Process and Memory Management

1 Background

You will **simulate** a process manager in a system where all processes are fully CPU-bound (i.e., have a single CPU burst and do no I/O). The process manager i) allocates processes to a CPU in a round-robin manner and ii) supports contiguous, paged, and virtual memory management.

2 Process Manager Overview

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 (T_S) starts at 0 and increases by the length of the quantum (Q) every cycle. 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 all processes that have been submitted to the system since the last cycle occurred and add them to the *process 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.
2. Identify whether the process (if any) that is currently running (i.e., was given CPU time in the previous cycle) has completed its execution. If it has:
   - The process's state is updated (see Section 3)
   - The process is removed from the process queue
   - The process's memory is deallocated
3. Determine the process that runs in this cycle. This decision is made based on the scheduling algorithm (round robin) and the memory allocation strategy. This step entails:
   - Updating the state of the process that is currently running (if any) and the state of the newly allocated process (see Section 3)
   - Updating the process queue if needed

   A detailed explanation of this stage is given for each task.
4. Increment the simulation time by Q seconds.

This cycle is repeated iteratively until all the processes that were submitted to the system have completed their execution.

3 Process Lifecycle

The lifecycle of a process is as follows:

1. A process is submitted to the process manager via an input file (See Section 6 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 process queue based on their arrival time and the current simulation time.
2. A process is in a *READY* state after it has arrived (arrival time less than or equal to the simulation time). *READY* processes are considered by the scheduling algorithm as candidates to be allocated to the CPU.
3. The process that has been selected to use the CPU enters a *RUNNING* state.
4. After running for one quantum,
   - 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 transitions back to the *READY* state to await more CPU time.
   - If the process requires more CPU time and there are no other *READY* processes, the process remains in the RUNNING state and runs for another quantum.

   For simplicity, a process can only transition to the *FINISHED* state at the end of a quantum. This means that, in cases in which the *service time* of a process is not a multiple of the quantum, the total amount of time the process spends in the *RUNNING* state will be greater than its *service time*.

4 Process Scheduling

In this section, you will focus on implementing the scheduling logic of the process manager. For this purpose, you will assume infinite memory that requires no management.

4.1 Task 1: Round-Robin Scheduling with Infinite Memory

In this task, you will implement a round-robin scheduler under the assumption that the memory requirements of processes are immediately satisfied upon arrival. This will allow you to focus on implementing the scheduling logic before moving on to implementing memory management approaches in subsequent tasks.

In round-robin scheduling, processes execute on the CPU one quantum at a time. The scheduler allocates the CPU to the process at the head of the process queue (i.e., the process enters the *RUNNING* state). After one quantum has elapsed, the process returns to the *READY* state, moves to the tail of the process queue, and the CPU is allocated to the next process in the queue (i.e., the new head of the queue).

There are two special cases in which a process does not transition from *RUNNING* to *READY* at the end of a quantum (as defined in Section 3):

1. There are no other processes in the queue, and the process requires more CPU time. The process remains in a *RUNNING* state and continues to use the CPU for another quantum.
2. The process completed its execution. The process transitions to the *FINISHED* state *and* is removed from the process queue.

Note that, based on the order in which the process manager performs tasks (Section 2), a process that has exhausted its quantum is placed at the tail of the process queue *after* newly arrived processes have been inserted into said queue.

5 Memory Management

For the tasks in this section, you will assume memory is finite. Memory must be allocated to a process before said process is able to run on the CPU. Consequently, a process's memory must be deallocated upon completion of said process.

To accomplish this, you will extend the round-robin scheduler implemented in Task 1 to consider the memory requirements of a process before it is able to enter the *RUNNING* state. When it is a process' turn to execute (as determined by the round-robin algorithm), the process manager must first allocate memory to the process by following one of the following strategies:

- Allocating a contiguous block of memory (Task 2)
- Allocating all the pages of the process to frames in memory (Task 3)
- Allocating a subset of the pages of the process to frames in memory (Task 4)

Only if, and after, memory allocation is successful is a process allowed to use the CPU for the corresponding quantum.

