COMP30017: Operating Systems and Network Services

Project 1 2013

Due date: No later than 5:00pm, Friday April 12 **Weight:** 15%

Project Overview

The aim of this project is to increase your familiarity with process scheduling and issues in memory management. A number of simplifying assumptions related to scheduling and memory management have been made. We do this, so that the workload in the project is commensurate with the marks (and time commitment) available.

There are two parts to the project: you will write a process scheduling simulator program in C, and then use simulation to investigate the performance of particular scheduling scenarios.

Part A

Your task is to write a simulator which takes processes of different sizes; loads them into memory when required; and when needed, swaps processes out to create a sufficiently large hole for a new process to come into memory. The scheduling of processes is carried out using a specific scheduling algorithm.

Your simulator must be written in C. We will supply skeleton code and header files (see LMS). You will have to provide the implementations for the functions described in `memory.h` and `algorithms.h`. Submissions that do not compile and run on the Department's student machines may receive zero marks.

Assume there are 2 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.

A detailed description of the scheduling algorithms to be used; the memory management mechanisms; process state transitions; and the simulation steps are provided below.

Scheduling algorithms

Four different queueing algorithms will be investigated as part of the simplified scheduling function in this project.

1. *First-come first-served queue*: The queue contains a list of ready processes to be scheduled. The queue type corresponds to a non-preemptive algorithm. That is, once a process begins executing it continues executing until the total running time elapsed reaches the specified job-time or the process is blocked on IO.

   If a process is blocked on IO, the CPU is given to next ready process in the queue. When the blocked process transitions to the ready (runnable) state, it is added to the end of the first-come first-served queue.

2. *Round robin queue*: The queue contains a list of ready processes to be scheduled. Each process is allocated a quantum of CPU time, $Q = 8$. Once a process begins executing, it continues executing until either its quantum expires; or it is blocked on IO; or it terminates.

   If a process is still running at the end of its quantum, the CPU is given to another process, and the old process is added to the end of the round robin ready queue.

If a process is blocked on IO, the CPU is given to next ready process in the queue. When the blocked process transitions to the ready (runnable) state, it is added to the end of the round robin ready queue.

3. *Multi-level feedback queue – np*: A simplified three-level *round robin* multi-level feedback queue contains a list of ready process to be scheduled.

   New processes are added to queue Q_1 with quantum = 4. After exhausting their quantum, if additional CPU time is required the process is subsequently moved to queue Q_2 with quantum = 8. (You may think of this as the process having a lower priority). If a process requires additional CPU time, it is moved to the final queue Q_3 with quantum = 16 (reflecting a lower priority). The *round robin* mechanism is then used for scheduling until all processes have terminated.

   Once a process begins executing, it continues executing until either its quantum expires; or it is blocked on IO; or it terminates.

   When a previously blocked process (due to an IO operation) transitions to the ready (runnable) state, it is added to the end of the appropriate queue. The process is moved to a higher priority level (queue with a smaller quantum, or if at the highest priority level already, use Q_1) than the queue it was originally in.

4. *Multi-level feedback queue – pre*: Once again, a simplified three-level *round robin* multi-level feedback queue containing a list of ready process to be scheduled will be used. This time Q_1, Q_2 and Q_3 have quantum's 4, 8 and 32 respectively.

   The basic operation is similar to the previous multi-level feedback queue. However one important enhancement is included: a currently running processes can be pre-empted by a higher priority process. Therefore, you will have to carefully monitor the contents of all queues at each time step.

   When a process is pre-empted, it is added to the end of the queue that the process was orginally in. IO blocked processes are dealt with in the same way as in multi-level feedback queue – np.

Memory

Processes have a constant size block of memory (the specific size depends on the process). Processes are moved between memory and disk as needed. It should be noted that processes are not moved "memory-to-memory," only "disk-to-memory" and removed from memory.

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). An appropriate data structure is used to track the locations of processes in memory and the free memory locations. 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.

When a process is moved from disk into memory, the *first fit* algorithm should be used. The *first fit* algorithm 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.

