COMP30023 Project 1
Process* Scheduling

Out date: March 15, 2021
Due date: No later than 14:59 March 30, 2021 AEDT

Weight: 15% of the final mark

Background

In this project you will familiarise yourself with process scheduling in a multi-processor environment. You will write a simulator that allocates CPU (or processor) to the running processes.

Many details on how scheduling works have been omitted to make the project fit the timeline. Please note that you do not need multiple processes or threads (e.g., no need for `fork` or `pthread`) to implement this project. *We strongly advise you to start working on the project as soon as possible.*

The program will be invoked via the command line. It will be given a description of arriving processes including their arrival time, execution time in seconds, their id, and whether they are parallelisable or not. 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 the simulation begins.

1 Single processor scheduler

Processes will be allocated CPU time according to **shortest-time-remaining** algorithm, a preemptive scheduling algorithm. When a new process arrives, it is scheduled if there is no other process running. If, when process j arrives, there is a process i running, then i is postponed and added to a queue if and only if j has a shorter execution time than the remaining time of i. Process i is resumed where it left off, if it is the process with the shortest remaining time left among all other processes in the queue (i.e., including those that may have arrived after j). Process ids are used to break ties, giving preference to processes with smaller ids. Since there is only one CPU, you can ignore the information on whether a process is parallelisable or not.

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

2 Two processor scheduler

In this section you are asked to extend your scheduler to work with 2 processors: processors numbered 0 and 1. This will simulate (1) a situation where processes can run truly in parallel and (2) a process that can run faster with more computational resources (e.g., by creating child/sub processes).

If the process that arrives is not parallelisable, it is assigned to the queue of the CPU with the smallest amount of remaining execution time for all processes and subprocesses (defined below) allocated to it, using CPU id to break ties, giving preference to CPU 0. After that the simulation for this process behaves same as in Section 1. In this project, once a process is assigned to a CPU it cannot be moved to another CPU. (*Think of why it may not be a good idea to migrate processes between CPUs.*)

If arriving process with id i and execution time x is parallelisable, two *subprocesses* are created, $i.0$ and $i.1$, each with execution time $\lceil x/2 \rceil + 1$. The extra time simulates the cost of synchronisation. For example, a parallelisable process that runs on a single CPU in 6 seconds, can finish within 4 seconds if both CPUs are idle when it arrives.

Once subprocesses are created, subprocess $i.0$ is added to the queue of CPU 0 and $i.1$ is added to CPU 1. After that they are treated as regular processes when scheduling them on each CPU. A parallelisable process is considered finished when both of its subprocesses have finished.

3 N processor scheduler

In this task we generalise the 2 processor setting to $N \geq 3$ processors. Similar to above, a non-parallelisable process is assigned to a CPU that has the shortest remaining time to complete all processes and subprocesses assigned to it so far.

A parallelisable process is split into $k \leq N$ subproceses, where k is the largest value for which $x/k \geq 1$. Each subprocess is then assigned execution time of $\lceil x/k \rceil + 1$. Subprocesses follow a similar naming convention as above: a process with id i is split into subprocesses with id's $i.0$, $i.1$, ..., $i.k'$, where $k' = k - 1$. Subprocesses are then assigned to k processors with shortest time left to complete such that subprocess $i.0$ is assigned to the CPU that can finish its processes the fastest, $i.1$ is assigned to the second fastest processor and so on. Use CPU ids to break ties, giving preference to smaller processor numbers.

Example: consider a scenario with 4 CPUs, each with one process left to complete; processes on CPU 0,1,2,3 have remaining time of 30,20,20,10, respectively. Then for a process i with execution time of 2, k is set to 2. Its subprocess $i.0$ and $i.1$ will be assigned to CPU 3 and 1 respectively.

Once a process or subprocess is assigned to a CPU it cannot migrate to another CPU. A parallelisable process is considered finished when all of its subprocesses have finished.

4 Challenge: Improve the performance

You will be asked to measure the overall time of your simulation (`Makespan`). The challenge task is to come up with an algorithm that has a shorter makespan on a set of tests (see details in Section 7). For this task the choice of k when splitting a parallelisable process is left to you. You are also allowed to "look into the future" and see what processes will be arriving and use this information if you choose to. (*Think of whether it is possible and how one would obtain such information in real life.*) You will be required to explain why your algorithm is more efficient in a short report.

