COMP30023: Computer Systems
Project 1: Process Management

Released: March 24, 2023 AEDT
Due: 9am April 17, 2023 AEST
Weight: 15%

1 Overview

In this project, you will implement a *process manager* capable of allocating memory to processes and scheduling them for execution. This project has two phases. In the first phase, a scheduling algorithm will allocate the CPU to processes, making the assumption that memory requirements are always satisfied. In the second phase, a memory allocation algorithm will be used to allocate memory to processes, before the scheduling takes place. Both the memory allocation and the process scheduling are **simulated**. There is also a *challenge task* that requires controlling the execution of real processes during the process scheduling. Process scheduling decisions will be based on a scheduling algorithm with the assumption of a single CPU (i.e, only one process can be running at a time).

2 The Process Manager

The process manager runs in **cycles**. A cycle occurs after one quantum has elapsed. The process manager has its own notion of time, referred to from here on as the *simulation time*. The simulation time starts at 0 and increases by the length of the quantum every cycle. Hence, at any given cycle, the simulation time is equal to the number of completed cycles times the length of the quantum:

$$T_S = N \times Q$$

where T_S is the current simulation time, N is the number of cycles that have elapsed, and Q is the length of the quantum. For this project, Q will be an integer value between 1 and 3 ($1 \leq Q \leq 3$).

At the start of each cycle, the process manager must carry out the following tasks in sequence:

1. Identify whether the currently running process (if any) has completed. If so, it should be terminated and its memory deallocated before subsequent scheduling tasks are performed.

2. Identify all processes that have been submitted to the system since the last cycle occurred and add them to the *input queue* in the order they appear in the process file. A process is considered to have been submitted to the system if its arrival time is less than or equal to the current simulation time T_s.

   The input queue contains processes that have been submitted to the system and are waiting to be brought into memory (from disk) to compete for CPU time. At no time should the input queue contain processes whose arrival time is larger than the current simulation time.

3. Move processes from the *input* queue to the *ready* queue upon successful memory allocation. The *ready* queue holds processes that are ready to run.

   The process manager will use ONE of the following methods to allocate memory to processes:

   - Assume that the memory requirements of the arrived processes are always immediately satisfied (e.g., there is an *infinite* amount of memory). All the arrived processes will automatically enter a *READY* state, and be placed at the end of the *ready* queue in the same order as they appear in the input queue. **OR,**
   - Allocate memory to processes in the *input* queue according to a memory allocation algorithm. Processes for which memory allocation is successful enter a *READY* state, and are placed at the end of the *ready* queue in order of memory allocation to await CPU time (Section 3.3).

4. Determine the process (if any) which runs in this cycle. Depending on the scheduling algorithm (Section 3.1, 3.2), this could be the process that was previously running, a resumed process that was previously placed back into the *ready* queue, or a *READY* process which has not been previously been executed.

After completing these tasks, the process manager should sleep for Q seconds, after which a new cycle begins.

2.1 Process Lifecycle

The lifecycle of a process is as follows:

1. The process is submitted to the manager via an input file (See Section 4 for more details). Note that you may read all the processes in the input file into a data structure and use said data structure to determine which processes should be added to the input queue based on their arrival time and the current simulation time.
2. The process is in a *READY* state after it has arrived (arrival time less than or equal to the simulation time), and memory has been **successfully** allocated to it.
3. Once a process is in the *READY* state, it can now be considered by a process scheduling algorithm as a candidate to enter the *RUNNING* state, i.e., be allocated to the CPU.
4. Once a process is in the *RUNNING* state, it runs on the CPU for at least one quantum. **If** the scheduling algorithm decides to allocate CPU time to a different process, the *RUNNING* process is suspended and placed in the *ready* queue. At this point, the state of the process will change from *RUNNING* to *READY*.
5. The total time a process gets to run on the CPU (i.e., is in the *RUNNING* state) must be equal to or greater than its *service time*. Note that the service time of a process is the amount of CPU time (not wall time) it requires before completion.
6. After the service time of a process has elapsed, the process moves to the *FINISHED* state. When a process terminates, its memory must be deallocated and it should cease competing for the CPU (must not enter the *ready* queue).

2.2 Program termination

The process manager exits when all the processes in the input file have finished execution, that is, there are no more processes in the input queue and no *READY* or *RUNNING* processes.

3 Programming Tasks

3.1 Task 1: Process Scheduling (Non-preemptive)

In this task, your program selects the next process to run among the *READY* processes based on the Shortest Job First (SJF) scheduling algorithm.

