School of Computing and Information Systems
COMP30023: Computer Systems

Project 1

Due date: No later than 11:59pm on Thursday, April 6.

Weight: 15%

Project Overview

The aim of this project is to increase your familiarity with issues in memory management, as well as process scheduling.

Your task is to write a simulator which takes process of different sizes; loads them into memory when required, using one of three different algorithms and when needed, swaps processes out to create a sufficiently large hole for a new process to come into memory. It also takes care of scheduling processes currently in memory using a *Round Robin* algorithm.

Your simulator must be written in C. Submissions that do not compile and run on the School of Engineering Linux servers (eg., `dimefox.eng.unimelb.edu.au` or `nutmeg.eng.unimelb.edu.au`) may receive zero marks.

Project Details

Assume there are two CPUs, the first one is dedicated to running the swapper and system code and can be ignored for the purposes of this simulation. The second CPU is used for running the (user) processes. The times required to do the swapping and scheduling are ignored in the simulation.

For bringing a process from disk into memory, we use one of three different algorithms: *first fit, best fit* and *worst fit*. Assume that memory is partitioned into contiguous segments, where each segment is either occupied by a process or is a hole (a contiguous area of free memory).

The *free list* is a list of all the holes. Holes in the free list are kept in *descending* order of memory address. Adjacent holes in the free list should be merged into a single hole.

The algorithms to be used for placing a process in memory are:

- *First fit:* First fit starts searching the free list from the beginning (highest address), and uses the first hole large enough to satisfy the request. If the hole is larger than necessary, it is split, with the process occupying the higher address range portion of the hole and the remainder being put on the free list.
- *Best fit:* Chooses the smallest hole from the free list that will satisfy the request. If multiple holes meet this criterion, choose the highest address one in the free list. If the hole is larger than necessary, it is split, with the process occupying the higher address range portion of the hole and the remainder being put on the free list.
- *Worst fit:* Chooses the largest hole from the free list that will satisfy the request. If multiple holes meet this criterion, choose the highest address one in the free list. If the hole is larger than necessary, it is split, with the process occupying the higher address range portion of the hole and the remainder being put on the free list.

The simulation should behave as follows:

A *process file* (see `input.txt`) is a sequence of entries which describes a list of processes that are to be created. The first entry refers to the first process that is created, and the last entry refers to the last process that is created. Each entry consists of a tuple (time-created, process-id, memory-size, job-time). For example:

```
0 3 85 30
5 1 100 20
20 6 225 15
24 10 50 14
```

This models a list of created processes where process 3 is created at time 0, is 85 MB in size, and needs 30 seconds running time to finish; process 1 is created at time 5, is size 100 MB, and needs 20 seconds of time to get its job done; process 6 is created at time 20, is size 225 MB, and requires the CPU for 15 seconds; process 10 is created at time 24, is size 50, and required the CPU for 14 seconds.

Once created, processes begin their life on disk. You do not have to worry about how processes get created in this simulation.

Points to note:

- The first process is always created at time zero.
- Each process id is a unique positive integer.
- A process id also represents the priority of the process, where lower process id indicates higher priority.
- Each process size is a positive integer $\leq m$ (the main memory size).
- The processes in the process file are in ascending order of the time they are created.

You may assume the input file being read in will always be in the correct format.

The simulation should obey the following cycle:

```
On E1 or E2 or E3
Swap(); Schedule()
```

Where `E1, E2` and `E3` are events of the form:
`E1` – A process has been created and memory is currently empty.
`E2` – The quantum has expired for the process running on the CPU;
`E3` – A process that was running on the CPU has called exit and terminated.

The swap function loads a process from disk into memory (if one exists). It will choose the process that has been sitting on the disk the longest (measured from the time it was created or swapped out, i.e. most recently placed on the disk), according to one of the three algorithms. If two or more processes have spent the same length of time sitting on the disk, the process with highest priority should be chosen.

The schedule function schedules another process to use the CPU, using a Round Robin strategy with quantum q. In particular, a process p whose quantum has just expired, should be placed at the end of the Round Robin queue. The process that has just been swapped in from disk (if any), should be placed either immediately before p, if p is still in the Round Robin queue, or otherwise it should be placed last in the the Round Robin queue.

