COMP2046 Coursework Autumn 2019

Weight: 25% module marks

Deadline: 6th Dec 2019, 12AM Local 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 implement simple shared memory problem and scheduling algorithm with a producer- consumer modle.

To maximise your chances of completing this coursework successfully, and to give you the best chance to get a good mark, the coursework is divided into multiple independent components, each one gradually more difficult to implement. The individual solutions should then be used/extended into the final program, which combines the principles of the individual programs. Completing this coursework successfully will give a good understanding of:

- Basic process creation, memory image overwriting and shared memory usage.
- 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 `TASKX.c`, any output files should be named `TASKX.txt`, with X being the number of the task. 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 school's Linux servers. 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).

Coding and Compiling Your Coursework

You are free to use a code editor of your choice, but your code MUST compile and run on the school's Linux servers. It will be tested and marked on these machines.

There are several 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, linkedlist.h`) contain a number of definitions of constants and several function prototypes. The source file (`coursework.c, linkedlist.c`) contain 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 –std=c99 TASKX.c coursework.c linkedlist.c"), and include the `coursework.h/linkedlist.c` file in your code (using `#include "coursework.h"/ #include "linkedlist.h"`).

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 school's linux servers with given source/header files will receive **ZERO** marks.

Task 1 – Shared Memory (40 marks):

Use the knowledge you've obtained during the labs, finish the following question. You are asked to implement a solution for using **shared memory** for inter-process communication. Requirements of implementation are:

- A parent process (TASK1.c) that create two new child processes;
- The process image of the two child processes is overwritten to execute ChildP1.c and ChildP2.c respectively.
- In TASK1.c, you are required to randomly generate an integer (RandInt) within the range of 1 to 20 and print out the value of RandInt. The integer RandInt will be stored in a shared memory with appropriate structure for future usage.
- The ChildP1 will wait for a random amount of time and then double the value of RandInt.
- As soon as ChildP1 modified the RandInt, the other child process ChildP2 subtract a value of 10 from the existing value.
- ChildP1 and ChildP2 need to take turn to perform doubling and subtraction on RandInt for 10 times.
- Make sure that the ChildP2.c won't be able to access to the RandInt before ChildP1.c finishes its task. Mutex and Semephore are **NOT** allowed to be used in this task.
- Use shared memory to store all the values (intermediate results) of RandInt.
- Once all child processes have finished their tasks, parent process need to print out the value of RandInt at each instance. Sample output:

```
The RandInt = 15, created by the parent process
In round 1, RandInt = 30, child process 1,
In round 1, RandInt = 20, child process 2,
In round 2, RandInt = 40, child process 1,
In round 2, RandInt = 30, child process 2,
In round 3, RandInt = 60, child process 1,
In round 3, RandInt = 50, child process 2,
```

. . .

The criteria used in the marking **TASK1.c**, **ChildP1.c** and **ChildP2.c** of your coursework 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.
- Whether you utilise and manipulate your Processes and shared memory in the correct manner.
- Whether the two child processes take turn to modify the shared memory.
- Whether the structure of shared memory is designed in an appropriate manner.
- Whether all the three processes end gracefully/in a correct manner.
- Whether the shared memory has been controlled in an appropriate manner.
- Whether the output generated by your code follows the format of the examples

For this task, you are required to submit three files: **TASK1.c**, **ChildP1.c** and **ChildP2.c**

Task 2: Static Process Scheduling (20 marks):

The goal of this task it to implement two basic process scheduling algorithms (**First Come First Served** (FCFS) and **Priority Queues** (PQ)) in a static environment. That is, all jobs are known at the start. You are expected to calculate response and turnaround times for each of the processes, as well as averages for all jobs. Note that the priority queue algorithm uses a **Round Robin** within the priority levels.

Both algorithms should be implemented in separate source files (`TASK2a.c` for FCFS, `TASK2b.c` for PQ) and use the linked list implementation provided in (`linkedlist.c` and `linkedlist.h`) for the underlying data structure. In both cases, your implementation should contain a function that generates a pre-defined NUMBER_OF_PROCESSES (this constant is defined in `coursework.h`) and stores them in a linked list (using a FCFS approach in both cases). This linked list simulates a **ready queue** in a real operating system. The implementation of the FCFS and PQ will remove jobs from the ready queues in the order in which they should run. Note that you are expected to use multiple linked lists for the PQ algorithm, one for each priority level. **In the linked list ready queue implementation, the process will be added to the end and removed from the front.**

You must use the coursework source and header file provided on Moodle (`linkedlist.c`, `linkedlist.h`, `coursework.c` and `coursework.h`). The header file contains a number of definitions of constants, a definition of a simple process control block, and several function prototypes. The source file (`coursework.c`) contains the implementation of these function prototypes and must be specified on the command line when compiling your code.