Shortest Job First (SJF): The process with the shortest *service time* should be chosen among all the *READY* processes to run. The process should be allowed to run for the entire duration of its service time without getting suspended or terminated (i.e., non-preemptive). If there is a tie when choosing the shortest process (i.e. two or more processes have the same service time), choose the process which has the earlier arrival time. If there are two or more processes that have the same service time as well as the same arrival time, select the process whose name comes first lexicographically.

In the case of **SJF**, after one quantum has elapsed:

- If the process has completed its execution, the process is terminated and moves to the *FINISHED* state.
- If the process requires more CPU time, the process continues to run for another quantum.

3.2 Task 2: Process Scheduling (Preemptive)

In this task, your program selects the next process to run among the *READY* processes based on the Round Robin (RR) scheduling algorithm.

Round Robin (RR): The process at the head of the *READY* queue is chosen to run for *one quantum*. After a process has run for one quantum, it should be suspended and placed at the **tail** of the *ready* queue **unless** there are no other processes currently in the *ready* queue.

In the case of **RR**, after one quantum has elapsed:

- If the process has completed its execution, the process is terminated and moves to the *FINISHED* state.
- If the process requires more CPU time and there are other *READY* processes, the process is suspended and enters the *READY* state again to await more CPU time.
- If the process requires more CPU time and there are no other *READY* processes, the process can continue to run for another quantum.

3.3 Task 3: Memory Allocation

In this task, you have to implement an additional memory allocation phase before your program starts the scheduling phase of the simulation. Note that memory allocation is also **simulated**, i.e. the process manager only has to track of and report simulated memory addresses.

In each process management cycle (i.e., after each quantum), your program must attempt to allocate memory to **all** the processes in the input queue, serving processes in the order of appearance (i.e. from the first process in the input queue to the last), and move successfully allocated processes to the *ready* queue in order of allocation. Processes for which memory allocation cannot be currently met should remain in the input queue.

Processes whose memory allocation has succeeded are considered *READY* to run. The input queue should not contain any *READY* processes.

When allocating memory to a particular process, you are required to implement the **Best Fit** memory allocation algorithm[1] as explained in the lectures and in the textbook.

Important Notes:

1. The memory capacity (in MB) of the simulated computer is static. For this project, you will assume a total of **2048 MB** is available to allocate to user processes.
2. The memory requirement (in MB) of each process is known in advance and is static, i.e., the amount of memory a process is allocated remains constant throughout its execution.
3. A process's memory must be allocated in its entirety (i.e., there is no paging) and in a contiguous block. The allocation remains for the duration of the process's runtime (i.e., there is no swapping). Always choose the block with the lowest sequence number. e.g. Consider a sequence number [0..2048) for all the blocks. If there are 2 equal-sized blocks, say [100, 199], and [300, 399], choose the block starting at address 100.
4. Once a process terminates, its memory must be freed and merged into any adjacent holes if they exist.

3.4 Task 4 (Challenge Task): Creating and Controlling Real Processes

If you choose to complete this task, every process scheduled by the manager should trigger the execution of a `process.c` program, supplied as part of this project.

Include the following line in your code for this task to be marked:

```
#define IMPLEMENTS_REAL_PROCESS
```

[1] Hint: Best Fit algorithm selects the smallest available contiguous block of memory that is big enough to accommodate the memory requirement of a process.

Get a copy of `process.c` from the Project 1 repository on GitLab. Compile this into an executable named `process`, which will be `exec`-ed by your code. In the marking environment, the `process` executable will be automatically placed into the same directory as your program.

Implement these additional steps in your program:

1. When a scheduling algorithm selects a process to run for the first time, the process is *created*.
2. In the case of RR, after one quantum has elapsed:
   - If the process has completed its execution, the process is *terminated*.
   - If the process requires more CPU time and there are other *READY* processes, the process is *suspended* and enters the *READY* state again to await for more CPU time. The next time that the *suspended* process is scheduled, it is *resumed*.
   - If the process requires more CPU time and there are no other *READY* processes, the process is *continued* to run for another quantum.
3. In the case of SJF, after one quantum has elapsed:
   - If the process has completed its execution, the process is *terminated*.
   - If the process requires more CPU time, it is *continued* to run for another quantum.

The command line specification for `process` is as follows:

```
Usage: process [-v|--verbose] <process-name>
```

Creating `process`

1. `fork()` and `exec` an instance of `process` from your program.
   Provide the name of the scheduled process as a command line argument.
   Optionally specify the `-v` flag to get `process` to print debug logs to standard error.
2. Send the 32 bit simulation time (See Section 5.1 `RUNNING`) of when the process starts running to the standard input of `process`[2], in Big Endian Byte Ordering (See Section A).
3. Read 1 byte from the standard output of `process`, and verify that it's the same as the least significant byte (last byte) that was sent.