5 Program Specification

Your program must be called `allocate` 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.

`-f` ***filename*** will specify the name of the file describing the processes.
`-p` ***processors*** where *processors* is one of {`1`,`2`,`N`}, `N` ≤ 1024.
`-c` an optional parameter, when provided, invokes your own scheduler from Section 4.

The *filename* contains the processes to be executed and has the following format. Each line of the file corresponds to a process. The first line refers to the first process that needs to be executed, and the last line refers to the last process to be executed. Each line consists of a space-separated tuple (*time-arrived*, *process-id*, *execution-time*, *parallelisable*). 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; *execution-time* will be an integer in $[1, 2^{32})$ indicating seconds; *parallelisable* is either `n` or `p`. If it is `p`, then the corresponding process is parallelisable; if it is `n`, it is not. You can ignore `n`/`p` when `-p` is 1. More than one process can arrive at the same time.

Example: `./allocate -f processes.txt -p 1`.

The allocation program is required to simulate execution of processes in the file `processes.txt` on a single CPU.

Given `processes.txt` with the following information:

```
0 4 30 n
```

```
3 2 40 n
5 1 20 n
20 3 30 n
```

The program should simulate execution of 4 processes where process 4 arrives at time 0, needs 30 seconds running time to finish; process 2 arrives at time 3 and needs 40 seconds of time to complete.
Each line (including the last) will be terminated with a LF (ASCII `0x0a`) control character.

You can read the whole file before starting the simulation or read one line at a time. We will not give malformed input (e.g., negative number of processors after `-p` or more than 4 columns in the process description file).

6 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.

6.1 Execution transcript

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

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

  `<current-time>,RUNNING,pid=<process-id>,remaining_time=<T>,cpu=<cpu-id>\n`

  where:
  '`current-time`' refers to the time at which CPU is given to a process;
  '`process-id`' refers to the *id* of the process that is about to be run;
  '`T`' refers to the remaining execution time for this process;
  '`cpu-id`' refers to the processor where the process is scheduled on. It can be $0, 1, 2 \ldots, N-1$ when `-p` N for $N \geq 1$;

  Sample output could be:
  `20,RUNNING,pid=15,remaining_time=10,cpu=0`
- Every time a **process** finishes:

  `<current-time>,FINISHED,pid=<process-id>,proc_remaining=<num-proc-left>\n`

  where:
  - '`<current-time>`' is as above for the `RUNNING` event;
  - '`process-id`' refers to the *id* of the process that has just been completed;
  - '`num-proc-left`' refers to the number of **processes** that are waiting to be executed over all processors (i.e., those that have already arrived but not yet completed, including those that have unfinished subprocesses).

If there is more than one event to be printed at the same time: print `FINISHED` before `RUNNING` and print events for smaller CPU ids first.

Example: Consider the last remaining process which has 10 seconds left to completion. Then the following lines may be printed:

```
20,RUNNING,pid=15,remaining_time=10,cpu=1
30,FINISHED,pid=15,proc_remaining=0
```

6.2 Performance statistics

When the simulation is completed, 3 lines with the following performance statistics about your simulation performance should be printed:

- 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 total execution time (i.e., the one specified in the process description file).
- Makespan: the time in seconds when your simulation ended.

Example: For the invocation with arguments `-p 1` and the processes file as described above, the simulation would print

```
Turnaround time 62
Time overhead 2.93 1.9
Makespan 120
```

7 Marking Criteria

The marks are broken down as follows:

Task # and description	**Marks**
1. Single processor (Section 1)	2
2. 2 processors, non-parallelisable processes only	2
3. 2 processors, parallelisable and non-parallelisable processes	2
4. N processors, non-parallelisable processes only	2
5. N processors, parallelisable and non-parallelisable processes	2
6. Performance statistics computation (Section 6.2)	1
7. Challenge task (Section 4)	2
8. Quality of software practices	1
9. Build quality	1

Tasks 1-6. We will run all baseline algorithms against our test cases. You will be given access to half of these test cases and their expected outputs. Half of your mark for the first 6 tasks will be based on these known test cases. Hence, we strongly recommend you to test your code against them. The first 5 tasks will be marked based on the execution transcript (Section 6.1) and task 6 based on performance statistics (Section 6.2).