5.1 Task 2: Round-Robin Scheduling with Contiguous Memory Allocation

In this task, the process manager allocates a process's memory in its entirety (i.e., there is no paging) and in a contiguous block. Memory must be allocated following the **First Fit** memory allocation algorithm[1]. The memory remains allocated for the duration of the process's runtime (i.e., there is no swapping).

A process for which memory allocation cannot be currently met should remain in a *READY* state, and be moved from the head to the tail of the process queue. Within the same cycle, the scheduler must continue to iterate over the process queue until it finds a process that can execute (i.e., memory has been allocated). Note that it is only after a process has successfully transitioned from *READY* to *RUNNING* *or* when the process queue is empty that the process manager moves on to the next cycle, and hence, the next quantum.

Important Notes:

- The memory capacity of the simulated computer is static. For this project, you will assume a total of **2048 KB** is available to allocate to user processes.
- The memory requirement (in KB) 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.
- For simplicity, you will assume memory is addressed in blocks of 1 KB. Memory addresses in the system are therefore in the range [0..2048).
- When allocating a memory block, always allocate the block starting at the lowest memory address of a memory hole. For example, a block of 10 KB needs to be allocated. The identified memory hole (according to first-fit) is [10..30]. The memory block should then be allocated to addresses [10..19].
- Once a process terminates, its memory must be freed and merged into any adjacent holes if they exist.

A sample execution flow, as specified by this task, would be as follows:

1. The round-robin scheduler determines process p is the next process to be allocated to the CPU.
2. Before allocating the process to the CPU, the process manager checks whether p has been allocated memory.

[1]Hint: The First Fit algorithm selects the first available contiguous block of memory that is large enough to accommodate the memory requirement of a process.

(a) If p's memory has already been allocated, p gets to use the CPU for the corresponding quantum.

(b) If p's memory has not been allocated, the process manager attempts to allocate a contiguous block.

i. If successful, p gets to use the CPU for the corresponding quantum.

ii. If the allocation is unsuccessful (i.e., there is no sufficient memory in the system at this time), p does not execute, remains in a *READY* state, and is moved to the tail of the process queue. The scheduler looks for another process to execute by returning to step 1.

5.2 Task 3: Round-Robin Scheduling with Paged Memory Allocation

This task assumes a paged memory system with swapping. The memory required by a process is divided into pages, and physical memory is divided into frames. Pages that are mapped to frames in memory are considered to be allocated.

Before a process runs on the CPU, **all of its pages** must be allocated to frames in memory. If there are not enough empty frames to fit a process's pages, then pages of *another* process or processes need to be swapped to disk to make space for the process. When choosing a process to swap, you must choose the process that was **least recently executed** among other processes (excluding the current one) and evict **all of its pages**. If there is still not enough space, continue evicting all pages of other processes following the least-recently executed policy until there is sufficient space.

Important Notes:

- You will assume a total of **2048 KB** is available to allocate to user processes.
- The memory requirement of each process (in KB) is known in advance and is static, i.e., the amount of memory a process requires, and hence the number of pages, remains constant throughout its execution.
- Once a process terminates, all of its pages must be evicted from memory (i.e., deallocated).
- The size of pages and frames is **4 KB**.
- Each frame is numbered, starting from 0 and increasing by 1. For the assumed memory size of 2048 KB, there are 512 pages in total, with page numbers from 0 to 511.
- Pages should be allocated to frames in increasing frame number. For example, if a process requires 3 pages to be allocated, and frames 0, 1, 5, 8, and 9, are free (or were freed via swapping). The process pages must be mapped to frames 0, 1, and 5.

