specification

You are to implement an operating system simulation that manages several processes and resources. The executable implementing the simulation must be named `ossim` and will take two command line arguments: the input file name and output file name. A sample command line for executing your program is

```
% ossim 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:

2.1.1 Initialization Commands

- `init <mem> <clock> <quantum> <context> <processes> <levels>` initializes the operating system to have `<mem>` units of memory (an integer), initial system clock value `<clock>` (a real number), the time quantum allocated to each process `<quantum>` (a real number), and the amount of overhead required for a context switch `<context>` (a real number). The operating system can handle up to `<processes>` processes (an integer) at a time (only includes processes in one of the ready, running, or blocked states). There are `<levels>` levels (an integer) in the multi-level feedback queue.

  This line must be the first line in the input file. If the line does not appear first, it is an error and the simulation should be stopped.

- `resource <name> <type> <units>` initializes a resource named `<name>` (a string) which is either reusable or consumable as indicated by `<type>` (either `R` or `C`). If the resource is reusable, then `<units>` is the number of units available for the resource.

If the resource is consumable, then `<units>` is the initial number of units available. The parameter `<units>` is an integer.

Resource initialization lines appear after the `init` line and before any other input commands are present.

2.1.2 System Commands

These commands should be placed in the system command queue as they are parsed.

- `create 0 <id> <memory> <cpu>` creates a system process with process id `<id>`. The process uses `<memory>` units (an integer) and will execute for at most `<cpu>` time units (a real). This command interrupts the currently running process (if one is running). The operating system then creates the new process and places it into the ready queue.

  If there is not enough memory to create the new process, the process is not created and an out of memory error occurs. A process that is not created as a result of insufficient memory does **NOT** get placed on the termination list.

- `produce <name> <units>` causes the simulation to produce `<units>` units (an integer) of a consumable resource named `<name>` (a string). This command interrupts the currently executing process to produce the units. It is an error if `<name>` does not refer to a consumable resource.

- `print` prints the current state of the operating system, including current system time, active processes, terminated processes, resources, and various statistics. This command does **NOT** interrupt the currently executing process.

  Below is a sample output file containing a complete example of all of the data to display. This format must be used for your assignment. *This output still contains fictional data, but the format of the output is fixed.*

```
System time: 3.076
Memory available: 6

Active Processes
================
Process   Parent   State     Blocked on   Memory   Time   Pri
-----------------------------------------------------------------
1         0        Ready                      10   0.568    2
2         0        Running                     2   0.234    1
3         0        Blocked   waiting           5   1.450    4
4         3        Blocked   keyboard          7   0.325    2

Terminated Processes
====================
Process    Service    Wait    Turnaround  Weighted
-----------------------------------------------------------------
10           1.098   0.050        2.047     2.056
-----------------------------------------------------------------
Averages     1.098   0.050        2.047     2.056
```

```
Resources
=========
Name        Kind  Available  Maximum  Queue
-----------------------------------------------------------------
keyboard    C             0      INF  4

