Operating Systems and Concurrency (COMP2046)

Deadline: 13th December 2018 16:00.

Weight: 25% of the module mark (each task is worth 20% of this coursework which is equivalent to 5% of the module mark).

Format: a single zip file through Moodle (see submission requirements).

Overview

The goal of this coursework is to make use of Linux APIs and simple concurrency directives to solve a number of synchronisation problems that occur on scheduling systems of fairly simple operating systems.

To maximise your chances of completing this coursework successfully, it is divided into multiple tasks. The later tasks will build upon the experience and knowledge you have gained in the earlier tasks.

Completing all tasks will give you a good understanding of:

- The use of operating system APIs in Linux.
- Critical sections, semaphores, and the principles of synchronisation.
- The implementation of linear bounded buffers of a fixed size.
- The basics of concurrent/parallel programming using an operating system's functionalities.
- Different process scheduling algorithms.

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.

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.

You should ensure that your code compiles using the GNU C-compiler, e.g., with the command: gcc -std=c99 taskX.c coursework.c osc_queue.c

Note that your code should always use the standard output (i.e. display) for any visualisations. Please do not write your output directly into a file.

Copying Code and Plagiarism

You may freely copy and adapt any of the 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. Error checking may however be necessary in your code (and may help you with debugging).

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 Qian Zhang, Fei Yang or Prapa Rattadilok 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 and runnable on the school's Linux servers. An additional tutorial on compiling source code in Linux using the GNU c-compiler can be found on the Moodle page.
- Additional information on programming in Linux, the use of POSIX APIs, and the specific use of concurrency directives in Linux can be found, e.g., here:
  http://richard.esplins.org/static/downloads/linux_book.pdf
  It is our understanding that this book was published freely online by the authors and that there are no copyright violations because of this.
- Note that information on functions in the APIs can be easily retrieved, for instance, by typing man shm_open on the command line.

- Additional information can be found in, e.g.:
    - Tanenbaum, Andrew S. 2014 Modern Operating Systems. 4th ed. Prentice Hall Press, Upper Saddle River, NJ, USA., Chapter 2, section 2.3.4
    - Silberschatz, Abraham, Peter Baer Galvin, Greg Gagne. 2008. Operating System Concepts. 8th ed. Wiley Publishing, Chapter 4 and 5
    - Stallings, William. 2008. Operating Systems: Internals and Design Principles. 6th ed. Prentice Hall Press, Upper Saddle River, NJ, USA, Chapter 5

  In addition to these books, much of the information in the lecture notes should be extremely helpful for some of the tasks.

Source/Header Files Provided

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) contains a number of definitions of constants and several function prototypes. The source file (coursework.c) contains the implementation of these function prototypes. 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 coursework.c file on the command line when using the gcc compiler (e.g. “gcc –std=c99 task1.c coursework.c”), and include the coursework.h file in your code (using #include “coursework.h”).

In addition to the coursework source and header files, a third file (osc_queue.h) is provided. This contains the function prototypes and element definitions that you should follow for the implementation of your ‘queue’ (which is used throughout all the later tasks). Again, the file contains documentation that should be self-explanatory. The reason for providing this file it to ensure that you successfully implement a nice and generic implementation of a ‘queue’.

Output Samples Provided

Outputs samples for the different tasks are provided on Moodle. Your code should generate outputs that look similar to those examples, but note that the numeric values in your output may differ due to non-deterministic behaviour of processes.

Task 1: Queue

This task focusses on the implementation of two key data structures: ‘queue’ and ‘element’. The ‘queue’ will be used to store a number of ‘element’. You are provided with a header file on Moodle (osc_queue.h) that contains the required function prototypes with their corresponding arguments. You are expected to implement these function prototypes in a separate source file (called osc_queue.c).

