G52OSC Coursework:
Process Scheduling, Operating System APIs, Threading, and Concurrency

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 solve a number of synchronisation problems that occur on scheduling systems that are similar to the ones that you may find in fairly simple operating systems.

To maximise your chances of completing this coursework successfully (and to give you the best chance to get a good mark), it is divided into multiple sub-tasks with different difficulty levels and multiple deadlines. 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, mutexes, and the principles of synchronisation/mutual exclusion.
- Basic process/thread scheduling algorithms and evaluation criteria for process scheduling.
- The implementation of linear bounded and unbounded buffers using linked lists.
- The basics of concurrent/parallel programming using an operating system's functionalities.
- Different process scheduling algorithms and the working of the scheduling algorithm in Windows 7.

Group work

You are allowed to work in pairs, but working on your own is also possible. If you decide to either work in pairs or alone, you will have to do so for all the tasks of this coursework! To confirm your choice, you are asked to complete and submit the coursework team file that can be found on Moodle to the submission system on Moodle. If working on your own, leave the second line of the file empty and nominate yourself as the administrator (in addition to filling out the first line). Please stick rigorously to the naming conventions to ensure that we can process the files automatically (for instance, submit it as a CSV file, not as an IOS "numbers" file).

(Hard) Deadline: 14th of October – On Moodle

If you work in pairs, you will be asked to anonymously complete a "Peer review assessment" that will be used to evaluate the contributions of both partners across all tasks. To do this, please download the "peer review template" spreadsheet from Moodle (you should be able to use any editor to edit it) and assess your partner. Note that, if working in team, only one submission is allowed per team, and that there is a separate submission system available for the peer review. When completing the peer review assessment form, you are asked to be fair and take effort and commitment into account more than actual ability. The template for the peer review is available on Moodle for download. Please do not change the format of the file.

Submission requirements and deadlines

This coursework is composed of 6 different tasks for which we have set up two submission deadlines:

- **Part 1: Tasks 1, 2 and 3** – Submission deadline is **8th of November 2019**

- **Part 2: Tasks 4 and 5** - Submission deadline is **13th of December 2019**

Note that **late submissions are not permitted, unless you have approved ECs.** You are encouraged to **submit your coursework solutions as many times as you want** before the deadline (to not risk not submitting at all if you miss the deadline). If you are applying for ECs before the deadline, we ask that (when possible) you submit your "current version" before the deadline. This can be overwritten with your new version if the ECs are approved.

We will do our best to mark your coursework as soon as possible. However, please be aware that we are unable to use automatic marking, and that we are marking all coursework ourselves. Hence, we would ask for your understanding that this may take up some time, but will do our best to meet the 3 week deadline.

You are asked to rigorously stick to the naming conventions when submitting your source code and output files. The source files must be named `taskX.c` (case sensitive), any output files should be named `taskX.txt` (case sensitive) with X being the number of the task (on occasions followed by a letter for the different sub-task). When marking your code, we will compile it automatically using scripts/make files. Ignoring the naming conventions above may make it more challenging for us to get your code compiled **(and could result in you loosing marks)**.

For each 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 do not to include the source or header files we provide in your submission (coursework and linked list implementations). Finally, please use `firstNameLastName-StudentId-userName` as the name of the directory. An "example directory/file structure" is provided in Moodle if you are unsure about the submission format.

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 servers** (e.g. bann.cs.nott.ac.uk). It will be tested/marked on these machines, and we cannot allow for potential differences between, e.g., Apple and Linux users.

You should **ensure** that your code compiles using the GNU C-compiler, e.g., with the command:

```
gcc -std=c99 taskX.c coursework.c linkedlist.c
```

And for those cases in which you use threads, you should add the pthread library as follows:

```
gcc -std=c99 taskX.c coursework.c linkedlist.c -pthread
```

This should generate an executable file called `a.out`, which you can run by calling "`./a.out`" from the command line. Note that, if you wanted to automatically run your code multiple times, you could do so by using the following on the command line:

```
for i in `seq 1 1000`; do ./a.out; done
```

This will run the same executable 1000 times sequentially (this is one of the ways in which we will test your code for deadlocks and segmentation faults ☺)

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** since output files can be generated easily using redirections, e.g.:

```
./a.out > taskX.txt
```

Make sure you redirect your output to a different file. Accidentally writing `./a.out > taskX.c` instead of `./a.out > taskX.txt` is easily done and will overwrite your source file (not that this has ever happened to us of course 😊).