A sample execution flow, as specified by this task, would be as follows:

1. The round-robin scheduler determines process p is the next process to be allocated to the CPU.
2. Before allocating the process to the CPU, the process manager checks whether p's pages are allocated in memory.
   (a) If p's pages are allocated, p uses the CPU for the corresponding quantum.
   (b) If p's pages have not been allocated and there are not enough free frames in memory, the process manager evicts the pages of one or more processes following the least-recently executed policy.
   (c) Once there are sufficient free frames in memory, the process manager allocates p's pages and p runs on the CPU for the corresponding quantum.

5.3 Task 4: Round-Robin Scheduling with Virtual Memory Allocation

This task will assume a paged system with swapping similar to that in Task 3. However, we will now consider the case of virtual memory providing the illusion of a larger-than-available memory to processes.

You will now assume that a process does not need all pages to be allocated before it is allowed to execute. In this task, a process can be executed if **at least** 4 of its pages are allocated (or all pages in case of processes requiring less than 4 pages). If there are more than 4 frames available at the time of allocation(or reallocation), the process manager must allocate as many pages as possible. For example, if a process requires 7 pages and there are 6 frames available, the process manager must allocate 6 of the 7 pages to the available frames. If a process requires 7 pages and there are 10 frames available, the process manager must allocate all 7 pages to the free frames.

Similar to swapping, if there are not enough empty frames for the process that is scheduled to be executed, pages of the least recently executed process need to be evicted *one at a time* until there are 4 empty pages (or less if the process requires less than 4 pages). The lowest numbered frames belonging to the least recently executed process must be evicted first. For example, if the least recently executed process was allocated frames `1,5,7,9`, and 2 frames need to be evicted, frames `1,5` must be evicted. This is in contrast to Task 3, where *all* pages of the least recently executed process would be evicted.

Important Notes:

- You will assume a total of **2048 KB** is available to allocate to user processes.
- Once a process terminates, any allocated pages must be evicted from memory (i.e., deallocated).
- The size of pages and frames is **4 KB**.
- Each frame is numbered, starting from 0 and increasing by 1. For the assumed memory size of 2048 KB, there are 512 pages in total, with page numbers from 0 to 511.
- Pages should be allocated to frames in increasing frame number. For example, if a process requires 3 pages to be allocated, and frames 0, 1, 5, 8, and 9, are free (or were freed via swapping). The process pages must be mapped to frames 0, 1, and 5.

A sample execution flow, as specified by this task, would be as follows:

1. The round-robin scheduler determines process p, requiring n pages, is the next process to be allocated to the CPU.
2. Before allocating the process to the CPU, the process manager checks whether p has at least 4 ($n \geq 4$) or all ($n < 4$) pages allocated.
   (a) If p's page allocation requirements are met, p uses the CPU for the corresponding quantum.
   (b) If p's page allocation requirements are not met and there are not enough free frames in memory, the process manager evicts just enough pages to meet the page allocation requirements of p following the least-recently executed policy.
   (c) Once there are sufficient free frames in memory, the process manager allocates p's pages and p runs on the CPU for the corresponding quantum.