Your osc_queue.c code should implement the functions to:

- add an ‘element’ at the beginning of the ‘queue’ (addFirst)
- add an ‘element’ at the end of the ‘queue’ (addLast)
- add an ‘element’ at a specific location in the ‘queue’ (addHere)
- free all dynamic memory allocation used by the ‘queue’ (freeAll)
- get the total number of the ‘element’ in the ‘queue’ (getCount)
- use malloc to create a specified number of ‘element’ within the ‘queue’ and initialise all variables to zero (init). The ‘max’ variable should be initialised using arr_size.
- print the content of the ‘queue’ (printAll)
- remove the last ‘element’ from the ‘queue’ (removeLast)

You are also provided with coursework.c and coursework.h on Moodle that contains the function prototypes with their corresponding arguments. You are expected to correctly implement these function prototypes and demonstrate FIFO and LIFO in a separate file (task1.c).

For FIFO, *MAX_ PROCESSES* should be added to the beginning of the ‘queue’. Once all ‘element’ are added, the program should remove the ‘element’ one at a time from the end of the ‘queue’ until the ‘queue’ is empty, and display the data value for each ‘element’ on the screen whilst removing it.

For LIFO, *MAX_ PROCESSES* should be added to the end of the ‘queue’. Once all ‘element’ are added, the program should remove the ‘element’ one at a time from the end of the ‘queue’ until the ‘queue’ is empty, and display the data value for each ‘element’ on the screen whilst removing it.

Notes:

- Both addFirst, addHere and addLast must return 0 if the ‘element’ can be added to the queue. Otherwise, the function should return 1 if it fails to add a new item i.e. the ‘queue’ is completely filled. The output of the function should be handled appropriately.
- For addHere, prior to adding an ‘element’ in a specific location, all of the ‘element’ following the specified location must be relocated to the right. In the case when the ‘queue’ is empty, the function should add the ‘element’ to the beginning of the queue.
- If malloc in init() is successful, notification similar to the example provided must be provided.
- If malloc in init() is not successful, appropriate notification must be provided, the function must also return 1 then exit the program. Your task1.c must correctly use this output from init().

- generateProcess() should be called to add a new ‘element’ to the ‘queue’.
- printAll must print both the count and the content of the ‘queue’ as output similar to the example provided on Moodle.
- #define directive *MAX_PROCESSES* must be used in task1.c to declare the total number of *‘element’* in ‘queue’.

Marking criteria:

The criteria used in the marking of Task 1 of your coursework include:

- Correctness of the implementation of ‘queue’ that allows ‘element’ to be added at the beginning and at the end of the ‘queue’.
- Correctness of the implementation of ‘queue’ that allows ‘element’ to be removed from the end of the ‘queue’.
- The use of a separate source file for the implementation of the ‘queue’ and ‘element’ (osc_queue.c), so that your code is easy to re-use in the later tasks of this coursework.
- Correctness of your code to achieve all requirements in ‘Notes’.
- The correct internal use of the ‘element’ and ‘queue’ structure in your implementation.
- The correct use of malloc and free in your implementation.
- The correct code to visualise the working of the algorithms and generate output similar to the example provided on Moodle.
- A sample output file (task1.txt) generated with your own code for 5 ‘element’ (N = 5).

Submission requirements:

- Name your code “task1.c”, “osc_queue.c” and the output “task1.txt”. Please stick rigorously to the naming conventions, including capitalisation.
- Your code must compile using gcc -std=c99 task1.c osc_queue.c coursework.c on the school’s linux servers.
- You must submit your code on Moodle and the source file must be configured for 5 ‘element’ (N = 5).

Task 2: Shortest Job First (SJF) - Static Process Scheduling

The goal of this task it to implement a basic process scheduling algorithm, Shortest Job First (SJF) in a static environment. That is, all jobs are known at the start (in practice, in a dynamic environment, jobs would show up over time). You are expected to calculate response and turnaround times for each of the processes, as well as averages for all jobs.

The algorithm should be implemented in a separate source file (task2.c) and use the 'queue' developed in Task 1 for the underlying data structure. Your implementation should contain the following three functions:

- (generateSJF): generates a pre-defined *MAX_ PROCESSES* and stores them in the 'queue' corresponding to the queue in a SJF fashion. The longest job should be stored at the beginning of the 'queue' i.e. index 0. This 'queue' simulates a ready queue in a real operating system. The function should use MAX_PROCESSES to create the desired number of 'element'. If the duration of the jobs are equal, they should be queued in first come first served fashion.
- (runSJF): runs the processes according to their 'pid_time'. The function should also calculate the response time and the turnaround time for each process. The function should use MAX_PROCESSES to run through all of the processes.
- (average): calculates the average of the values in an array.