Ready List
==========
Level       Processes
-----------------------------------------------------------------
1           2
2           1
3
4
```

The current system time and amount of available memory are printed first. Four tables containing the status of simulation follow the system time and available memory. The contents of each table are described below.

Active Processes This table contains the status of active processes in the system. The rows of the table are sorted in ascending order by process id.

Column	Description
Process	The process id
Parent	The parent proces id
State	The current state of the process.
Blocked on	The reason (resource name or `waiting`) a process is blocked.
Memory	The amount of memory allocated to the process.
Time	The amount of CPU (service) time used by the process.
Pri	The priority (level in the multi-level feedback queue) of the process.

Terminated Processes This table contains the list of terminated processes. The rows of the table are sorted in ascending order by process id.

Column	Description
Process	The process id
Service	The service time for the process.
Wait	The wait time for the process.
Turnaround	The turnaround time for the process.
Weighted	The weighted turnaround time for the process.

When computing the turnaround time, do **NOT** include the context switch after process termination.

Resources This table contains the list and status of configured resources. The rows of this table are displayed in ascending order by resource name (using standard C/C++ string ordering).

Column	Description
Name	The resource name.
Type	The type of resource (reusable or consumable).
Available	The number of units available.
Maximum	The maximum number of units for the resource. `INF` for consumable resources
Queue	Processes waiting for resources in their queued order (head at the left).

Ready List This table displays the contents of the ready list. The rows of this table are ordered from highest priority to lowest priority.

Column	Description
Level	The level in the multi-level feedback queue.
Queue	The contents of the queue in queued order from left to right.

2.1.3 Process Commands

The commands should be placed into the command queue for the specified process as they are parsed.

- `create <parent> <id> <memory> <cpu>` creates a process with process id `<id>` (an integer) as a child of the process with process id `<parent>` (an integer $\neq 0$). The process uses `<memory>` units (an integer) and will execute for at most `<cpu>` time units (a real). The child process is created and placed in the ready queue when the parent executes this command. The parent process is not interrupted, continuing to execute its time quantum.

  If there is not enough memory to create the child process, the child process is not created and an out of memory error occurs. A process that is not created as a result of insufficient memory does **NOT** get placed on the termination list. The parent process continues to execute following the error.

- `request <id> <name> <units>` causes the process `<id>` (an integer) to request `<units>` units (an integer) of the resource `<name>` (a string).

  If the resource is a reusable resource and the total number of units allocated and requested by process `<id>` exceeds the maximum number of units for `<name>`, then the process should be terminated and an error message is printed. The terminated process is placed on the termination list.

  Consumable resources are consumed immediately after they have been allocated to the process and the process executes.

- `release <id> <name> <units>` causes the process `<id>` (an integer) to release `<units>` units (an integer) of the reusable resource `<name>` (a string). If the process releases more units than it has allocated, all units should be released, but it is not an error. It is an error if resource named is not reusable.

- `terminate <id>` is a request to terminate the process with process id `<id>` (an integer). The next time the process executes, it terminates itself, releasing its resources and returning control to the operating system. Child processes continue to be simulated, but their parent process id becomes -1.
- `wait <id> <child>` causes the process with id `<id>` (an integer) to wait for the child process with id `<child>` (an integer) to terminate. If the child process has already terminated, no waiting occurs and `<id>` continues to execute. When the child terminates, process `<id>` should be placed back in the ready queue.

  If process `<child>` is not a child process of `<id>`, it is an error. The process `<id>` continues to execute following the error.

2.1.4 Simulation Command

- `simulate <time>` causes the operating system to simulate execution for `<time>` (a real) amount of time. Processes should be moved throughout the system taking into account the time quantum, the context switch overhead, etc.

The simulation ends when the end of the input file is reached. Except for an error occurring with the `init` statement, when an error occurs, an error message must be printed out and the input line ignored.

2.2 Error Handling

For this project, a small subset of possible errors are required to be handled. The complete list of errors that must be handled is

- Failure to include an `init` command;
- Not enough memory to create a process;
- Not enough total units of a reusable resource to satisfy a request;
- Releasing a resource that is consumable;
- Producing a resource that is reusable;
- Waiting for a process that is not a child.

All other possible error situations such as referencing invalid resource names or invalid process identifiers, configuring the same resource twice, reusing process identifiers, etc. are **NOT** required to be handled. If you handle these extra error cases, you can get extra credit. An addendum describing a complete list of possible errors and the amount of extra credit you can receive will be posted later.

Errors do not stop the execution of the simulation (with the exception of the `init` command). The command containing the error is ignored, and your program should continue to read and execute lines of input.

Errors should be printed when the command containing the error is executed in the simulation. So, you may parse an input line with one of the errors listed above, but not print the error message until a process executes the command. If a process never executes the deviant command, because of deadlock for example, an error message will **NOT** be printed.

2.3 Output File

The output file contains the output of print commands and error messages.

The output of the print commands must match the format given in the description of the `print` command.

Error messages should have the following format:

```
Error (line XXX): ERROR MESSAGE
```

where `XXX` is the line number of the input file containing the error and `ERROR MESSAGE` is a brief message describing the error. Failure to follow this format may result in points being lost.

2.4 Process Scheduling

Your scheduler must implement the *multi-level feedback queue* approach for scheduling. Within each queue, round-robin (RR) scheduling is used. All higher priority processes are executed before lower priority processes.

When a process first arrives, it is placed into the highest priority (lowest numbered) queue. Upon completing a **FULL** time quantum, the priority is lowered by one (that is, placed in the next higher numbered queue). This means, for example, that if a process is blocked in the middle of its time quantum, its priority does **NOT** change. A process never has its priority increased in this simulation.

The priorities and levels in the feedback queue are labeled 1–`<levels>`.

2.5 Resource Allocation

Resources are allocated on a first-come first-served (FCFS) basis. In some cases, there might be enough resources to allow some process A requesting resources to execute. Using FCFS, A has to wait until all all preceding processes are allocated the resources they need to use.

If resources are available, they are allocated immediately and the process is not blocked. Otherwise, the process is blocked until the resources requested are available.

If a resource is replenished (as the result of process termination, `release`, or `produce` commands), the blocked queue for the resource should be checked for blocked processes. If enough resources are available to unblock the process, the resources should be allocated and the process moved to the ready queue immediately.

The resource manager should unblock as many processes as possible from the blocked queue. As a result, several processes may be placed into the ready state as the result of a resource being replenished.

2.6 Process Termination

When a process is terminated, all allocated resources are released and the process is placed on a *termination list*. If the process' parent is waiting for the process to terminate, the parent process should be placed back into the ready queue before releasing resources. Resources are released in ascending order by resource name (using the standard C/C++ string ordering). Note that the process must be context switched off of the CPU before the resources are released and the process is placed on the termination list.

For each terminated process, calculate and store the service, turnaround, and weighted turnaround times. The turnaround time does **NOT** include the final context switch after process termination.

2.7 Interrupts

When a process is interrupted (as the result of creating a system process or producing resources), it should be context switched off and placed in the ready queue before handling the system request.

2.8 Context Switches

To ease the handling of context switch overhead, a context switch is defined to occur only when a process changes from the ready state to the run state or from the run state to the ready or blocked states or moved to the termination list. Simply put, a context switch occurs whenever a process is given control of or is removed from the CPU.

Requesting a resource that is immediately available or creating a process (that is not a system process) do not cause context switches.

If a system command is read during the execution of a context switch, the command is not handled until the context switch completes. For example, if the `produce` command is read and the system is in the middle of a context switch, the resources are not produced until the context switch is completed.

As soon as the time quantum for a process has completed or a process has completed execution, a context switch is immediately initiated. As a result, if the simulation time from a `simulate` command and time quantum complete simultaneously, any system commands and process commands read from the input file are executed after the context switch completes.

3 Executing the Simulation

The simulation only moves forward as the result of the `simulate` command. This means that other commands need to be queued for execution until a `simulate` command is read. During the simulation time, the commands read in and their effects on processes and resources in the system are simulated.

For example, assume

```
request 3 keyboard 10
```

is read, indicating that process 3 is requesting 10 units of the keyboard resource. The command should be queued in a command queue for process 3. Then, when a `simulate` command is executed so that process 3 is placed in the run state, the process will request resources from the operating system (potentially blocking).

Commands handled by the operating system directly, such as `produce` and `create` with a parent process id of 0, are always handled before other commands during simulation execution time.

4 Implementation Details

You may assume that no line of input is longer than 255 characters. All input lines will be formatted properly (e.g., no invalid commands and no extra data).

The maximum number of processes that can exist in the system is given as part of the simulation initialization. You may take advantage of this limit to simplify the implementation of some of your data structures. Note that there is no limitation given regarding the overall number of processes handled during the execution of your simulation, so you will need a dynamic data structure for implementing the termination list.

All real numbers will have at most three (3) digits following the decimal point. In order to avoid roundoff errors, you should **READ AND STORE** real numbers as integer values instead. If your output does not match with the output of the grading program because of roundoff errors, you will lose points.

You may assume that when a process command is in the input file, that the associated process has been created as the result of a `create` command and the passing of sufficient simulation time. This implies that when you read in a process command, you can store the command in an already existing process command queue.

You must use separate compilation for this program. Programs implemented in a single source file will not be accepted.

5 Test Files

Sample input and expected output files will be available on the course web site. Initially, simple sample input and output files will be available. On Monday, October 16, we will post more complex input and output files. 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 examples of usage with your test files;
- 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 the top level directory of the archive (i.e. not placed in a subdirectory). 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 severe penalties. There will be no exceptions made for this assignment.

7 Programming Environment

As stated in the syllabus, you may use either FreeBSD or Linux and ANSI C/C++ to implement this project. **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.