Suspending `process`

1. Send the 32 bit simulation time of when the process is suspended (Similar to `FINISHED` – See Section 5.1) to the standard input of `process`, in Big Endian Byte Ordering.
2. Send a `SIGTSTP` signal to `process`, and wait for `process` to enter a stopped state. e.g.:

```
kill(child_pid, SIGTSTP);

pid_t w = waitpid(child_pid, &wstatus, WUNTRACED);   // See $ man 2 waitpid
if (WIFSTOPPED(wstatus)) {
    // Loop until this condition is met (Loop ommitted for brevity)
}
```

Resuming or Continuing `process`

1. Send the 32 bit simulation time of when the process is resumed or continued (See Section 5.1 `RUNNING`) to the standard input of `process`, in Big Endian Byte Ordering.
2. Send a `SIGCONT` signal to `process`.
3. Read 1 byte from the standard output of `process`, and verify that it's the same as the least significant byte (last byte) that was sent.

[2]Hint: You should use `pipes` and `dup2` to write to and read from `process` – See Practical 3 Section 4.

Terminating `process`

1. Send the 32 bit simulation time of when the process is finished (See Section 5.1 `FINISHED`) to the standard input of `process`, in Big Endian Byte Ordering.
2. Send a `SIGTERM` signal to `process`.
3. Read a 64 byte string from the standard output of `process`, and include it in the execution transcript (Section 5.1) for marking.

4 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 all the arguments will be passed correctly, and each argument will be passed exactly once.

```
Usage: allocate -f <filename> -s (SJF | RR) -m (infinite | best-fit) -q (1 | 2 | 3)
```

`-f` *filename* will specify a valid *relative* or *absolute* path to the input file describing the processes.

`-s` *scheduler* where *scheduler* is one of {`SJF`, `RR`}.

`-m` *memory-strategy* where *memory-strategy* is one of {`infinite`, `best-fit`}.

`-q` *quantum* where *quantum* is one of {`1`, `2`, `3`}.

The *filename* contains the list of processes to be executed, with each line containing a process. Each process is represented by a space-separated tuple (*time-arrived*, *process-name*, *service-time*, *memory-requirement*).

You can assume:

- The file will be sorted by *time-arrived* which is an integer in $[0, 2^{32})$ indicating seconds
- All *process-name*s will be distinct uppercase alphanumeric strings of minimum length 1 and maximum length 8.
- The first process will always have *time-arrived* set to 0.
- *service-time* will be an integer in $[1, 2^{32})$ indicating seconds.
- *memory-requirement* will be an integer in $[1, 2048]$ indicating MBs of memory required.
- The file is space delimited, and each line (including the last) will be terminated with an LF (ASCII `0x0a`) control character.

You can read the whole file before starting the simulation or read one line at a time.
Note that there can be inputs with large gaps in the process arrival time, like in a batch system.

We will not give malformed input (e.g., negative memory requirement or more than 4 columns in the process description file). If you want to reject malformed command line arguments or input, your program should exit with a non-zero exit code per convention.

Example: `./allocate -f processes.txt -s RR -m best-fit -q 3`.

The allocation program is required to simulate the execution of processes in the file `processes.txt` using the Round Robin scheduling algorithm and the Best Fit memory strategy with a quantum of 3 seconds.

Given `processes.txt` with the following information:

```
0 P4 30 16
29 P2 40 64
99 P1 20 32
```

The program should simulate the execution of 3 processes where process P4 arrives at time 0, needs 30 seconds of CPU time to finish, and requires 16 MB of memory; process P2 arrives at time 29, needs 40 seconds of time to complete and requires 64 MB of memory, etc.

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.

5.1 Execution transcript

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

- If completing memory management (Task 3.3), when a process is allocated memory with the best-fit memory strategy and becomes `READY`:

  `<time>,READY,process_name=<process-name>,assigned_at=<addr>\n`

  where:
  - '`time`' refers to the simulation time at which the process becomes `READY`;
  - '`process-name`' refers to the *name* of the process as specified in the process file;
  - '`addr`' is the memory address $[0, 2048)$, that the allocation for this process starts at.

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

  `<time>,RUNNING,process_name=<process-name>,remaining_time=<T>\n`

  where:
  - '`time`' refers to the simulation time at which CPU is given to a process;
  - '`process-name`' refers to the *name* of the process that is about to be run;
  - '`T`' refers to the remaining execution time for this process.

- Every time a **process** finishes:

  `<time>,FINISHED,process_name=<process-name>,proc_remaining=<num-proc-left>\n`

  where:
  - '`time`' is the simulation time at which the process transitions to the *FINISHED* state;
  - '`process-name`' refers to the *name* 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. The number of processes that are in the *input* and *ready* queues when this particular process terminates.