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 (other than your coursework partner) 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 Geert De Maere, Isaac Triguero, or any of the lab helpers 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 or posted on the coursework forum so that everyone can see them.

You may **NOT get help from anybody else (other than your coursework partner) to actually do the coursework** (i.e. *how* to do it), including ourselves or the lab helpers. 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/runnable on the school's Linux servers (e.g. bann.cs.nott.ac.uk). 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 threads and 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 pthread_create` on the command line.
- Additional information on the bounded buffer problem 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**. The header file (`coursework.h`) 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. We also provide an implementation of a generic linked list that must be used as it is (`linkedlist.h`, and `linkedlist.c`). Documentation is included in all files and they should therefore 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"`).

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 the process schedulers on the school's servers.

COMP2007: Coursework Part 1

Task 1: Static Process Scheduling

The goal of this task is to implement two basic process scheduling algorithms: **First Come First Served** (FCFS) and **Round Robin** 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, which is what we will be simulating in later tasks). You are expected to calculate response and turnaround times for each of the processes, as well as averages for all jobs.

Both algorithms should be implemented in separate source files (`task1a.c` for FCFS, `task1b.c` for RR) and use the linked list provided as underlying data structure. In both cases, your implementation should contain a function that generates a pre-defined `NUMBER_OF_JOBS` (this constant is defined in `coursework.h`) and stores them in a linked list (using a FIFO approach in both cases). This linked list simulates the **ready queue** in a real operating system. Your implementation of FCFS and RR will remove jobs from the ready queues in the order in which they should run. Note that you will be required to use multiple linked lists in the second part of this coursework, which is why we provided a generic linked list implementation using "pointers to pointers".

You must use the coursework and linked list source and header files provided on Moodle (`coursework.c`, `coursework.h`, `linkedlist.c`, and `linkedlist.h`). The header files contain a number of definitions of constants, a definition of a basic process control block, and several function prototypes. The source files (`coursework.c`, and `linkedlist.c`) contain the implementation of these function prototypes. As explained above, these files must be specified on the command line when compiling your code. Documentation is included in all source files, which should make them self-explanatory.

To make your code/simulation more realistic, a `runNonPreemptiveJob()` (FCFS) and a `runPreemptiveJob()` (RR) 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 (including the remaining burst time). **The respective functions must be called every time a process "runs" on the CPU.**

Marking criteria:

The criteria used in the marking task1a and task1b of your coursework include:

- Whether the files you submitted use the correct naming conventions.
- Whether you submitted code, the code compiles, the code runs in an acceptable manner.
- 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 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 implementations of the FCFS and the RR algorithms are 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 (we will use `valgrind` to test for memory leak).
- Code that visualises the working of the algorithms and that generates output from which the syntax is similar to the example provided on Moodle. Note that the exact numbers may differ slightly due to non-deterministic of the OS's process schedulers.
- Two sample output files generated with your own code for 100 jobs and a time slice of 5ms (for the RR).

Submission requirements:

- Name your code "`task1a.c`" (FCFS) and "`task1b.c`" (RR) and the output for the tasks "`task1a.txt`" and "`task1b.txt`", respectively. **Please stick rigorously to the naming conventions, including capitalisation.**

Your code **must compile** using `gcc -std=c99 task1a.c linkedlist.c coursework.c` **on the school's linux servers**.

Task 2: Unbounded Buffer with binary semaphores (a.k.a. producer/consumer problem)

You are asked to implement an ``imaginary unbounded buffer'' with a single producer and a single consumer. The producer adds jobs to the buffer, the consumer removes jobs from the buffer (provided that elements are available). Different implementations are possible, but we are asking you to use only **binary semaphores** in your implementation. That is, the value for the semaphores should never exceed 1. This corresponds to the first "implementation" discussed in the lecture slides (in which a semaphore is used to delay the consumer by putting it to sleep if no elements are available in the buffer).

