G52OSC Coursework: Operating System APIs, Threading, and Concurrency for Scheduling Systems

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 deadlines and difficulty levels. 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 buffers of a fixed size using linked lists.
- The basics of concurrent/parallel programming using an operating system's functionalities.
- Different process scheduling algorithms and the concept of priority boosting.

Group work

To complete the tasks associated with this coursework, 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 will have to submit an empty file on Moodle with the following naming convention:

- Working on your own: "<Your University ID (e.g. pszit) >.txt".
- Working in pairs: "<Your Partner University ID (e.g. pszgd) >.txt".

For example:

- Geert De Maere (pszgd) works with Isaac Triguero (pszit), Geert De Maere submits a file named pszit.txt, Isaac Triguero submits a file named pszgd.txt.
- Geert De Maere (pszgd) works on his own, Geert De Maere submits a file "pszgd.txt".

Note that:

- If Moodle does not accept empty files, please just put your partner's username in the file as well, or enter a blank.
- If working in team, both partners must submit a txt file.
- For this submission, we only accept txt files, that is, no png, zip, or rtf or docx files.
- If you do not have a coursework partner, but would like to work in team, then please drop us an email so we can introduce you to a potential partner.

(Hard) Deadline: 10th of October – On Moodle

If you work in pairs, for each single task, you are asked to anonymously complete a "Peer review assessment" that will be used to evaluate the individual contributions. 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. Both members of the group should **submit their own copy of each task** (which should be the same as your partner's submission). You should include the **peer review form** together with your individual submission and **you do not show it** to your partner (this is not optional). When completing the peer review assessment form, you are asked to be fair and take effort and commitment into account more than actual ability. Name this file peerReviewTaskX.csv.

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 (on occasions followed by a letter for the different sub-task – names are **case sensitive**). 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 task, 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 (that is, for Geert De Maere: **pszgd.zip**, for Isaac Triguero: **pszit.zip**).

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.

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. 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 (`linkedlist.h`) is provided. This contains the function prototypes and element definitions that you should follow for the implementation of your linked list (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 linked list.

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.

Part 1 - Requirements

Task 1: Generic Linked List

This task focusses on the implementation of the key data structure that will be used across all tasks of this coursework: **a flexible and generic implementation of a linked list**. You are provided with a header file on Moodle (`linkedlist.h`) that contains the required function prototypes (with their corresponding arguments) that you are expected to implement in a separate source file (called `linkedlist.c`).

Your code should implement the functions to:

- add elements at the beginning of the list (`addFirst`)
- add elements at the end of the list (`addLast`)
- remove the first element from the list (`removeFirst`)

You are asked to demonstrate the correct use of these functions in a separate source file (`task1.c`), by adding `N` ascending integers (i.e. 0, 1, 2, …) to the start and end of the list. Note that structure defined in the header file contains a void pointer rather than a pointer to a specific data type. Hence, your implementation of the linked list should allow you to add any data type with appropriate casting of the data types (this happens outside the linked list). Once all elements are added, the program should remove the elements one at a time from the beginning of the list until the list is empty, and display the data value for each element on the screen whilst removing it (first for FIFO, then for LIFO).

Tips:

- Note that if you pass a datatype using a pointer into a function, all changes made within the function to the variable will be visible outside this function as well. That is, the parameter is passed in "by reference". Since the head and tail (which are pointers to the first/last element) of the linked list may change when adding/removing an element, the head and tail have to be passed into the function using a pointer. Hence, you end up using a pointer (to head/tail) to a pointer (the first/last element). Only then will you be able to make changes to the head/tail pointer in the function that are visible outside the function.
- Making some simple drawings while coding may be helpful.

Marking criteria:

The criteria used in the marking task1 of your coursework include:

- Correctness of the implementation of a generic and flexible linked list that allows us to add elements at the beginning and at the end of the list, as well as removing the first element of the list.
- The use of a separate source file for the implementation of the linked list (`linkedlist.c`), so that your code is easy to re-use in the later tasks of this coursework.
- The correct initialisations of pointers to `NULL` whenever necessary.
- The correct use of dynamic memory and absence of memory leaks. That means, your code correctly frees the elements in the linked list and its data values.
- The correct internal use of the generic element structure 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 (`task1.txt`) generated with your own code for 100 elements (N = 100).

