COMP30023 Project 2
Process and Memory Management

Out date: May 15, 2020
Due date: No later than 15:59 June 5, 2020

Weight: 25% of the final mark. *Please refer to the LMS announcement for the changes made to the assessment percentage.*

Background

In this project you will familiarise yourself with process scheduling and memory management. You will write a simulator that allocates processes to a CPU and manages memory allocation among the running processes. You will also get to experiment with several algorithms in order to see under which scenarios some scheduling and memory allocation choices perform better than others. Finally, you will be required to improve over baseline algorithms specified in the project spec by implementing algorithms of your choice and justifying why you chose them.

Many details on how scheduling and memory allocation work have been omitted to make the project fit the timeline.

This project has higher weight than the first project. Hence, we advise you to start working on it soon after you finish reading the spec.

1 Process Scheduler

The program will be invoked via command line. It will be given a description of arriving processes, system properties and algorithm names to be used for scheduling these processes on CPU and memory allocation. You do not have to worry about how processes get created or executed in this simulation. The first process always arrives at time 0 which is when your simulation begins.

Scheduling processes Processes will be allocated CPU time depending on the time when they arrive and a scheduling algorithm. You will implement two baseline scheduling algorithms:

First-come first-served (`ff`) This is a "non-preemptive" algorithm which means that once a process begins executing, it continues executing until its total running time reaches the specified job-time. Every process that arrives starts executing only after every process that arrived before it has completed. If more than one process arrives at the same time, they are executed in increasing order of their process-ids.

Round-robin (`rr`) Every process is executed on a CPU for a limited time interval at a time, called quantum. Once process's quantum has elapsed or the process has completed, the CPU is given to the next process in the queue. Any new process that has arrived in the meantime is placed in the end of the queue. If more than one process arrives at the same time, they are placed in the queue in the increasing order of their process id (i.e., the process with the smallest id would be executed first). The process whose quantum has just elapsed is suspended and placed in the queue unless it has completed. Note that if the quantum of process p has elapsed at time t then any uncompleted process that has arrived before or at time t will be given CPU time before p's next quantum.

The simulation should terminate once all processes have run to completion.

2 Memory Management

When the process arrives, all pages that it requires are initially stored on disk. Before the process can be executed on the CPU, it needs to be given space in memory (RAM) in order to copy some or all of its pages from disk to memory.

Note: The "unlimited memory" option lets you work on scheduling algorithms, finish and test them before moving to this part of the project. If the memory is unlimited then memory management and any time delays associated with it should be ignored (i.e., there can be unlimited number of processes in memory).

The maximum memory size available to CPU will be indicated through a command line argument. You can assume that the memory is empty at the beginning of your simulation. Memory will be split into 4KB pages. You are required to implement two strategies for allocating memory to a process:

Swapping-X (`p`) [1] The process can be executed only if all pages it requires are in memory. If there are not enough empty pages to fit a process, then pages of *another* process or processes need to be swapped to disk to make space for this process. When choosing a process(es) to swap, choose the one that was least-recently-executed among other processes (excluding the current one) and evict all of its pages (as a result this strategy evicts least-recently used pages). If there is still not enough space, continue evicting other processes following the same policy on least-recently-executed process until there is sufficient space.

In order to simulate the time it takes to load a process in memory, the load time of the process is calculated by adding 2 seconds[2] for every page required by the process.

Example: If the process's memory requirement is 32KB and its pages are not in memory, then the load time is $2 \times 32\text{KB}/4\text{KB}$.

Virtual memory (`v`) Since virtual memory mimics availability of larger memory, we will assume that a process can be executed if it is allocated at least 16KB of its memory requirement (i.e., 4 pages) or all memory it requires if its requirement is less than 16KB. However, if there are more empty pages available, the process should be given either all of the empty pages or enough to meet its memory requirements (e.g., if the process requires 7 pages and there are 6 empty pages then it should be given all 6 pages; if the process requires 7 pages and there are 10 empty pages, then it should be given 7 of these pages).