A correct implementation of this requirement includes:

- A producer thread that generates a predefined number of elements (e.g. `NUMBER_OF_JOBS` = 1000), and increments a shared counter that keeps track of the number of elements in the "imaginary buffer" (i.e., you do not have to declare/implement a buffer in this requirement, e.g. there is no need to define an linked list of an unbounded size).
- A consumer function that removes elements and decrements the shared counter for every element removed.
- A visualisation function that displays the exact number of elements currently in the buffer on the screen every time an element is added to or removed from the buffer.
- The code to:
    - Define and initialise all semaphores that you require (note that you are only allowed to use binary semaphores).
    - Create the producer/consumer threads.
    - Join the consumer/producer threads with the main thread (preventing the main thread from ending before the consumers/producers have completed).
    - All semaphores have the correct value when your code terminates, and their values are printed on screen using a format `<semaphore name> = <semaphore value>, <semaphore name> = <semaphore value>, <semaphore name> = <semaphore value>,…` Note that the pthread API contains functions that enable you to retrieve the value of a semaphore.
    - Generate output that resembles the example provided on Moodle (assuming 1000 jobs instead of the 10 jobs in the example provided). Note that we will explicitly check that the output file you submit is different from the example provided on Moodle and is indeed generated by your own code.

Marking criteria:

The criteria used in the marking task2 of your coursework include:

- Whether the files you submitted use the correct naming conventions.
- Whether you submitted code, the code compiles, the code runs in an acceptable manner.
- Whether binary semaphores are defined in a correct manner, initialised, and utilised.
- Whether consumer and producer threads are joined correctly.
- Whether consumers and producers end gracefully/in a correct manner.
- Whether the exact number of elements is produced and consumed.
- Whether the visualisation function displays the correct number of elements in the buffer.
- Whether the values for the semaphores are displayed and the values are correct/consistent.
- 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 on Moodle.

Submission requirements:

- Name your code "`task2.c`" and the output file "`task2.txt`". **Please stick rigorously to the naming conventions, including capitalisation.**
- Your code **must compile** using `gcc -std=c99 task2.c -lpthread` **on the school's linux servers**.

Task 3: Bounded Buffer with binary semaphores

You are asked to implement a FCFS bounded buffer (represented as linked list) using a binary semaphore (the value cannot exceed 1) to make the producer go to sleep, and a full "counting semaphore" that represents the number of "jobs" in the bounded buffer (note that you may use additional semaphores or mutexes to synchronise critical sections, but not to implement the "sleep mechanisms"). The buffer can contain at most MAX_BUFFER_SIZE (defined in `coursework.h`) elements, and each element contains one character (a '*' in this case). You are asked to implement a single producer and a single consumer. The producer generates NUMBER_OF_JOBS (defined in `coursework.h`) '*' characters, adds them to the end of the buffer, and goes to sleep when the buffer is full. The consumer removes '*' characters from the start of the buffer (provided that "jobs" are available). Each time the producer (consumer) adds (removes) an element, the number of elements currently in the buffer is shown on the screen as a line of stars (see output sample provided on Moodle). Different implementations and synchronisation approaches are possible, however, we are asking you to implement the sleep mechanism for the producer using a binary semaphore (or mutex if you believe this would be a better choice), and use a counting semaphore to represent the number of full buffers in your implementation.

The final version of your code will include:

- A linked list of characters representing the bounded buffer utilised in the correct manner. The maximum size of this list (MAX_BUFFER_SIZE) should be configured to not exceed, e.g., 50 elements.
- A producer function that generates stars ('*') and adds them to the end of the buffer (using `addLast`) as soon as space is available.
- A consumer function that removes elements from the start of the buffer using `removeFirst` (one at a time).
- A visualisation function that displays the exact number of elements currently in the buffer (using a line of stars) on the screen every time an element is added to or removed from the buffer.
- All semaphores have the correct value when your code terminates, and their values are printed on screen using a format `<semaphore name> = <semaphore value>, <semaphore name> = <semaphore value>, <semaphore name> = <semaphore value>,…` Note that the pthread API contains functions that enable you to retrieve the value of a semaphore.
- 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.
    - Generate output similar in format to the example provided on Moodle for this requirement (for 100 jobs, using a buffer size of 10).