Submission requirements:

- Name your code "`task1.c`", "`linkedlist.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 linkedlist.c` on the school's linux servers.
- You must **submit your code on Moodle** and the source file must be **configured for 100 elements** (N = 100).
- Submission deadline: **Friday 19th October 3pm.**

Task 2: Static Process Scheduling

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 (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. 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 developed in task 1 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 (hence why we made you implement a generic version using "pointers to pointers" in task 1).

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, 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. Documentation is included in both files and should be self-explanatory.

In order to make your code/simulation more realistic, a `runNonPreemptiveJob()`(FCFS) and a `runPreemptiveJob()`(PQ) 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.**

Marking criteria:

The criteria used in the marking task2a and task2b of your coursework include:

- Whether you submitted code, the code compiles, the code runs.
- 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. We used a time slice of 5ms and 10 jobs to generate these sample outputs. Note that the exact numbers may differ slightly due to non-deterministic behaviour of processes (but the number should remain correct).
- Two sample output files generated with your own code for 100 jobs and a time slice of 5ms.

Submission requirements:

- Name your code "`task2a.c`" (FCFS) and "`task2b.c`" (PQ) and the output for the tasks "`task2a.txt`" and "`task2b.txt`" respectively. **Please stick rigorously to the naming conventions, including capitalisation.**
- Your code **must compile** using `gcc -std=c99 task2a.c linkedlist.c coursework.c` on the school's linux servers.
- You must **submit your code on Moodle** and the source file must be **configured for 100 elements** (N = 100)
- Submission deadline: **Friday 2nd November 3pm.**

Part 2 - Requirements

Task 3: Bounded buffers

In this task you will implement a bounded buffer (i.e. of fixed size) of characters in two different ways. In task 3a, you will:

- Use a single producer and single consumer.
- Synchronise using binary semaphores (or mutexes where appropriate).

In task 3b, you will:

- Use multiple producers and multiple consumers.
- Synchronise using counting semaphores (or mutexes where appropriate).

In both cases, you will produce/consume a pre-determined number of elements. Both tasks will give you good insights into the different synchronisation approaches for bounded buffers. These insights are required for the implementation of task 4 (and potentially for the exam ☺).

Task 3a: bounded buffer with binary semaphores

You are asked to implement a FCFS bounded buffer (represented as linked list). 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 MAX_NUMBER_OF_JOBS (defined in `coursework.h`) '*' characters and adds them to the end of the buffer (provided that empty spaces are available). The consumer removes '*' characters from the start of the buffer (provided that full elements 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 use **only binary semaphores (or mutexes where appropriate) 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 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.
- 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).

Task 3b: bounded buffer with counting semaphores

You are asked to, similar to task 3a, implement a bounded buffer of characters, however, this time with multiple producers, multiple consumers, and counting semaphores. The producers generate a total of MAX_NUMBER_OF_JOBS elements, which are removed by multiple consumers. Every time one of the producers or consumers adds or removes an element, the number of elements currently in the buffer is shown as a line of '*' characters (see output sample on Moodle). Note that every producer/consumer has a unique Id assigned to them in the output.

Similar to task 3a, a correct implementation of this requirement includes:

- A linked list of characters 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 '*' characters, 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. Note that every producer may end up producing a different number of elements, depending on the way threads are interleaved, but that the total items produced should be equal to MAX_NUMBER_OF_JOBS.
- A consumer function that removes elements from the start of the buffer using `removeFirst` (one at a time).
- A mechanism to ensure that all consumers terminate gracefully when MAX_NUMBER_OF_JOBS have been consumed.
- 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.
- 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).

Marking criteria:

The criteria used in the marking task3a and task3b of your coursework include:

- Whether you have submitted the code and you are using the correct naming conventions and format (see below).
- 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 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.

