FIT2100 Assignment Part B: Process Scheduling Simulation Semester 2 2021

Dr Tawfiq Islam
Lecturer, Faculty of IT
Email: tawfiq.islam@monash.edu

September 19, 2021

Revision Status

version 3: September 2021 by Tawfiq Islam

1 Introduction

Simulating scheduling algorithms is an excellent way to test and compare different algorithms in various scenarios. It helps in choosing an appropriate algorithm both for an OS or large scale systems. On such systems, the access patterns of resources (e.g, CPU, memory) by users/processes, and the necessary system metrics (e.g., process turnaround time, CPU/memory usage) can be recorded for later analysis. Also, data from a real system can be used as an input to simulate the processes/applications from a real system, and to test how the system would perform with different scheduling algorithms. **In this assignment, you will create three different scheduler programs that each implement a different process scheduling algorithm to simulate the scheduling of processes in a system**.

You will not be doing scheduling of real processes, however your programs will determine the sequence in which a certain number of 'imaginary' processes are to be executed.

This document constitutes the requirement specification for this assignment. You will be assessed on your ability to both comprehend and comply with the requirements as specified herein.

Due: 8th October 2021 (Friday) 5PM AEDT

Late submissions: A late submission penalty of 10% of the total assignment marks per day will apply. No submissions will be accepted after 22nd October 2021 (end of semester 2) except in exceptional circumstances.

This assignment is worth 15% of the total marks for this unit.

2 If you require extra help

If you are stuck and/or not knowing where to begin, refer to the helpful hints in section 6.

This assignment is an independent learning and assessment exercise.

You may utilise the **Ed Discussion Forum** to ask questions and obtain clarification, however you may not share details or code in your implementation with other students, nor may you show your code to the teaching team prior to submission. This is an assessment task: tutors and lecturers should not be helping you debug your assignment code *directly* (you are expected

to debug and test your own code), but can help with more general queries, such as queries related to C programming syntax, concepts and debugging tips.

You may make use of online references with appropriate citation in accordance with academic integrity policies. However, your work must be your own.

3 About the 'processes'

In this assignment, we will use a simplified model of a process, where each process:

1. has a pre-defined total service time, and
2. does not use I/O and can never be in a blocked state, and
3. will eventually run to completion and *does not* encounter any errors (e.g. segmentation faults).

Each process should be represented within your program as a process control block (PCB) instance defined as follows:

```c
/* Special enumerated data type for process state */
typedef enum {
    READY, RUNNING, EXIT
} process_state_t;

/* C data structure used as process control block. The scheduler
 * should create one instance per running process in the system.
 */
typedef struct {
    char process_name[11]; // A string that identifies the process

    /* Times are measured in seconds. */
    int entryTime;    // The time process entered system
    int serviceTime; // The total CPU time required by the process
    int remainingTime; // Remaining service time until completion.

    process_state_t state; //current process state (e.g. READY).
} pcb_t;
```

As a PCB is essentially a container containing information about a process, using a `struct` keeps all the information about a process neatly encapsulated and makes it easier to manage from a coding perspective[1].

[1] A real operating system needs to manage far more information about a process, and we are only looking at a simplified model for this assignment. For example, the process control block structure in Linux is called task_struct, and if you are curious, you can see its very complicated definition here: `https://elixir.bootlin.com/linux/latest/source/include/linux/sched.h`

3.1 Important notes

PCBs should be used to hold the information about the current processes of the system only. You **MUST NOT**:

- store the information of a process which has not yet arrived (i.e. has an arrival time in the future), or
- has been terminated.

You **may**:

- modify the above `struct` definition to include additional process information, and
- also include additional states in the `process_state_t` enumerated type if it is useful for your implementation of the tasks in Section 5.

3.2 How to 'run' a process

Our processes have state information, but no program code to be executed! Times are measured in seconds and represented as integers. Your scheduler should progress through each second during the simulation and update the process states accordingly. The scheduling program should instantly `move` and start working on the next second without any delay/wait after it finishes updating the process states for the current second.

4 About the scheduler programs you will write

4.1 Receiving process data

Each of these programs should take a filepath as its **only** argument, and should read process data from this file[2]. If no argument is specified, use the default file with the name `processes.txt` in the same directory as your programs instead.

[2]Hint: Unlike the previous assignment which was focused on low level system calls, you are **ALLOWED** (if you wish) to use the high-level functions found in `<stdio.h>` such as `fopen`, `fclose` and `fscanf` for reading the file contents. Note that these high-level functions do not work with the low-level `open`, `close`, etc. functions you have used in the previous assignment.

4.2 Format of process data

Each line in the process data file (`processes.txt`) should contain *space-separated* values for a single process as follows:

[Process Name] [Arrival Time] [Service Time] [Deadline]

For example, the lines of the process data file `processes.txt` can be as follows:

P1	0	3	5
P2	1	6	7
P3	4	4	6
P4	6	2	2

Based on the above, the process named **P1**:

- *arrives* (see Section 4.2.2) into the system at time: **0 seconds**, and
- has a total required *service time* (see Section 4.2.2) of **3 seconds**, and
- has a *deadline* (see Section 4.2.3) of **5 seconds**.