Similar to swapping, if there are not enough empty pages for the process that is scheduled to be executed, pages of the least-recently-executed process need to be evicted *one by one* until there are 4 empty pages or less, if the process needs less memory. The lowest number pages belonging to the processes must be evicted first (e.g., if the least-recently-executed process was allocated pages 1,5,7,9, and 2 pages need to be evicted, pages 1,5 must be evicted). In contrast, for the swapping option *all* pages of the least-recently-executed process would be removed.

The load time of the process is computed in the same manner as for the swapping setting above: 2 seconds for every loaded page.

In order to simulate the time it takes to serve a page fault (i.e., due to accesses to pages that have not been loaded) during process execution, add the following penalty to the remaining execution time of the process: 1 second for every page of the process that is not in memory. You do not need to worry about swapping the pages of the same process and handling page faults.

Example: Consider a process with memory requirement of 32KB. If half of it is already in memory and it has 16 seconds remaining till completion when it starts executing, then its remaining running time is extended to $16 + 16\text{KB}/4\text{KB}$.

The simulation should also obey the following:

- Each page is numbered, starting from 0. For example, if the total memory size is 200KB then there are 50 pages in total with page numbers from 0 to 49.

[1] This option is a relaxation of "Swapping" as defined in the lecture where the process has to be allocated to a continuous block of memory.

[2] Though we use seconds in the simulation, think of them simply as time units. In real systems, these events are orders of magnitude faster.

- Each process is allocated pages in increasing order of the page numbers, that is the first process that arrives is always given a memory page 0.
- The delay for both options is added for every page that is loaded regardless of whether there was already an empty spot or eviction was required.
- A process that has 0 seconds left to run, should be 'evicted' from memory before marking the process as 'finished'.
- The quantum for `rr` starts after the process's pages have been loaded. Quantum is subtracted from the remaining execution time that includes page fault penalties. You can assume that we will not give you processes that would run infinitely due to execution time continuing to increase from page faults.

Note that with first-come first-served scheduling process's pages are loaded into memory only once at the beginning of its execution and, similarly, removed only once immediately before it is marked as 'finished'.

3 BYO Algorithms

You will notice that the algorithms that you are required to implement for scheduling and memory allocation may not be ideal for certain workloads in terms of completion time. We will refer to these algorithms as *baseline* algorithms. To this end, you are asked to implement:

Customised Scheduling (`cs`) a third scheduling algorithm that is different from first-come first-served and round-robin algorithms. Note that round-robin algorithm with a different quantum time does not qualify as a different algorithm.

Customised Memory Management (`cm`) a memory replacement policy that is different from least-recently-executed process but follows the same time penalties for loading pages and page faults as described in Section 2.

Remember to state assumptions and hypotheses

In addition to implementing the algorithms, you will need to justify your choices with a short report (**maximum length of 200 words**) articulating for each algorithm (1) its short description; (2) scenarios where it performs better than the baselines and why; (3) scenarios where it does not perform well and why (if applicable). In order to support your choice, you are also required to submit two benchmarks on which your algorithms outperform the baseline algorithms. That is, one of the benchmark files should demonstrate the superiority of the scheduling algorithm that you choose over `rr` and `ff`, and another one would show the superiority over the swapping and virtual memory configurations. The superiority is defined through improvements on performance statistics as described in Sections 5 and 6.

Note that (1) you can implement any of the algorithms you find in the textbook or online as long as you can justify why you chose them; (2) you can make progress on this part of the project independent of `ff,rr` and with partially finished `p,v` algorithms.

4 Program Specification

Your program must be called `scheduler` and take the following command line arguments. The arguments can be passed in any order but you can assume that they will be passed exactly once and optional parameter `-q` at most once.

`-f` ***filename*** will specify the name of the file describing the processes.
`-a` ***scheduling-algorithm*** where *scheduling algorithm* is one of {`ff`,`rr`,`cs`} where `cs` is the third scheduling algorithm that you will choose to implement.
`-m` ***memory-allocation*** where *memory-allocation* is one of {`u`,`p`,`v`,`cm`} where `u` indicates unlimited memory and `cm` is the process/memory replacement algorithm that you will implement.
`-s` ***memory-size*** where *memory-size* is an integer indicating the size of memory in KB. This option can be ignored in the case of unlimited memory, i.e., when `-m u`.
`-q` ***quantum*** where *quantum* is an integer (in seconds). The parameter will be used only for round-robin scheduling algorithm with the default value set to 10 seconds.