Process data file

A *process data file* is a sequence of entries that describes a list of processes to be used in your simulation.

The process data file has a text file format (Unix line endings) with one process per line. Each process has a start time, PID and memory size, then the word "start", followed by the job-time (or runtime) and optional IO blocking time(s) and subsequent runtime(s), then the word "end." An example data file is described on the next page.

For example:

```
0 1 100 start 100 end
10 2 120 start 50 30 15 10 20 end
20 3 70 start 60 end
25 4 100 start 40 end
```

At time 0, process PID=1 is created, it is 100 MB in size and requires 100 units of running time to finish. At time 10, process PID=2 is created, it is 50 MB in size and initially requires 50 units of running time; it is then blocked on IO for 30 time units; it then requires an additional 15 time units of running time before blocking on IO for another 10 units of time, finally it requires 20 units of time to finish. At time 20, process PID=3 is created, it is 70 MB in size and requires 60 units running time to finish. At time 25, process PID=4 is created, it is 100 MB in size and requires 40 units running time to finish.

Important things to note about the process data files: the first process is always created at time 0; the input file is sorted by process creation time then PID; PIDs are positive integers; process memory size $\leq$ main memory size (a command line parameter in MB); there is an odd number of run times and IO times between "start and "end" to facilitate smooth state transitions between queues in the simulation.

Once created, processes begin their life on disk. (You do not have to worry about how processes are created in this simulation). Newly created processes are inserted at the end of the appropriate queue (highest priority queue for the multi-level feedback queue models).

A parser for this input process data file will be provided. You may assume the input file being read in will always be in the correct format. You may also assume that the quantum values listed previously have the same units as the job-times listed in the process data file.

Process state transitions

A number of process state transitions are possible in this simulation. When particular events occur, the order in which you deal with the process transitions is important. Therefore, you must use the "hierarchy" list below to order the steps in your code when dealing with transitions events – this will ensure that any queues used will maintain processes in the correct order based on this project specification.

`running` $\rightarrow$ `terminated`

`blocked` $\rightarrow$ `ready (IO completed)`

`running` $\rightarrow$ `blocked (runtime completed, IO time starting)`

`new` $\rightarrow$ `ready`

`running` $\rightarrow$ `ready (end of quantum)`

`running` $\rightarrow$ `ready (preeemption)`

`ready` $\rightarrow$ `running (process scheduler)`

The IO blocked processes should be maintained using a list. Newly blocked processes are added at the end of the list.

Assume that IO blocked processes actually do their IO in parallel, that is, more than one process performs IO operations at the same time. Therefore, at each time step you should check the IO list and decrement the required IO time remaining for each process. You should also determine which processes transition from the blocked to the ready state (if any), and thus are transferred to the appropriate scheduling queue. You should work from the front of the list when checking the status of each process.

The simulation should behave as follows:

- Parse the process data file to obtain the lists of processes to be scheduled, then executed on the CPU, in your simulation.
- The schedule function (based on queue type) schedules which of the processes will execute next on the CPU. Only one process can execute on the CPU at any point in time. To use the CPU, the process must be in memory. Assume memory is initially empty.
- The simulation is time driven – use a timer/counter (starting from `time=0`) to control the overall simulation steps. At specific time steps, specific events will occur. For example, new processes arriving (based on the initial lists of processes read in from the process data file), processes moving from disk to memory, processes executing, processes terminating, processing blocking on IO etc.
- At each time step, identify which events have occurred and follow the "hierarchy" of process transitions above to determine the correct order in which the events will be handled.
- At each time step, check the list of IO blocked processes. If necessary, transfer the previously blocked process to the appropriate ready queue.
- At each time step, check the queue of processes to determine which process is to be run next. You should also check how long this process will run for (this will depend on the queue type used).
- If a process needs to be loaded into memory, use the *first fit* algorithm. If there is no hole large enough to fit the process, 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 the largest size. If two processes have equal largest size, choose the one which has been in memory the longest (measured from the time it was most recently placed in memory).
- If a process is blocked on IO, move it to the appropriate "blocked" list.
- When a process has executed for the total running time specified by its job-time, its image is removed from memory and the process is terminated.
- The simulation should finish when all processes listed in the process data file have terminated.
- Depending on the entries in the process data file, it is possible that the CPU will be idle in some circumstances waiting for new processes to be created.