Submission requirements:

- Name your code `"task3.c"` and the output file `"task3.txt"`. **Please stick rigorously to the naming conventions, including capitalisation.**
- Your code **must compile** using `gcc -std=c99 task3.c linkedlist.c coursework.c -lpthread` **on the school's linux servers**.

Marking criteria:

The criteria used in the marking of task 3 include:

- Whether the files you submitted use the correct naming conventions.
- Whether you submitted code, the code compiles, the code 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 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 on Moodle.

Part 1 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 do not include the source or header files we provided on Moodle (coursework and linked list implementations). Finally, please use your `firstNameLastName-StudentId-userName` as the name (for the team administrator if you work in pairs) of the directory.

If you have completed all parts of this submission successfully, the directory should contain:

- task1a.c, task1a.txt, task1b.c, task1b.txt
- task2.c, task2.txt
- task3.c, task3.txt

The deadline for this submission is **3pm, Friday 8th of November (Late submissions are not allowed, unless you have approved ECs).**

COMP2007: Coursework Part 2

Important Note: A new (and backwards compatible) version of "`coursework.c`" and "`coursework.h`" are available to facilitate the implementation of Part 2. Please download the new files from Moodle.

Task 4: A Simplified Windows Scheduler

In task 1, it was assumed that all jobs are available at the start. 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 (which typically has an upper limit on the number of processes that can exist at any one time, determined by the size of the process table).

The goal of this task to implement the process scheduling algorithms from task 1 (FCFS and RR) using multiple bounded buffers (each one representing a different queue at a different priority level) to approximate a Windows Process Scheduling System. To do so, you are asked to extend the code from task 3 to use **multiple consumers with a single producer.** This time, you are allowed to use multiple counting semaphores if you wish to do so. The code must integrate your implementation of FCFS and RR from task 1 and every consumer should have a unique "consumer id".

The producer thread is responsible for generating jobs and adding them to the appropriate queues based on their priority (`process->iPriority`). The consumer thread(s) iterate through the queues in order of priority and remove the first available job from the queues (with 0 representing the highest priority, 31 the lowest priority for MAX_PRIORITY set to 32). The consumer(s) also simulate the jobs "running" on the CPU, using the `runJob()` function which, depending on the priority level, calls `runNonPreemptiveJob()` for a FCFS queue (if priority < MAX_PRIORITY / 2) and the `runPreemptiveJob()` for the RR queues (if MAX_PRIORITY / 2 <= priority). All necessary functions are provided in the `coursework.c` file.

The final version of your code should use 32 bounded buffers (for MAX_PRIORITY set to 32). The bounded buffers must be implemented as linked lists (one for each priority level) and the maximum number of elements across all bounded buffers should not exceed MAX_BUFFER_SIZE. **Hint**: you could use arrays of the size MAX_PRIORITY to keep track of the heads and tails of the different linked lists.

Different queues will use different process scheduling algorithm:

- The first set of queues (for priorities in [0, `MAX_PRIORITY/2[`) should use a FCFS algorithm to consume jobs.
- The second set of queues (for priorities in [`MAX_PRIORITY/2, MAX_PRIORITY[`) should use a RR algorithm. In this case, jobs that have not fully completed in their ''current run'' (i.e. the remaining burst time is not 0) must be added to the end of the relevant queue/buffer again.

Note that, as stated above, the total number of jobs in the system should not exceed MAX_BUFFER_SIZE (that is, the total number of jobs currently running or waiting in the ready queue(s)).