The *filename* contains the processes to be executed and has the following format. Each line of the file corresponds to a process. The first entry refers to the first process that needs to be executed, and the last entry refers to the last process to be executed. Each entry consists of a tuple (*time-arrived*, *process-id*, *memory-size-req*, *job-time*). You can assume that the file will be sorted by *time-arrived* which is an integer in $[0, 2^{32})$ indicating seconds; all *process-id*s will be distinct integers in the domain of $[0, 2^{32})$ and the first process will always have time-arrived set to 0; *memory-size-req* will specify memory requirement of the process in KB and will be divisible by 4, you can assume it will be always smaller than *memory-size* and will be an integer in $[4, 2^{32})$; *job-time* will be an integer in $[1, 2^{32})$ indicating seconds. More than one process can arrive at the same time.

Example: `./scheduler -f processes.txt -a ff -s 200 -m p`.

The scheduler is required to simulate execution of processes in the file `processes.txt` using first-come first-served algorithm assuming access to 200KB of memory.

Given `processes.txt` with the following information

```
0 4 96 30
3 2 32 40
5 1 100 20
20 3 4 30
```

the program should simulate execution of 4 processes where process 4 arrives at time 0, requires 96KB in size, needs 30 seconds running time to finish; process 2 arrives at time 3, is size 32KB, and needs 40 seconds of time to complete its job; etc.

Though you can read the whole file before starting the simulation, your simulation is required to schedule processes in an online manner. That is, it should mimic the real world where the OS schedules processes as they come and cannot see into the future.

5 Expected Output

In order for us to verify that your code meets the above specification, it should print to standard output (`stderr` will be ignored) information regarding the states of the system and statistics of its performance. All times are to be printed in seconds. The output below uses spaces and `\n` for new lines.

Execution transcript For the following events the code should print out a line in the following format:

- When a process starts and every time it resumes its execution:

  `current_time, RUNNING, id=<process-id>, remaining-time=<T_rem>, load-time=<T_load>, mem-usage=<mem_usage>%, mem-addresses=[<set_of_pages>]\n`

  where:
  - '`current_time`' refers to the time at which CPU is given to a process but *before* the process's pages have been loaded, if loading is required;
  - '`process-id`' refers to the *process-id* of the process that is about to be loaded/run;
  - '`T_rem`' refers to the time required until the process is finished including the time taken for any potential page faults;
  - '`T_load`' refers to the time it takes to load process's pages in memory, if loading is required;
  - '`mem_usage`' is a (rounded up) integer referring to the percentage of memory currently occupied by processes, after `process-id` has been loaded;
  - '`set_of_pages`' is a list of page addresses (given in increasing order) that are allocated to the current process, separated by commas.

  If the memory given to the CPU is unlimited (i.e., `-m u`) then your program should not print out any information about memory allocation or usage. That is, memory loading and page fault time is not considered in this mode: assume that all programs are already in memory. Simplified output would be:
  `20, RUNNING, id=15, remaining-time=10`

- Every time memory pages are evicted or deallocated from memory:

  `current_time, EVICTED, mem-addresses=<[set_of_pages]>\n`

  where:
  - '`current_time`' is as above for the `RUNNING` event;
  - '`set-of-pages`' refers to the list of page addresses (given in increasing order), separated by commas, that are evicted.

  Only *one* `EVICTED` event should be printed out, i.e., even if pages of multiple processes are being evicted. That is, your program should never print two `EVICTED` events in two consecutive lines. If memory is set to 'unlimited', no `EVICTED` events should be printed.

- Every time a process finishes:

  `current_time, FINISHED, id=<process-id>, proc-remaining=<num_proc_left>\n`

  where:
  - '`current_time`' is as above for the `RUNNING` event;
  - '`process-id`' refers to the *process-id* of the process that has just been completed;
  - '`num_proc_left`' refers to the number of processes that are waiting to be executed (i.e., those that have arrived but not yet completed).