6 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> -m (infinite | first-fit | paged | virtual) -q (1 | 2 | 3)
```

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

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

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

The input file, *filename*, contains the list of processes to be executed, with each line containing a process. Each process is represented by a single 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 KBs of memory required.
- The file is space delimited, and each line (including the last) will be terminated with an LF (ASCII `0x0a`) control character.
- Simulation time will be an integer in $[0, 2^{32})$ indicating seconds.

Note that no assumptions may be made about the number of processes in the input file and that there can be input files with large gaps in the process arrival time. You can, however, assume that the input files used to test your program are such that simulations will complete in a reasonable amount of time.

In addition, no assumptions may be made about the length of the file name (*filename*).

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 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 -m infinite -q 3`.

The allocate program is required to simulate the execution of processes in the file `processes.txt` using the round-robin scheduling algorithm and the infinite 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 KB of memory; process P2 arrives at time 29, needs 40 seconds of time to complete and requires 64 KB of memory, etc.

7 Expected Output

In order for to verify that the 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.

7.1 Execution transcript

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

- When a process runs on the CPU (this includes the first time and every time it resumes its execution):

  `<time>,RUNNING,process-name=<pname>,remaining-time=<rtime>,`
  `mem-usage=<musage>%,allocated-at=<addr>,mem-frames=[<frames>]`

  where:

  - '`time`' refers to the simulation time at which CPU is given to the process;
  - '`pname`' refers to the *name* of the process as specified in the process file;
  - '`rtime`' refers to the remaining execution time for this process;
  - '`musage`' is a (rounded up) integer referring to the percentage of memory currently occupied by all processes, after `pname` has been allocated memory;
  - '`addr`' is the memory address (between $[0, 2048)$) at which the memory allocation for `pname` starts at;
  - '`frames`' is a list of frame numbers (given in increasing order) that are allocated to the current process, separated by commas.

  In the case of infinite memory (Task 1, `-m infinite`), your program should not print out any information about memory allocation or usage. That is, `mem-usage`, `allocated-at`, and `mem-frames` should not be printed.

  An example of the simplified output would be:

  ```
  20,RUNNING,process-name=P4,remaining-time=10
  ```

  In the case of first-fit (Task 2, `-m first-fit`), your program should not print out the set of allocated frames. That is, `mem-frames` should not be printed.

  An example of the simplified output would be:

  ```
  20,RUNNING,process-name=P4,remaining-time=10,mem-usage=50%,allocated-at=10
  ```

  In the case of paged (Task 3, `-m paged`) and virtual memory(Task 4, `-m virtual`), your program should not print out the memory allocation address. That is, `allocated-at` should not be printed.

  An example of the simplified output would be:

  ```
  20,RUNNING,process-name=P4,remaining-time=10,mem-usage=50%,mem-frames=[0,1,2]
  ```

- In the case of paged (Task 3, `-m paged`) and virtual memory (Task 4, `-m virtual`), every time pages are deallocated from memory:

  ```
  <time>,EVICTED,evicted-frames=<[frames]>
  ```

  where:

  - '`time`' is as above for the `RUNNING` event;
  - '`frames`' refers to the list of frame numbers (given in increasing order), separated by commas, that were freed.

  In cases in which pages of more than one process are evicted, Only *one* `EVICTED` event should be printed. This means your program should never print two `EVICTED` events in two consecutive lines.

  In the cases of infinite memory (Task 1, `-m infinite`) and first-fit (Task 2, `-m first-fit`), no `EVICTED` events should be printed.

- Every time a **process** finishes:

  `<time>,FINISHED,process-name=<pname>,proc-remaining=<pleft>`

where:

- '`time`' is the simulation time at which the process transitions to the *FINISHED* state;
- '`pname`' refers to the *name* of the process as specified in the process file;
- '`pleft`' refers to the number of **processes** that are waiting to be executed.
  i.e. The number of processes that are in the process queue when this particular process terminates.

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

7.2 Task 5: Performance Statistics

When the simulation completes, three lines with the following performance statistics about your simulation performance should be printed:

- **Turnaround time**: average turnaround time (in seconds, rounded up to an integer) for all processes in the simulation. Recall the turnaround time is the time elapsed between the arrival and the completion of a process.
- **Time overhead**: maximum and average time overhead, both rounded to the first two decimal points. The time overhead of a process is defined as its turnaround time divided by its service time.
- **Makespan**: The length of the simulation. That is, simulation time when all processes in the input completed their execution.

Example:

```
Turnaround time 31
Time overhead 1.03 1.02
Makespan 119
```

Build quality

- The repository must contain a `Makefile` that 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.
- 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.
- The test script expects `allocate` to exit with status code 0 (i.e. it successfully runs and terminates).