For the successful completion of this task, we recommend the following steps:

1. Extend your bounded buffer implementation from task 3 to multiple consumers and integrate your FCFS algorithm from task 1 into the code without considering priorities. This means that you could work with a single bounded buffer. The producer puts at total NUMBER_OF_JOBS (defined in `coursework.h`) in the bounded buffer, which can contain at most MAX_BUFFER_SIZE jobs at any one time. The consumer(s) take jobs from the buffer in a FCFS fashion.

2. Create arrays of size MAX_PRIORITY to contain the heads and tails for the bounded buffers corresponding to the different priority levels. The producer is responsible for creating and adding the jobs in the correct queue (based on their priority). The consumers remove jobs from the queues and simulate them running in a FCFS fashion (using `runNonPreemptiveJob()`).
3. Introduce RR for the second set of queues. Remember that this implies that jobs that haven't finished need to be put back at the end of their current priority queue without exceeding the maximum number of jobs that can co-exist in the system.

If you don't manage to complete all the above steps, we encourage you to submit your partially completed code indicating (as a comment in the code) which steps you have achieved.

The final version of your code should include:

- Multiple linked lists that represent the bounded buffers, one for each priority level. Note that the total number of jobs in the system should never exceed MAX_BUFFER_SIZE.
- A producer thread that generates jobs until the NUMBER_OF_JOBS is reached, and not more. Elements are added to the buffer as soon as a free space is available.
- A consumer function that removes jobs from the queues in the correct order and simulates their running (using the `runJob()` function).
- A mechanism to ensure that all consumers terminate gracefully when NUMBER_OF_JOBS have been consumed.
- The code to:
    - Create the producer thread and the consumer threads, assigning a unique ID to each consumer.
    - Join all threads with the main thread to prevent the main thread from finishing before the consumers/producer 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. Critical sections should be kept to the smallest possible set(s) of instructions.
    - Generate output similar in format to the example provided on Moodle for this requirement (for 1000 jobs, using a buffer size of 100) and two consumers. You are allowed to modify the definition of the relevant constants in the `coursework.h` file to create this output.

To facilitate the implementation of this task, we have provided:

- A processJob() function that prints the correct output on the screen and sums the response and turnaround times. This function should be included in your task4.c and may require further synchronisation.
- A `runJob()` function that calls `runNonPreemptiveJob()` for a FCFS job and `runPreemptiveJob()` for a RR job.

Marking criteria:

The criteria used in the marking of task 4 include:

- Whether you have submitted the code and you are using the correct naming conventions and format (see submission format available on Moodle).
- Whether the code compiles and runs in an acceptable manner.
- Whether you utilise and manipulate your linked lists in the correct manner.
- Whether semaphores/mutexes are defined, initialised, and utilised in a correct manner.

- Whether consumers and producer are joined in a correct manner.
- Whether the correct number of consumers has been utilised, as specified in the coursework description.
- Whether consumers and producer threads end gracefully/in a correct manner.
- Whether consumers have been assigned a unique id (as can be seen from the output provided on Moodle).
- Whether the exact number of jobs are produced and consumed.
- Whether the calculation of (average) response and (average) turnaround times remain correct.
- Whether synchronisation is done efficiently.
- Whether the integration of FCFS/RR is correct for the bounded buffer problem.
- Whether your code is efficient, easy to understand, and allows for maximum parallelism.
- Whether unnecessary/inefficient/excessive busy waiting is not applied within your code.
- Whether your code runs free of deadlocks.
- Whether the output generated by your code follows the format of the examples provided on Moodle.

Submission requirements:

- Name your code "`task4.c`" and the output file "`task4.txt`". **Please stick rigorously to the naming conventions, including capitalisation.**
- Your code **must compile** using `gcc -std=c99 task4.c linkedlist.c coursework.c -lpthread` **on the school's linux servers**.

Task 5: Pre-emptive FCFS and Priority Boosting

Note: This task is meant as a challenge, and carries between 5 and 10% of the total coursework mark (that is, at most 1.5 to 3% of your total module mark). It is aimed at students who would like to put their knowledge to the test (which we of course would like to offer support for). Assuming that you have submitted all the previous requirements on time and that your implementations are correct, you will still be able to obtain 90% to 95% for your coursework without doing this task.