Your program should print out a line of the following form, each time a process enteres the running state:

```
time 20, 15 running, numprocesses=3, numholes=2, memusage=77%
```

where '`time`' refers to the time when the event happens, '`15`' refers to the PID of the process that is now running, '`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 2000, simulation finished.
```

Your executable program must be called `sched`. (Note: see Makefile and skeleton code).

The name of the process data file should be specified at run time using a '-f' *filename* option. The scheduling algorithm to be used should be specified using a '-a' *algorithm_name* option, where *algorithm_name* is one of {fcfs, rr, mlfq_pre, mlfq_np} corresponding to the *first come first served, round robin* and the two *multi-level feedback queues* respectively. The size of main memory should be specified using a '-m' *memsize* option, where *memsize* is an integer (in MB).

Implementation details

You should use the skeleton code supplied and provide appropriate implementations for the functions described in the header files `memory.h` and `algorithms.h`. That is, you should write the files `memory.c` and `algorithms.c` that implement the necessary functions.

You must make use of your SVN repository (address below) when developing your solution.

`/home/studproj/student_repositories/`*`username`*`/`

```
trunk
trunk/comp30017
trunk/comp30017/project1
```

You should add the skeleton code (and Makefile) to your subversion repository for the project. As you modify `memory.c` and `algorithms.c` they should be added/checked-in to your repository.

Part B

In this task, you will compare the performance of alternative scheduling algorithms across a range of different process data files. This task is more open-ended that Part A and will require some planning on your part. You do not have to use your simulator program from Part A to complete this task. However, a working simulator will certainly help!

Simulation experiments

You should design and perform appropriate simulation experiments (by changing the command line arguments); collect performance metrics (eg. average waiting time, average turnaround time); and carry out statistical tests.

For example, you could investigate short run time jobs (eg. those produced by running shell scripts), short runtime/short IO time jobs (eg. text editing waiting for user input), long running jobs with some IO time (eg. running gzip on big input), long running jobs without IO. Obviously, there are many other simulation experiments that you could try. It is up to you to determine the most appropriate simulation experiments so that you drawn informed conclusions.

Note: we will **not** be supplying any additional process data files apart from the samples provided on the LMS. We encourage you to develop your own process data files (with specific characteristics) and post them on the LMS.

Report

You should write a report describing the outcomes of your simulation experiments. Your report, named `report.txt`, will be ≤ 800 words (approximately ≤ 7500 characters) in length using plain text format. Ideally, your report will include a clear description of your simulation experiments; table(s) of results and a discussion of your findings.

You must add the file `report.txt` to your SVN repository.

Assessment

Submission details

Our plan is to directly harvest your submissions on the due date from your SVN repository. Therefore, 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 subversion repository, then type `make` to produce the executable `sched`. We will then run your program with appropriate command line arguments.

Mark allocation

This project is worth 15% of your final mark for the subject. Your submission will be tested and marked with the following criteria:

- Part A
  - 8 marks for passing test cases (2 test cases for each scheduling algorithm)
    A range of test cases of varying complexity will be used when assessing your submission. The tests will examine whether your *first fit* memory management implementation functions as specified for each of the queueing models.
  - 2 marks for clear and succinct design and implementation
- Part B
  - 5 marks for the report. The following criteria will be used when assessing your report:
    * you have designed and implemented appropriate simulation experiments
    * you have clearly presented the results of your investigation using appropriate tables and statistical analysis
    * you have discussed the results and drawn appropriate conclusions based on these results
    * your writing demonstrates a well-expressed, clearly proof-read and coherent development of ideas

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

If you submit late, it is very important that you email the head tutor, Tom Gale <`tom@tg.id.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 head tutor or the lecturer 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 may be required to supply supporting evidence such as a medical certificate.

Plagarism 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.

Questions about the project specification should be directed to the LMS discussion forum.