FIT2100 Operating Systems
Assignment #2
Process Scheduling Simulation

Muhammed Tawfiqul Islam
Lecturer, Faculty of IT.
Email: tawfiq.islam@monash.edu
©2020, Monash University

October 2, 2020

© 2020, Faculty of IT, Monash University

1 Introduction

Aim

In this assignment, you will create programs to simulate three different scheduling algorithms.

You will not be doing scheduling of real processes, however your program 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: 26th October 2020 (Monday) 11am AEDT

Late submissions: A late submission penalty of 5% of assignment total per day will apply. No submissions will be accepted after 6th November 2020 (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 knowing where to begin, refer to the helpful hints in section 5.

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.

© 2020, Faculty of IT, Monash University

3 About the 'processes'

We will use a simplified model of a process. Each process has a pre-defined total service time. Our processes do not use I/O and can never be in a blocked state.

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

```
/* 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 measured in seconds... */
    int entryTime;    //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].

You may modify this `struct` definition to include additional process information. You may also include additional states in the `process_state_t` enumerated type if it is useful for your program's implementation.

[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`

© 2020, Faculty of IT, Monash University

3.1 To ‘run’ a process

Our processes have state information, but no program code to be executed! Therefore, to ‘run’ a process for one second, your scheduling program should simply `sleep` for one second (or the amount of time until the state of the system changes).

(Although we don’t have real processes to run, it makes sense that the scheduler would be sleeping while the scheduled process is ‘running.’)

4 Programming Tasks

Each task for this assignment **must** be implemented as a separate program. All of these programs should take a filename as its **only** argument. If no argument is specified, use the default filename `process-data.txt`.

4.1 Task 1: non-preemptive scheduling

Choose *one* of these scheduling algorithms to implement.

- FCFS (First come, first served)
- SPN (Shortest process next)

Write a scheduler for your chosen algorithm. Your scheduler should read the process data from the file.[2] It should then begin ‘running’ the processes according to your chosen algorithm. Each process in the system should be stored in its own process control block using the `pcb_t` data structure as defined above, which should be kept updated throughout the lifetime of the process. While a process is ‘running’, your scheduler should sleep for the appropriate length of time using the `sleep` function.

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

© 2020, Faculty of IT, Monash University

Each line in the process data file (`process-data.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 file process-data.txt can be as follows:

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

Note: you can assume that the **name** of each process is never more than 10 characters in length and does not contain spaces.

The 'deadline' is an expected maximum time from the arrival to completion of a process. For this task, a process should still continue to run to completion whether it meets its deadline or not, however this will form part of the results at the end of the simulation.

Times are measured in **seconds** and represented as **integers**. In the above file, the process named P1 enters the system at time: 0 seconds, and has a total required service time of 3 seconds, and has a deadline of 5 seconds and so on. You *can* assume that the given deadline for a process will be always greater or equal to the service time of that process. You *cannot* assume that the entries will be listed in order of the arrival time. You *can* assume that the maximum number of processes in the system is limited to 10.

During the simulation, your scheduler should progress through time starting from 0. It should print a message whenever a process enters the system, or enters the running state, or finishes execution, along with the current integer time, as the event happens. 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.
......
```

© 2020, Faculty of IT, Monash University

After your program has finished simulating the execution of all the processes, the results of the simulation should be written to a file named `scheduler-result.txt`. Each line of this file 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 file `scheduler-result.txt` can be as follows:

P1	0	3	1
P2	2	8	0
P3	5	9	0
P4	7	13	0

In the output file, the `Deadline Met` value indicates whether the corresponding process was able to finish execution within the given deadline. If a process's turnaround time is `lesser or equal` to it's given deadline, then that process has successfully met it's deadline. A value of 1 means the deadline was met, where a value of 0 means the deadline was not met.

Thus, in the example output file, process P2 has:

- 2 seconds of wait time
- 8 seconds of turn around time
- failed to finish execution within the given deadline

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

4.2 Task 2: Preemptive Round Robin scheduling

Now implement a second scheduling program according to the **same requirements** as task 1, except:

- Implement the RR (Round Robin) scheduling algorithm on the given process data, with a time-slice quantum of 2 seconds.

4.3 Task 3: Deadline-based scheduling

Now implement a third scheduling program, which has an objective to **maximise** 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. Note that, your algorithm can be either preemptive or non-preemptive. You have complete independence to design/find the algorithm you will use for this task.

In your user documentation for Task 3, write about your assumptions for your chosen (or invented) algorithm. Also, discuss the approach used by your algorithm to try to outperform the other algorithms you have implemented in Task 1 and Task 2 in terms of maximising the number of processes that meet their specified deadlines.

4.4 Important: format of the implemented programs

All your Task 1, Task 2, and Task 3 implementations should be **separate programs**. Your main C source file for a particular task should be named with your student ID. For example, for task-1, your program should be named as: `task-1-123456789.c`, where 123456789 is your student ID.[3]

User documentation for all the tasks must be included in a single file. If you need to make an assumption about how to interpret the scheduling in a certain circumstance, you should mention any such assumptions in your documentation.

4.5 Important: commenting is required

Commenting your code is essential as part of the assessment criteria (refer to Section 4.6). All program code should include three types of comments: (a) 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. (b) Function header comments at the beginning of each function should describe the function, arguments and interpretation of return value. (c) In-line comments within the program are also part of the required documentation.

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

© 2020, Faculty of IT, Monash University

4.6 Marking criteria

Tasks 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, critical assumptions are documented, code is well-commented with file-header-, function-header-, and inline- comments.)

5 Helpful hints

5.1 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.)

© 2020, Faculty of IT, Monash University

5.2 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 10 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.

6 Submission

You required to submit the assignment via the submission link on the FIT2100 Moodle site by the due date specified in Section 1, i.e. **26th October 2020 (Monday) by 11:00am AEDT**.

You are required to archive and compress all your deliverables into a single `.tar.gz` or `.tar.xz` file named with your Student ID, for submission. For example, if your Student ID is 12345678, you would submit a zipped file named `12345678_A2.tar.gz`.

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.

6.1 Deliverables

Your submission should be archived and compressed into a single `.tar.gz` or `.tar.xz` file containing the following documents:

- Electronic copies of ALL your files (e.g. C source file(s)) that are needed to compile and run your program. (Note that your program must run in the Linux Virtual Machine

© 2020, Faculty of IT, Monash University

environment which has been provided for this unit. Any implementation that does not run at all in this environment will receive no marks.)

- A user documentation file (not more than 300 lines) in plain `.txt` format with clear and complete instructions on how to compile and run your program.

Marks will deducted for any of these requirements that are not strictly complied with.

6.2 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`

© 2020, Faculty of IT, Monash University