You are asked to extend the code from task 4: (1) to allow the producer thread to pre-empt a FCFS job from running if a higher priority job was generated, and (2) to boost the priority of RR jobs (i.e. implement dynamic priority boosting).

Pre-emptive FCFS:

- The idea is that if a higher priority FCFS job shows up, the lowest priority job that is currently running is interrupted to allow the (new) higher priority job to run. A `preemptJob()` is included in `coursework.c` that enables you to interrupt a job.
- To successfully simulate a pre-emptive FCFS algorithm, the average rate at which jobs arrive in the system should be roughly the same as the rate at which they are processed. In order to achieve this, you are allowed to use a sleep function with an average waiting time equal average duration of the jobs.
- Every time a FCFS job is pre-empted, a message should be printed on the screen in the following format shown below:

```
Pre-empted job: Pre-empted Process Id = 95, Pre-empted Priority 13, New Process Id 113, New priority 1
```

Priority Boosting for RR jobs:

Priority boosting can be implemented as a separate thread that iterates through the different RR priority levels (i.e. 16 to 31). The thread checks the first job in every priority level, and if the job has not run for a pre-determined amount of time - called the boost interval, e.g. 100ms, the priority of the job is boosted to the maximum level for RR jobs (level 16 for MAX_PRIORITY set to 32), independent of what the priority of the original job is. The boost interval is determined by the value of the `BOOST_INTERVAL` constant defined in the `coursework.h` file. Note that, to keep things simple, the priority booster loops through the different priority levels and only checks the first job in the respective queue. If a job's priority is boosted, a message is printed in the output following the format below:

```
Boost priority: Process Id = 95, Priority = 21, New Priority = 16
```

After the message is printed, the job is added to the front of the queue at the highest priority level using the `addFirst()` function (provided in `linkedlist.c`). It will thereby jump over the other jobs in this queue.

The final version of your code should:

- Include a function that checks if a newly generated FCFS job has a higher priority than any of the currently running FCFS jobs, and that preempts the lowest priority FCFS job to allow the new job to run.
- Include a booster thread that loops through each priority level and checks the first job in the queue. If it has not run for, e.g., 100ms, the priority is boosted by removing it from the front of its current queue, and adding it at the start of the highest priority RR queue.
- Include code to join the booster thread with the main thread.
- Ensure that the booster thread terminates automatically once the consumers have consumed all jobs.
- The code to print the "booster output" on the screen.

Tips:

- Much of the code for task 5 will remain the same as for task 4.
- Note that the booster thread and the consumer threads will interact with each other. That is, whilst the booster thread removes one of the jobs from a lower priority queue, it is temporarily not present in the queueing system, which means that the consumer may temporarily not "find" the given job in the queue (despite a "full" buffer being present). This can result in deadlocks, in particular when few jobs are left.
- Maintain an `iItemsProduced` and `iItemsConsumed` counter (which need to be synchronised appropriately)

Marking criteria:

The criteria used in the marking of task 5 include:

- The correctness of your FCFS pre-emption mechanism, which it is only applied on relevant queues.
- The correctness of your RR booster function, which it is only applied on relevant queues.
- The correct manipulation of the queues.
- The correct synchronisation between the consumer, producer and booster threads.
- The presence of deadlocks.
- The efficiency of your synchronisation.

Submission requirements:

- Name your code `"task5.c"`. **Please stick rigorously to the naming conventions, including capitalisation.**
- Your code **must compile** using `gcc -std=c99 task5.c linkedlist.c coursework.c -lpthread` **on the school's linux servers**.

Part 2 Submission

Create a single .zip file following the example provided on Moodle. This file should contain 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 do not include the source or header files we provided on Moodle (coursework and linked list implementations).

If you have completed all parts of this submission successfully, the directory should contain:

- task4.c, task4.txt
- task5.c

The deadline for this submission is **3pm, Friday 13th of December (Late submissions are not allowed, unless you have approved ECs).**