You must use the coursework source and header file provided on Moodle (coursework.c and coursework.h). The header file contains a number of definitions of constants and several function prototypes. The source file (coursework.c) contains the implementation of these function prototypes. Documentation is included in both files and should be self-explanatory.

In order to make your code more realistic, a runNonPreemptiveJob function is provided in the coursework.c file. This function simulates the processes running on the CPU for a certain amount of time and must be called every time a process runs.

Marking criteria:

The criteria used in the marking of Task 2 of your coursework include:

- Whether your submitted code compiles and runs.
- The correctness of the three required functions: generateSJF, runSJF and average.
- The correct use of the appropriate functions from coursework.c and coursework.h.
- The correct use of malloc and free in your implementation.
- Code to that visualises the working of the algorithms and that generates output similar to the example provided on Moodle.
- A sample output file (task2.txt) generated with your own code for 5 'element' (N = 5).

Submission requirements:

- Name your code “task2.c” and the output for the task “task2.txt”. Please stick rigorously to the naming conventions, including capitalisation.
- Your code must compile using gcc –std=c99 task2.c osc_queue.c coursework.c on the school’s linux servers.
- You must submit your code on Moodle and the source file must be configured for 5 ‘element’ (N = 5).

Task 3: Priority Queue (PQ) - Static Process Scheduling

The goal of this task it to implement a basic process scheduling algorithm, Priority Queue (PQ) in a static environment. That is, all jobs are known at the start (in practice, in a dynamic environment, jobs would show up over time). 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.

The algorithm should be implemented in separate source files (task3.c) and use the 'queue' developed in Task 1 for the underlying data structure. Your implementation should contain the following three functions:

- (generatePQ): generates a pre-defined *MAX_ PROCESSES* and stores them in the 'queue' corresponding to the queue in a PQ fashion. This 'queue' simulates a ready queue in a real operating system. The function should use MAX_PROCESSES to create the desired number of 'element'. Note that you are expected to use multiple 'queue' for the PQ algorithm, one for each priority level.
- (runPQ): runs the processes according to their 'pid_time'. The function should also calculate the response time and the turnaround time for each process. The function should use MAX_PROCESSES to run through all of the processes.
- (average): calculates the average of the values in an array.

You must use the coursework source and header file provided on Moodle (coursework.c and coursework.h). The header file contains a number of definitions of constants and several function prototypes. The source file (coursework.c) contains the implementation of these function prototypes. Documentation is included in both files and should be self-explanatory.

In order to make your code more realistic, a runPreemptiveJob function is provided in the coursework.c file. This function simulates the processes running on the CPU for a certain amount of time and must be called every time a process runs.

Marking criteria:

The criteria used in the marking of Task 3 of your coursework include:

- Whether your submitted code compiles and runs.
- The correctness of the two required functions: generatePQ, runPQ and average.
- The correct use of the appropriate functions from coursework.c and coursework.h.
- The correct use of malloc and free in your implementation.
- Code that visualises the working of the algorithms and that generates output similar to the example provided on Moodle.
- A sample output file (task3.txt) generated with your own code for 5 'element' (N = 5).

Submission requirements:

- Name your code “task3.c” and the output for the tasks “task3.txt”. Please stick rigorously to the naming conventions, including capitalisation.
- Your code must compile using gcc –std=c99 task3.c osc_queue.c coursework.c on the school’s linux servers.
- You must submit your code on Moodle and the source file must be configured for 5 ‘element’ (N = 5).

Task 4: Dynamic Process Creation/Consumption with Bounded Buffer (SJF)

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 process can simultaneously co-exist in an operating system. Typically, there is an upper limit on the number of processes that can exist at any one time. This is determined by the size of the process table.

Therefore, you are asked to implement the process scheduling algorithm from Task 2 (SJF) using a bounded buffer (of which the size models the maximum number of processes that can co-exist in the system) to simulate how an Operating System performs process scheduling. The buffer can contain at most MAX_BUFFER_SIZE elements. You are asked to implement a single producer and a single consumer solution. The total number of elements generated is MAX_NUMBER_OF_JOBS, where MAX_NUMBER_OF_JOBS > MAX_BUFFER_SIZE.

