Coursework COMP2046 Autumn 2021

Weight: 25% module marks

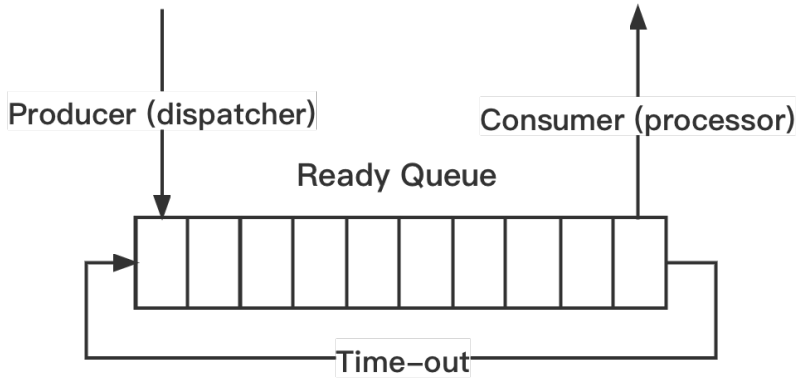
Deadline: 17th December 2021, 5pm Beiging time

Submission:

Create a single .zip file containing your source code files. We will need to rebuild your code to test your implementation. You should submit your single zip file through Moodle.

Overview

The goal of this coursework is to make use of operating system APIs (specifically, the POSIX API in Linux) and simple concurrency directives to model a process scheduling system that is similar to ones that you may find in fairly simple operating systems. More specifically, we would like you implement the process scheduling system in the context of producer-consumer problem as shown in the figure.



A ready queue contains processes that have been admitted to the system, are in the READY state, and hence ready to run on the procecessor. A process running (i.e. it is in the RUNNING state) on the processor is simulated by calling one of the `simulateProcess()` functions. A process scheduling algorithm is used to determine which process from the ready queue is next to go onto the processor (or one of the processors). A process that in the RUNNING state either has exhausted its time slice or it finishes. In the former case, the process's state is changed from RUNNING to READY, and the process is placed back in the ready queue where it awaits its turn to use the processor

again. In the latter case, the process's state is changed from RUNNING to FINISHED, and the process is removed from the system.

Completing this coursework successfully will give a good understanding of:

- Basic process creation and memory image overwriting
- Basic process scheduling algorithms and evaluation criteria.
- The use operating system APIs in Linux.
- Critical sections and the need to implement mutual exclusion.
- The basics of concurrent/parallel programming using an operating system's facilities (e.g. semaphores, mutex).

Submission requirements

You are asked to rigorously stick to the naming conventions for your source code and output files. The source files must be named `task.c`, any output files should be named `task.txt`. Ignoring the naming conventions above will result in you loosing marks.

For submission, create a single .zip file containing all your source code and output files in one single directory (i.e., create a directory first, place the files inside the directory, and zip up the directory). Please use your username as the name of the directory.

Copying Code and Plagiarism

You may freely copy and adapt any code samples provided in the lab exercises or lectures. You may freely copy code samples from the Linux/POSIX websites, which has many examples explaining how to do specific tasks. This coursework assumes that you will do so and doing so is a part of the coursework. You are therefore not passing someone else's code off as your own, thus doing so does not count as plagiarism. Note that some of the examples provided omit error checking for clarity of the code. You are required to add error checking wherever necessary.

You must not copy code samples from any other source, including another student on this or any other course, or any third party. If you do so then you are attempting to pass someone else's work off as your own and this is plagiarism. The University takes plagiarism extremely seriously and this can result in getting 0 for the coursework, the entire module, or potentially much worse.

Getting Help

You MAY ask Dr Qian Zhang and Dr Saeid Pourroostaei Ardakani for help in understanding coursework requirements if they are not clear (i.e. what you need to achieve). Any necessary clarifications will then be added to the Moodle page so that everyone can see them.

You may NOT get help from anybody else to actually do the coursework (i.e. how to do it), including ourselves. You may get help on any of the code samples provided, since these are designed to help you to do the coursework without giving you the answers directly.