Each process in the system should be stored in its own process control block using the `pcb_t` data structure as defined in Section 3, which should be kept updated throughout the lifetime of the process.

For all tasks in Section 5, you **can** assume that the `processes.txt` file will have:

- processes listed in order of arrival time, and
- at most 100 processes.

4.2.1 Process names

For all tasks in Section 5, you **can** assume that the name of each process:

- is never more than 10 characters in length, and
- does not contain spaces.

4.2.2 Arrival and service times

Arrival and service times are measured in **seconds**, and represented as **integers**.

For all tasks in Section 5, you **MUST NOT** assume that:

- there will be no processes with the same arrival time.

4.2.3 Deadlines

The deadline for a process is the expected maximum time (in **seconds**) from the arrival to completion of that process. When a process finishes execution, it may either successfully meet or fail to meet its deadline (see Section 4.3.3).

For all tasks in Section 5, you **can** assume that:

- the given deadline for a process will be always greater or equal to the service time of that process.

4.3 Outputting simulation results

4.3.1 During the simulation

During the scheduling simulation conducted for each of the tasks in Section 5, each of your scheduler programs should progress through time starting from 0.

Additionally, each of your scheduler programs should print to **standard output** a message, along with the current integer time, as soon as these scheduling events happen for any process:

- a process enters the system (i.e. arrives), or
- a process enters the running state, or
- a process finishes execution.

Some example messages are shown below:

```
Time 0:  P1 has entered the system.
Time 0:  P1 is in the running state.
Time 1:  P2 has entered the system.
Time 3:  P1 has finished execution.
Time 3:  P2 is in the running state.
......
```

4.3.2 After the simulation finishes

After the scheduling simulation has finished, the scheduler program for each task in Section 5 should write their respective results to a results output file named `results-tasknum.txt`, where `tasknum` is the task number (i.e. 1, 2, or 3).

Each line in the results output file (`results-tasknum.txt`) should contain *space-separated* values for a single process as follows:

[Process Name] [Wait Time] [Turnaround Time] [Deadline Met]

For example, the lines of the results output file `results-tasknum.txt` can be as follows:

P1	0	3	1
P2	2	8	0
P3	5	9	0
P4	7	9	0

Based on the above, the process named **P2** has:

- a *wait time* of 2 seconds, and
- a *turnaround time* of 8 seconds, and
- **failed** to finish execution within the given deadline.

4.3.3 The 'Deadline Met' value

The `Deadline Met` value indicates whether the corresponding process was able to finish its execution within the given deadline (see Section 4.2.3). For example:

- a process with a service time of 2 seconds, a deadline of 5 seconds, and waited for a total of 3 seconds will have **met** its deadline, and
- a process with a service time of 5 seconds, a deadline of 10 seconds, and waited for a total of 6 seconds will have **failed to meet** its deadline.

Whether a process will meet its deadline or not, is dependent on its turnaround time:

- if its turnaround time is `lesser or equal to` its given deadline, then that process has successfully met its deadline, and
- if its turnaround time is `greater than` its given deadline, then that process did not meet its deadline.

When interpreting the `Deadline Met` value of a process:

- a value of 1 means its deadline **was met**, and
- a value of 0 means its deadline **was not met**.

5 Programming tasks

Each of the assignment tasks in Section 5 **must** be implemented as a **separate program**.

Your main C source file for a particular task should be named with your student ID. For example, your program for Task 1 should be named as `task1-123456789.c`, where `123456789` is your student ID.[3]

Your user documentation for all the tasks **must** be included in a **single** `.txt` file.

[3] If any of your completed programs contain additional source files, you may name other source and header files as you wish.

5.1 Task 1: Non-preemptive scheduling

Write a scheduling program that implements **one** of these non-preemptive scheduling algorithms:

- **FCFS** (First Come, First Served)
- **SPN** (Shortest Process Next)

For this task, a process should *still continue to run to completion* even if it does not manage to meet its deadline.

In your user documentation for this task (Task 1), you **must**:

- document the usage of your program in a plain text file and include this with your submission, and
- document any assumptions you have made about your interpretation of the scheduling algorithm's implementation.

5.2 Task 2: Preemptive scheduling

Now, implement a second scheduling program according to the following preemptive algorithm:

- **SRTN** (Shortest Remaining Time Next) scheduling algorithm, with a **time quantum of 3**.

For this task, a process should *still continue to run to completion* even if it does not manage to meet its deadline.

In your user documentation for this task (Task 2), you **must**:

- document the usage of your program in a plain text file and include this with your submission, and
- document any assumptions you have made about your interpretation of the scheduling algorithm's implementation.

5.3 Task 3: Deadline-based scheduling

Now, implement a third scheduling program, which has the primary objective of **maximising the number of processes that meet their specified deadlines**.

For this particular task, you should come up with your own algorithm, or implement any existing deadline-based scheduling algorithm you can find. Your algorithm can be either preemptive or non-preemptive.

You have complete independence to innovate/design/find the algorithm you will use for this task. However, your algorithm **must utilise the deadline values in the decision-making process**.