The producer generates element and adds it to the appropriate location of the buffer. The consumer removes the elements from the end of the buffer i.e. the jobs is sorting in a descend manner and simulates “running” them. Each time the producer (consumer) adds (removes) an element, the number of elements currently in the buffer is shown on the screen.

Synchronisation will be required. You are free to choose how you implement this synchronisation, but it must be as efficient as possible. Your implementation should contain the following functions:

- A producer thread that generates elements and adds them to the buffer as soon as space is available.
- A consumer thread that removes one element at a time from the buffer and runs it.
- Correctly use the runNonPreemptiveJob() function and correctly calculate and display the average response time as well as the turnaround time.

Marking criteria:

- 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 buffer in the correct manner.
- Whether semaphores/mutexs are correctly defined, initialised, and utilised.
- Whether consumer and producer threads are joined correctly i.e. all threads should be joined with the main thread to prevent the main thread from finishing before the consumers/producers have ended.

- Whether the calculation of (average) response and (average) turnaround time remain correct.
- Whether the correct number of producers and consumers has been utilised.
- Whether consumers and producers end gracefully/in a correct manner.
- Whether the exact number of elements is produced and consumed.
- Whether your code is efficient, easy to understand, and allows for maximum parallelism (i.e. no unnecessary synchronisation is applied).
- Whether unnecessary/inefficient/excessive busy waiting is used in your code.
- Whether your code runs free of deadlocks.
- Whether the output generated by your code follows the format of the examples provided.

Submission requirements:

- Name your code “task4.c” and the output for the tasks “task4.txt” respectively. Please stick rigorously to the naming conventions, including capitalisation.
- Your code must compile using gcc -std=c99 task4.c osc_queue.c coursework.c -lpthread on the school’s linux servers.
- You must submit your code on Moodle and the sources file must be configured for 50 jobs, using a buffer size of 10.

Task 5: Dynamic Process Creation/Consumption with Bounded Buffer and Multiple Consumers (PQ)

You are asked to, similar to Task 4, implement a bounded buffer solution. However, this time with multiple consumers and Priority Queue algorithm. The producer generate a total of MAX_NUMBER_OF_JOBS elements, which are removed by multiple consumers. 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)).

Every time the producer or consumers adds or removes an element, the number of elements currently in the buffer is shown on the screen. Note that every consumer has a unique consumer ID assigned to them in the output. In addition to the requirements under task 4, your implementation must include:

- Set of queues of jobs representing the bounded buffer utilised in the correct manner. The maximum size of these queues should be configured to not exceed, e.g., 30 elements. (10 elements each queue)
- Correctly use the runPreemptiveJob() function and correctly calculate (and display) the average response/turnaround times.
- A producer thread that generates elements and adds them to the buffer as soon as space is available.
- A consumer thread that removes one element at a time from the buffer and runs it.
- A mechanism to ensure that all consumers terminate gracefully when MAX_NUMBER_OF_JOBS have been consumed.

Marking criteria:

- 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 buffer in the correct manner.
- Whether semaphores/mutexes are correctly defined, initialised, and utilised.
- Whether consumer and producer threads are joined correctly i.e. all threads should be joined with the main thread to prevent the main thread from finishing before the consumers/producers have ended.
- Whether the calculation of (average) response and (average) turnaround time remain correct.
- Whether the correct number of consumers has been utilised.
- Whether consumers and producer end gracefully/in a correct manner.

- Whether the exact number of elements is produced and consumed.
- Whether your code is efficient, easy to understand, and allows for maximum parallelism (i.e. no unnecessary synchronisation is applied).
- Whether unnecessary/inefficient/excessive busy waiting is used in your code.
- Whether your code runs free of deadlocks.

Submission requirements:

- Name your code “task5.c” and the output for the tasks “task5.txt” respectively. Please stick rigorously to the naming conventions, including capitalisation.
- Your code must compile using gcc -std=c99 task5.c osc_queue.c coursework.c -lpthread on the school’s linux servers.
- You must submit your code on Moodle and the sources file must be configured for 50 jobs, using a buffer size of 10 per queue.