Background Information

All code should be implemented in C and tested/run on the a Linux environment. When you compile your program using gcc, please do not forget to use the –pthread option to suppress the semaphore related compilation messages.

Please note that if you use sem_init to create a semaphore, it will not work in Mac OS unfortunately but it should work on most linux systems. It certainly works fine on the school linux server.

Additional information on programming in Linux, the use of POSIX APIs, and the specific use of threads and concurrency directives in Linux can be found, e.g., here (it is my understanding that this book was published freely online by the authors and that there are no copyright violations because of this) or the book 'Advanced Programming in UNIX environment'.

Coding and Compiling Your Coursework

You are free to use a code editor of your choice, but your code MUST compile and run on a Linux environment (school's Linux server if you are offline students). It will be tested and marked on these machines.

There are source files available on Moodle for download that you must use. To ensure consistency across all students, **changes are not to be made on these given source files.** The header file `coursework.h` contain a number of definitions of constants and several function prototypes. The source file `coursework.c` contains the implementation of these functions. Documentation is included in both files and should be self-explanatory. Note that, in order to use these files with your own code, you will be required to specify the file on the command line when using the gcc compiler (e.g. `gcc task.c coursework.c –pthread –o task`), and include the header file `coursework.h` in your code (using `#include “coursework.h”`).

Your code should always use the standard output (i.e. display) for any visualisations. Please **do not write your output directly into a file** since output files can be generated easily using redirections, e.g.:

```
./task > task.txt
```

IMPORTANT: Apart from the **number** setting defined in `coursework.h`, you are not allowed to change anything in the given source/header files. Code cannot be successfully compiled on linux environment with given source/header files will receive **ZERO** marks.

Requirement

You are asked to implement the Round Robin (RR) process scheduling algorithm using a bounded buffer with multiple producers and consumers to simulate how an Operating System performs process scheduling. You are also expected to calculate response and turnaround times for each of the processes, as well as averages for all jobs.

The bounded buffer must be implemented as a linked list and with a maximum capacity of N elements (where N defined by `BUFFER_SIZE` in the header file `coursework.h`). The new processes should be appended to the end of the list and removed from the front (you should maintain both the head and tail of the linked list for an efficient implementation in this case). The implementation should separate the manipulation of the linked list from the execution of the processes (that is, you are expected to define separate "add" and "remove" methods).

In this design, the BUFFER_SIZE models the maximum number of processes that can co-exist in the system is fixed by buffer size. There are multiple producers (dispatchers) and consumers (processors) running simultaneously. The producers generate process and add them to the end of the bounded buffer. The consumers will remove process from the start of the list and simulate them "running" on the processors (using the `runProcess()` and a `simulateRoundRobinProcess()` functions provided in the `coursework.c` file). These functions simulate the processes running on the processors for a certain amount of time, and update their state accordingly. The respective functions must be called every time a process runs.

Since both the producers and the consumers manipulate the same data structure/variables, synchronisation will be required. You are free to choose how you implement this synchronisation, but it must be as efficient as possible.

A successful submision include:

- Whether you have submitted the code and you are using the correct naming conventions and format.
- Whether the code compiles correctly, and runs in an acceptable manner in the Linux environment.
- Whether you manipulate your linked list in the correct manner.
- The correct use of dynamic memory and absence memory leaks. That means, your code correctly frees the elements in the linked list and its data values.
- A correct implementation of the RR algorithms.
- Correctly use the simulateRoundRobinProcess() function and correctly calculate (and display) the average response/turnaround times.
- Whether semaphores/mutexes are correctly defined, initialised, and utilised.
- Whether you have correct implementation of producer and consumer threads. i.e., The producer can only add processes to the buffer if spaces are available, the consumer can only remove processes from the buffer if processes are available. The producer and consumer must run in separate threads and with maximum parallelism.
- Whether consumers and producers end gracefully/in a correct manner.
- Whether your code is efficient, easy to understand, and allows for maximum parallelism.
- Whether your code runs free of deadlocks.
- Whether the output generated by your code follows the format of the examples provided on Moodle.