Note that `EVICTED` and `FINISHED` events do not incur time. Hence, lines following these event lines may begin with the same '`current_time`'. If the eviction resulted due to process completion, `EVICTED` line precedes `FINISHED`.

Example: Consider a process that has 10 seconds left to completion and 20KB memory requirement. If it has pages [0,1] already loaded and only pages [6,7] are empty in memory, then the following lines may be printed with `-a rr -m v -q 11`:

```
20, RUNNING, id=15, remaining-time=11, load-time=4, mem-usage=100%, mem-addresses=[0,1,6,7]
35, EVICTED, mem-addresses=[0,1,6,7]
```

Performance statistics When the simulation is completed, 4 lines with the following performance statistics about your simulation performance should be printed:

- Throughput: average (rounded up to an integer), minimum and maximum number of processes completed in sequential non-overlapping 60 second intervals, with the first interval starting at 1. The intervals are thus [1,60], [61,120], [121,180] etc. Hence, if a process `FINISHED` at time 60, it is included in the first interval.
- Turnaround time: average time (in seconds, rounded up to an integer) between the time when the process completed and when it arrived;
- Time overhead: maximum and average time overhead when running a process, both rounded to the first two decimal points, where overhead is defined as the turnaround time of the process divided by its job-time.
- Makespan: the time in seconds when your simulation ended.

Example: For the invocation with arguments `-a ff -m u` and the processes file as described above, the simulation would print

```
Throughput 2, 1, 3
Turnaround time 71
Time overhead 4.25 2.56
Makespan 120
```

6 Marking Criteria

The marks are broken down as follows:

Marks	Task
2	First-come first-served (`ff`)
4	Round-robin (`rr`)
4	Swapping (`p`)
3	Virtual memory (`v`)
3	Own scheduling algorithm (`cs`)
3	Own page swapping algorithm (`cm`)
2	Section on `cs` in the report
2	Section on `cm` in the report
1	Quality of software practices
1	Build quality

Evaluation of algorithms We will run all baseline algorithms against our test cases. Your own algorithms will be checked for correctness (e.g., to ensure that no two processes are allocated the same page address at the same time, time delays are added appropriately and all processes are executed to completion) and how well they perform compared to the baselines w.r.t. `Time overhead`s. We will use your benchmarks to evaluate the latter.

Benchmarks Please submit the following files:

- `benchmark-cs.txt` is a file describing the processes (following the format of *filename* given using **-f** argument) on which invocation of
  ```
  ./scheduler -f benchmark-cs.txt -a cs -s 100 -m v
  ```
  would show a lower maximum and average `Time overhead`s than invocations of either of the baselines:
  ```
  ./scheduler -f benchmark-cs.txt -a rr -s 100 -m v -q 10
  ./scheduler -f benchmark-cs.txt -a ff -s 100 -m v
  ```
- `benchmark-cm.txt` is also a file describing the processes (following the format of *filename*) on which invocation of
  ```
  ./scheduler -f benchmark-cm.txt -a rr -s 100 -m cm -q 10
  ```
  would show a lower maximum and average `Time overhead`s than invocations of either of the baselines:
  ```
  ./scheduler -f benchmark-cm.txt -a rr -s 100 -m v -q 10
  ./scheduler -f benchmark-cm.txt -a rr -s 100 -m p -q 10
  ```

We will run our version of the baselines on these benchmarks and use your code with `cs` and `cm` options. Each of your benchmark files should contain 10 processes or more.

Reports Reports longer than 200 words will automatically get 0 marks. Hence, ensure that you get 200 or less words when you run `wc -w report.txt`. The report should be in ASCII format.

Quality of software practices Factors considered include **quality of code**, based on the choice of variable names, comments, indentation, abstraction, modularity, and **proper use of version control**, based on the regularity of commit and push events, their content and associated commit messages (e.g., repositories with a single commit and push, non-informative commit messages, or submissions without `submission` tag will lose 0.5 mark).

Build quality Running `make clean && make && ./scheduler <command line arguments>` should execute the submission. If this fails for any reason, you will be told the reason, and allowed to resubmit (with the usual late penalty) but lose the build quality marks.