In order to make your code/simulation more realistic, a `runNonPreemptiveJob()` and a `runPreemptiveJob()` function are provided in the `coursework.c` file. These

functions simulate the processes running on the CPU for a certain amount of time, and update their state accordingly. **The respective functions must be called every time a process runs.**

The criteria used in the marking **TASK2a.c** and **TASK2b.c** of your coursework include:

- Whether you submitted code, the code compiles, the code runs.
- Your code **must compile** using `gcc -std=c99 TASK2a/b.c linkedlist.c coursework.c` on the school's linux servers.
- The code to generate a pre-defined number of processes and store them in a linked list corresponding to the queue in a FCFS fashion.
- Whether jobs are added at the end of the linked list in an efficient manner, that is by using the tail of the linked list rather than traversing the entire linked list first.
- Whether the correct logic is used to calculate (average) response and (average) turnaround time for both algorithms. Note that both are calculated relative to the time that the process was created (which is a data field in the process structure).
- Whether the implementation of the FCFS and PQ algorithms is correct.
- Correct use of the appropriate run functions to simulate the processes running on the CPU.
- 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.
- Code to that visualises the working of the algorithms and that generates output similar to the example provided on Moodle.

For this task, you are required to submit three files: **TASK2a.c** and **TASK2b.c**

Task 3: Dynamic Process Creation/Consumption (40 marks)

In task 2, we assumed that all processes are available upon start-up. This is usually not the case in real world systems, nor can it be assumed that an infinite number of processes can simultaneously co-exist in an operating system.

Therefore, you are asked to implement the process scheduling algorithms from task 2 (FCFS and PQ) using a bounded buffer with multiple producer and consumer solution to simulate how an Operating System performs process scheduling. In this design, the size models the maximum number of processes that can co-exist in the system is fixed by buffer size, there are multiple dispatchers (producer) add processes to the ready queue while multiple CPUs (consumer) select processes to run simultaneously. To do this, you have to extend the code tasks 2a and 2b to a bounded buffer with multiple producer/consumer solution. Similarly to task 2, both solutions should be implemented in separate source files (`TASK3a.c` for FCFS, and `TASK3b.c` for PQ).

In both cases (FCFS and PQ), the bounded buffers must be implemented as a linked list. In the case of FCFS, the maximum number of elements in the list should not exceed N, where N is equal to the maximum co-exist process number defined by MAX_BUFFER_SIZE (defined in `coursework.h`). In the case of the PQs, the maximum number of elements **across all**

queues (every priority level is represented by a separate linked list) should not exceed MAX_BUFFER_SIZE. The producers generate process and adds 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 CPU (using the `runNonPreemptiveJob()` (FCFS) and `runPreemptiveJob()` (PQ) functions provided in the `coursework.c` file).

In the case of Priority Queues, jobs that have not fully completed in the ''current run'' (i.e. the remaining time was larger than the time slice) must be added to the end of the relevant queue/buffer again. Note that at any one point in time, there shouldn't be more than MAX_BUFFER_SIZE jobs in the system (that is, the total number of processes currently running or waiting in the ready queue(s)).

The final version of your code should include:

- A linked list (or set of linked lists for PQs) of jobs representing the bounded buffer utilised in the correct manner. The maximum size of this list should be configured to not exceed, e.g., 50 elements.
- Multiple producer threads that generate a total number of MAX_NUMBER_OF_JOBS processes, and not more. That is, the number of elements produced by the different threads sums up to MAX_NUMBER_OF_JOBS. Elements are added to the buffer as soon as free spaces are available. A consumer function that removes elements from the end of the buffer (one at a time).
- A mechanism to ensure that all consumers terminate gracefully when MAX_NUMBER_OF_JOBS have been consumed. The code to:
    - Declare all necessary semaphores/mutexes and initialise them to the correct values.
    - Create the producer/consumer threads.
    - Join all threads with the main thread to prevent the main thread from finishing before the consumers/producers have ended.
    - Calculate the average response time and average turnaround time and print them on the screen when all jobs have finished.
    - Synchronise all critical sections in a correct and efficient manner, and only when strictly necessary keeping the critical sections to the smallest possible code set(s).
    - Generate output similar in format to the example provided on Moodle for this requirement (for 100 jobs, using a buffer size of 10).

The criteria used in the marking **TASK3a.c** and **TASK3b.c** of your coursework include:

- Whether you have submitted the code and you are using the correct naming conventions and format.
- Your code **must compile** using `gcc -std=c99 TASK3a/b.c linkedlist.c coursework.c -pthread` on the school's linux servers.
- Whether the code compiles correctly, and runs in an acceptable manner.
- Whether you utilise and manipulate your linked list in the correct manner.
- Whether semaphores/mutexes are correctly defined, initialised, and utilised.
- Whether consumers and producers are joined correctly.

- Whether the correct number of producers and consumers has been utilised, as specified in the coursework description.
- Whether consumers and producers end gracefully/in a correct manner.
- Whether consumers/producers have been assigned a unique id (as can be seen from the output provided on Moodle).
- Whether the exact number of processes is produced and consumed.
- Whether the calculation of (average) response and (average) turnaround time remain correct.
- Whether the integration of FCFS/PQ is correct for the bounded buffer problem.
- 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.

For this task, you are required to submit three files: **TASK3a.c** and **TASK3b.c**