The simulation should also obey the following:

- Assume memory is initially empty.
- A process that has terminated, should be removed from memory before swapping in the next process.
- If a process needs to be loaded into memory, but there is no hole large enough to fit it, then processes should be swapped out, one by one, until there is a hole large enough to hold the process needing to be loaded. If a process needs to be swapped out, choose the one which has been in memory the longest (measured from the time it was most recently placed in memory).
- When a process is swapped out of memory and placed on disk, it must also be removed from the Round Robin queue.
- The simulation should terminate once all processes have been created and have run to completion.

Your program should print out a line of the following form, each time a process is swapped into memory

```
time 42, 10 loaded, numprocesses=3, numholes=1, memusage=94%
```

where '`time`' refers to the time when the event happens, '`10`' refers to the id of the process just loaded, '`numprocesses`' refers to the number of processes currently in memory and '`numholes`' refers to the number of holes currently in memory. '`memusage`' is a (rounded up) integer referring to the percentage of memory currently occupied by processes.

Once the simulation ends, it should print a line of the following form:

```
time 79, simulation finished.
```

Note: a `diff` command will be used to check your program output against the expected output. Therefore, you should examine the sample output file – make sure your output matches the formatting exactly.

Your program must be called **swap** and the name of the process size file should be specified at run time using a '-f' *filename* option. The placement algorithm to be used should be specified using a '-a' *algorithm_name* option, where *algorithm_name* is one of {first,best,worst}. The size of main memory should be specified using a '-m' *memsize* option, where *memsize* is an integer (in MB). The length of the quantum should be specified using a '-q' *quantum* option, where *quantum* is an integer (in seconds).

Submission details

Please include your *name* and *user_id* in a comment at the top of each file.

Our plan is to directly harvest your submissions on the due date from your SVN repository.

`https://svn.eng.unimelb.edu.au/comp30023/username/project1`

You must submit program file(s), including a `Makefile`. Make sure that your makefile, header files and source files are added/committed. Do not add/commit object files or executables. Anything you want to mention about your submission, write a text file called README.

If you do not use your SVN repository for the project you will NOT have a submission and may be awarded zero marks. It should be possible to "checkout" the SVN repository, then type `make` to produce the executable `swap`.

Late submissions will incur a deduction of 2 mark per day (or part thereof).

If you submit late, you MUST email the lecturer, Michael Kirley <`mkirley@unimelb.edu.au`>. Failure to do will result in our request to sysadmin for a copy of your repository to be denied.

Extension policy: If you believe you have a valid reason to require an extension you must contact the lecturer, Michael Kirley <`mkirley@unimelb.edu.au`> or the head tutor `Friedrich Burkhard von der Osten <fvon@student.unimelb.edu.au>` at the earliest opportunity, which in most instances should be well before the submission deadline.

Requests for extensions are not automatic and are considered on a case by case basis. You will be required to supply supporting evidence such as a medical certificate. In addition, your SVN log file should illustrate the progress made on the project up to the date of your request.

Plagiarism policy: You are reminded that all submitted project work in this subject is to be your own individual work. Automated similarity checking software will be used to compare submissions. It is University policy that cheating by students in any form is not permitted, and that work submitted for assessment purposes must be the independent work of the student concerned.

Using SVN is an important step in the verification of authorship.

Assessment

Your submission will be tested and marked with the following criteria:

- 4 points for the *first fit* option giving the expected output.
- 4 points for *best fit* option giving the expected output.
- 4 points for *worst fit* option giving the expected output.
- 3 points for:
  - code successfully added to your SVN repository (`project1` folder); evidence of multiple commits / program development
  - `make` file submitted (and it works)
  - code runs on `dimefox.eng.unimelb.edu.au` or `nutmeg.eng.unimelb.edu.au` without *seg faulting* and/or getting stuck in an infinite loop
  - appropriate use of data structures; evidence of design efficiency
  - clarity of code – appropriate documentation included where necessary