Compiling using "`-Wall`" should yield no warnings.

7 Submission

All code must be written in C (e.g., it should not be a C-wrapper over non C-code) and cannot use any external libraries. Your code will likely rely on data structures for managing processes and memory. You will be expected to write your own versions of these data structures. You may use standard libraries (e.g. to print, read files, sort, parse command line arguments[3] etc.) Your code must compile and run on the provided VMs.

You must push your submission to the repository named `comp30023-2020-project-2` in the subgroup with your username of the group `comp30023-2020-projects` on `https://gitlab.eng.unimelb.edu.au`. Do not forget to include `report.txt`, `benchmark-cs.txt` and `benchmark-cm.txt` in the project root directory of your repository.

We will harvest this repository on the due date (and on each subsequent day for late submissions). You are encouraged to complete the project and may continuously submit well beyond the official deadline. As per our grading calculation policy in Project 1, we will assign your grade according to your best submission. We decide your best submission by taking the `MAX(submission (Day 0), HEAD Master (Day 0), submission (Day 1)`[4]`, HEAD Master (Day 1)...)`. If you do not use your git repository for the project you will not have a submission and will be awarded zero marks.

The repository must contain a Makefile that produces an executable named "`scheduler`", along with all source files required to compile the executable. Place the Makefile at the root of your repository, and ensure that running `make` places the executable there too. Make sure that your Makefile, source code are committed and pushed. Do not add/commit object files or executables.

Place a git tag named `submission`[5] on the snapshot you wish to be marked. Please see lab sheet for week 6 on how to create a git tag.

For your own protection, it is advisable to commit your code to git at least once per day. Be sure to `push` after you `commit`. You can push debugging branches if your code at the end of a day is in an unfinished state, but make sure that your final submission is in the `master` branch.

Do not leave submission to the last minute to avoid unforeseen crashes on Gitlab due to server load. You are strongly advised to submit hours, if not days prior to the official deadline.

The git history may be considered for matters such as special consideration, extensions and potential plagiarism. Your commit messages should be a short-hand chronicle of your implementation progress and will be used for evaluation in the Quality of Software Practices criterion.

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

Extension policy: If you believe you have a valid reason to require an extension you must contact the subject coordinator, Olya Ohrimenko, at least a week before the due date with a supporting evidence (e.g., a medical certificate). Extension **will not be** considered otherwise. Note: **you have been allocated 3 weeks to complete the project**. Requests for extensions are not automatic and are considered on a case by case basis.

8 Testing

You have access to several test cases with expected outputs. However, these test cases are not exhaustive and will not cover all edge cases. Hence, you are also encouraged to write more tests to further test your own implementation.

Testing Locally You can clone the sample testcases to test locally from `comp30023-2020-projects/project-2`. You are welcome to write scripts for testing your program using these testcases.

Continuous Integration Testing Like Project 1, you can also use Gitlab CI to receive some limited feedback on your progress before the deadline. Though you are encouraged to use this service, the usage

[3] `https://www.gnu.org/software/libc/manual/html_node/Getopt.html`

[4] Days greater than 0 will incur late penalties, unless an extension has been granted

[5] Git tags are case-sensitive, the tag must be in lowercase

of CI is not examinable, i.e., we do not require CI tasks to complete for a submission to be considered for marking.

Note that the test cases which are available on Gitlab are the same as the set described above.

The requisite `.gitlab-ci.yml` file has been provisioned and placed in your repository, but is also available from the `project-2` repository linked above. You simply need to add code to your repository, commit and then push to trigger build and tests.

9 Collaboration and Plagiarism

You can discuss this project abstractly with your classmates, but should not share code. If possible, do not even look at anyone else's code. If you really want to help someone debug, do not look at the screen and pretend you are doing it over the phone.

You are welcome to use code fragments from Stack Exchange, or any other source of information on the web, but be sure to quote the URL whose code you used, so that it does not appear that you copied a classmate. Also, be warned that much of the code on question and answer sites is buggy. If you do reuse code, check it carefully to make sure it does what it claims to.

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 git is an important step in the verification of authorship.