- And additionally **if completing Task 3.4**:

  `<time>,FINISHED-PROCESS,process_name=<process-name>,sha=<sha256>\n`

  where:
  - '`time`', '`process-name`' are as above;
  - '`sha256`' is the 64 byte hexadecimal string read from `process`.

If there is more than one event to be printed at the same time: print `READY` before `RUNNING`, both `FINISHED` and `FINISHED-PROCESS` before `READY`, and print `FINISHED` before `FINISHED-PROCESS`.

Example: Consider the above `processes.txt` with `-q 3`.
Then the following lines may be printed:

```
0,READY,process_name=P4,assigned_at=0                (With best-fit memory strategy)
0,RUNNING,process_name=P4,remaining_time=30
30,FINISHED,process_name=P4,proc_remaining=0
30,FINISHED-PROCESS,process_name=P4,sha=4a5e392...   (If completing Task 4)
30,READY,process_name=P2,assigned_at=0
30,RUNNING,process_name=P2,remaining_time=40
...
```

5.2 Task 5: 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 is completed (transitioned to `FINISHED` state) 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 service time (i.e., the one specified in the process description file).
- **Makespan:** the simulation time (in seconds) of when the simulation ended.

Example: Consider the above `processes.txt` with `-q 3`. Then the following lines may be printed:

```
Turnaround time 32
Time overhead 1.08 1.04
Makespan 120
```

6 Marking Criteria

The marks are broken down as follows:

Task # and description	Marks
1. Shortest Job First Algorithm (Section 3.1)	2
2. Round Robin Algorithm (Section 3.2)	3
3. Best Fit Memory Allocation (Section 3.3)	3
4. Controlling Real processes (Section 3.4)	3
5. Performance statistics computation (Section 5.2)	2
6. Build quality	1
7. Quality of software practices	1

Tasks 1-5. We will run your submission against our test cases. You will be given access to a subset of test cases and their expected outputs. Half of your mark for these tasks will be based on these visible test cases.

Task 1 and 2 will be evaluated with the infinite memory strategy only. Task 3 will be evaluated with both scheduling algorithms. Task 4 and 5 will be evaluated with a combination of memory strategies and scheduling algorithms.

Tasks 1, 2, 3, and 4 will be marked based on the execution transcript (Section 5.1) and Task 5 based on performance statistics (Section 5.2).

Task 6. Build quality 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.

Running `make clean && make -B && ./allocate <...arguments>` should execute the submission. Compiling using "`-Wall`" should yield no warnings. Running `make clean` should remove all object code and executables.

Do not commit `allocate` or other executable files (see Practical 1). A 0.5-mark penalty will be applied if your final commit contains any such files. Scripts (with `.sh` extension) are exempted.

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

Task 7. Quality of software practices Factors considered include **quality of code**, based on the choice of variable names, comments, formatting (e.g. consistent indentation and spacing), structure (e.g. 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/or non-informative commit messages will lose 0.5 marks).

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 and produce deterministic output.

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`.

The git history may be considered for matters such as special consideration, extensions and potential plagiarism.

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 `comp30023-2023-project-1` in the subgroup with your username of the group `comp30023-2023-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 (or approved extension) will be marked without a 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/comp30023-2023-projects/<username>/ ...
    ... comp30023-2023-project-1
cd comp30023-2023-project-1
git checkout <commit-hash-submitted-to-lms>
```

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

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

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.

8 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 strongly encouraged to write tests to verify the correctness of your own implementation.

Testing Locally: You can clone the sample test cases to test locally, from:
comp30023-2023-projects/project1.

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 with the same set of test cases.

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.

The requisite `.gitlab-ci.yml` file has been provisioned and placed in your repository, but is also available from the `project1` repository linked above.

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

9 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 the subject's discussion board Ed. 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 will be used to compare submissions. It is the University's 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.

A Endianness

A single (unsigned) byte can represent values up to 255 by itself. To represent larger values, we must compose multiple bytes together. Endianness is concerned with the order in which these bytes are represented in memory or are transmitted over a communication channel, and varies across processor architectures and protocols.

There are 2 orderings. Big Endian stores the most significant byte at the lowest address and the least significant byte at the highest address. Little Endian is the opposite, storing the most significant byte at the highest address and the least significant byte at the lowest address.

For example, if we were to store the decimal number `640` (which is $2^9 + 2^7$, or `0x0280`) as a 16-bit integer in memory (e.g. `*(uint16_t*) arr = 640;`), and print the (unsigned) values of the 2 bytes of memory which contains the number (e.g. `printf(..., arr[0], arr[1]);`), we'd expect `[0x02, 0x80]` for big endian, and `[0x80, 0x02]` for little endian.