Any reasonable approach which has the ability to maximise the number of deadlines met for one or more scenarios will be acceptable.

In your user documentation for this task (Task 3), you **must**:

- document any assumptions you have made for your chosen (or invented) algorithm, and
- discuss **how and why** your algorithm works, and
- provide an example scenario/use-case/test-case where your chosen (or invented) algorithm will be able to maximise the number of processes that meet their specified deadlines.

5.4 Important: commenting is required

Commenting your code is essential as part of the assessment criteria (refer to Section 5.5). All program code should include three types of comments:

- file header comments at the beginning of your program file, which specify your name, your Student ID, the start date and the last modified date of the program, as well as with a high-level description of the program, and
- function header comments at the beginning of each function which describe the function, arguments and interpretation of return value, and
- in-line comments within the program which clearly and concisely explains your code.

5.5 Marking criteria

Task 1 is worth 40%, and both Task 2 and 3 are worth 30% each. The same marking criteria will be applied to all the tasks:

- 50% for working functionality according to specification.
- 20% for code architecture (algorithms, use of functions for clarity, appropriate use of libraries, correct use of pointers, etc. in your implementations of the three tasks.)
- 10% for professionalism (compliance with the assignment specifications and good coding style). Good coding style includes clarity in variable names, function names, blocks of code clearly indented, etc.
- 20% for documentation (user documentation describes functionality for the relevant task, documents how to use the programs, documents critical assumptions, provides explanations where required, etc., whereas source code is well-commented with file header comments, function header comments, and inline comments, etc.)

6 Helpful hints

6.1 If you aren't sure where to begin...

- Try breaking the problem down until you find a starting point that you are comfortable with.
- Once you have a small part working, you can extend the functionality later. **You can still pass the assignment without all of the functionality completed.**
- If you are having difficulty reading the process values from a file, *try hard-coding them at first*, in order to get other parts of your program working first. (Later, adapt your code to read the values from file instead.)
- Similarly, if you are having difficulty writing the program output values to a file, *try printing them at first*, in order to get other parts of your program working. (Later, adapt your code to write the values to the output file instead.)
- Make and keep frequent backups of your code! You may want to use a version control system such as Git to ensure that you minimise the loss of your work, should you encounter any unexpected mishaps such as your virtual machine suddenly breaking due to OS updates, hardware issues, power outages, etc.

6.2 DOs and DON'Ts

Do...	**Don't... (!)**
+ break your program logic into multiple functions to make things easier to handle	– stuff everything into a single `main` function.
+ follow a clear convention for variable names	– use vague names like `array` or `p`.
+ copy the specified `pcb_t` data type definition into your program	– hard-code lots of separate variables.
+ use a sensible data structure for holding up to 100 process control blocks	– hard-code separate variables for each process.
+ comment your code as you write it	– leave commenting until the last step.
+ check over this specification carefully (e.g. using a pen or highlighter)	– skip over specified instructions.
+ **follow the format of the specified file naming convention in your submission**	– disregard specified instructions for an arbitrary reason.

7 Submission

There will be NO hard copy submission required for this assignment. You are required to submit all your deliverables (see Section 7.1) as **individual files**. Do ensure that your submission complies with the requirements set out in this specifications document.

Your submission will be done via the assignment submission link on the FIT2100 Moodle site, and should be submitted by the deadline specified in Section 1, i.e. **8th October 2021 (Friday) 5:00pm AEDT**.

Note: You must ensure you complete the entire Moodle submission process (do not simply leave your assignment in draft status) to signify your acceptance of academic integrity requirements.

Additionally, your program must be able to run in the Linux Virtual Machine environment which has been provided for this unit. Any implementation that does not run at all in this environment will receive no marks.

7.1 Deliverables

Your submission **should include** the following files:

- **all** C source files (`.c` format) required to compile and run your programs, where **each task in Section 5 is implemented as a separate program**, and **where the main C source file for a particular task is appropriately named**, and
- a **single** user documentation file (`.txt` format) of not more than 300 lines which provides clear and complete instructions on how to compile your programs, and and how to run all of the requested features.

Your submission **may optionally include** the following files, if you require them for your implementation:

- C header files (`.h` format)
- A makefile

Do not submit compiled executable programs. Marks will be deducted for any of these requirements that are not strictly complied with.

7.2 Academic Integrity: Plagiarism and Collusion

Plagiarism Plagiarism means to take and use another person's ideas and or manner of expressing them and to pass them off as your own by failing to give appropriate acknowledgement. This includes materials sourced from the Internet, staff, other students, and from published and unpublished works.

Collusion Collusion means unauthorised collaboration on assessable work (written, oral, or practical) with other people. This occurs when you present group work as your own or as the work of another person. Collusion may be with another Monash student or with people or students external to the University. This applies to work assessed by Monash or another university.

It is your responsibility to make yourself familiar with the University's policies and procedures in the event of suspected breaches of academic integrity. (Note: Students will be asked to attend an interview should such a situation is detected.)

The University's policies are available at: `http://www.monash.edu/students/academic/policies/academic-integrity`