Task 7. The challenge task, as described in Section 4, will be evaluated based on a set of 9 test cases provided to you and your report. We will use these test cases to compare the `Makespan` of your scheduling algorithm (with `-c` option) with the default one. A scheduling algorithm that improves `Makespan` (i.e., results in a shorter `Makespan`) over the default algorithm on 2-4 of the 9 test cases will be awarded 0.5 marks, 1 mark for improving on 5-7 test cases and 1.5 marks for improving on 8-9 test cases. We may use a verification algorithm on your transcript to verify that your scheduler is sound (e.g., that every process is given CPU time as per its execution time requirements and does not start before its arrival time). You will not be provided with expected output of the default scheduler for test cases in this task.

Please submit a *report* explaining your scheduling algorithm and why it performs better than the default one. Reports longer than 50 words will automatically get 0 marks. Hence, ensure that you get 50 or less words when you run `wc -w report.txt`. The report should be in ASCII format. The report will get .5 of a mark if your algorithm improves a makespan on at least one of the 9 test cases.

Task 8. 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 will lose 0.5 mark).

Task 9. Build quality Running `make clean && make -B && ./allocate <command line arguments>` should execute the submission. If this fails for any reason, you will be told the reason, and be allowed to resubmit (with the usual late penalty). Compiling using "`-Wall`" should yield no warnings.

Do not commit `allocate` or any other executable file (see Practical 2). A 0.5 mark penalty will be applied if your final commit contains any such files. Executable scripts (with `.sh` extension) are excepted.

The automated test script expects `allocate` to exit/return with status code of 0 (i.e. it successfully runs and terminates).

We will not give partial marks or allow code edits for either known or hidden cases without applying late penalty (calculated from the deadline).

8 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[1] etc.). Your code must compile and run on the provided VMs and produce deterministic output.

The repository must contain a Makefile which produces an executable named "`allocate`", 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 all source code is committed and pushed. **Executable files** (that is, all files with the executable bit which are in your repository) **will be removed** before marking. Hence, ensure that none of your source files have the executable bit.

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 on the `master` branch. 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.

You must **submit the full 40-digit SHA1 hash** of your chosen commit to the **Project 1 Assignment** on LMS. You must **also push your submission** to the repository named `comp30023-2021-project-1` in the subgroup with your username of the group `comp30023-2021-projects` on `gitlab.eng.unimelb.edu.au`. You will be allowed to update your chosen commit. However, only the last commit hash submitted to LMS before the deadline will be marked without late penalty.

You should ensure that the commit which you submitted is accessible from a fresh clone of your repository. For example (below `...` are added for aesthetic purposes to break the line):

```
git clone https://gitlab.eng.unimelb.edu.au/comp30023-2021-projects/<username>/ ...
... comp30023-2021-project-1
cd comp30023-2021-project-1
git checkout <commit-hash-submitted-to-lms>
```

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, please fill in the form accessible on Project 1 Assignment on LMS. Extensions **will not be** considered otherwise. Requests for extensions are not automatic and are considered on a case by case basis.

9 Testing

You have access to several test cases and their 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.

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

Testing Locally: You can clone the sample test cases to test locally, from: `comp30023-2021-projects/project-1`.

Continuous Integration Testing: To provide you with feedback on your progress before the deadline, we have set up a Continuous Integration (CI) pipeline on Gitlab. Though you are strongly encouraged to use this service, the usage of CI is not assessed, 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-1` repository linked above. Please clone, commit and push to trigger.

10 Collaboration and Plagiarism

You may discuss this project abstractly with your classmates but what gets typed into your program must be individual work, not copied from anyone else. Do **not** share your code and do **not** ask others to give you their programs. Do **not** post your code on subject's discussion board Piazza. The best way to help your friends in this regard is to say a very firm "**no**" if they ask to see your program, pointing out that your "**no**", and their acceptance of that decision, are the only way to preserve your friendship. See `https://academicintegrity.unimelb.edu.au` for more information.

Note also that solicitation of solutions via posts to online forums, whether or not there is payment involved, is also Academic Misconduct. You should not post your code to any public location (e.g., `github`) while the assignment is active or prior to the release of the assignment marks.

If you use any code not written by you, you must attribute that code to the source you got it from (e.g., a book or Stack Exchange).

Plagiarism policy: You are reminded that all submitted project work in this subject is to be your own individual work. Automated similarity checking software may 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.