Submission requirements:

- Name your code "`task3a.c`" and "`task3b.c`" and the output for the tasks "`task3a.txt`" and "`task3b.txt`" respectively. **Please stick rigorously to the naming conventions, including capitalisation.**
- Create a single .zip file containing all your source code and output files (`linkedlist.c`, `task3a.c`, `task3a.txt`, `task3b.c`, `task3b.txt` and `peerReviewTask3.csv`) 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 (that is, for Geert De Maere: **pszgd.zip**, for Isaac Triguero: **pszit.zip**).
- Both partners submit the same code/source files (but different peer review forms).
- Your code **must compile** using `gcc -std=c99 task3a.c linkedlist.c coursework.c -lpthread` and `gcc -std=c99 task3b.c linkedlist.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 100 elements** (N = 100)
- Submission deadline: **Friday 16th November 3pm.**

Part 3 - Requirements

Task 4: Dynamic Process Creation/Consumption

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 (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).

Therefore, you are asked to implement the process scheduling algorithms from task 2 (FCFS and PQ) 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. To do this, you have to extend the code from task 3b (*or task 3a if you did not manage to complete 3b*) by integrating your implementation of the process scheduling algorithms implemented in tasks 2a and 2b. Similarly to task 2, both solutions should be implemented in separate source files (`task4a.c` for FCFS, and `task4b.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 number of processes and 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 producer(s) generate(s) jobs and adds them to the end of the bounded buffer. The consumer(s) will remove jobs 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).

Marking criteria:

The criteria used in the marking task4a and task4b of your coursework include:

- Whether you have submitted the code and you are using the correct naming conventions and format (see below).
- 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 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 `"task4a.c"` and `"task4b.c"` and the output for the tasks `"task4a.txt"` and `"task4b.txt"` respectively. **Please stick rigorously to the naming conventions, including capitalisation.**
- Create a single .zip file containing all your source code and output files (`linkedlist.c`, `task4a.c`, `task4a.txt`, `task4b.c`, `task4b.txt` and `peerReviewTask4.csv`) 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 (that is, for Geert De Maere: **pszgd.zip**, for Isaac Triguero: **pszit.zip**).
- Both partners submit the same code/source files (but different peer review forms).
- Your code **must compile** using `gcc -std=c99 task4a.c linkedlist.c coursework.c -lpthread` and `gcc -std=c99 task4b.c linkedlist.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 100 elements** (N = 100, and a buffer size of 10)
- Submission deadline: **Friday 23rd November 3pm.**

Part 4 – Requirements

Task 5: Priority Boosting

Note: This task is meant as a challenge, and will only carry at most 5 to 10% of the total coursework mark (that is, at most 1.5 to 3% of your total module mark). It aims 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% on your coursework without doing this task.

In this task, we are asking you to implement a priority boosting mechanism for task 4b. This can be implemented as a separate thread that iterates through the different priority levels, and checks the first process in every priority level. If this process has not run for, e.g., 100ms (`BOOST_INTERVAL`, an extra constant has been defined for this in `coursework.h`), the priority of the process is boosted to the maximum level (level 0 in this case), independently of what the priority of the process is. 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 process's priority is boosted, a message is printed in the output following the format below:

```
Boost priority: Process Id = 95, Priority = 21, New Priority = 0
```

After the message is printed, the process is added to the front of the queue for the highest priority level using the `addFirst()` function. It will thereby jump over the other jobs/processes in this queue.

The final version of your code should include:

- A booster thread that loops through each priority level and checks the first process/job in the queue. If it has not run for, e.g., 100ms, the priority is boosted.
- Code to join the booster thread with the main thread.
- The booster thread will automatically terminate once the consumers have consumed all processes (note that you can easily verify this by maintaining an iItemsConsumed counter).
- The code to remove the first element of the queue if it has not run for, e.g., 100ms.
- The code to add the job to the start of the highest priority queue using the `addFirst` method.
- 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 4b.
- Note that the booster thread and the consumer threads will interact with each other. That is, whilst the booster thread removes one of the processes 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 process in the queue (despite a "full" buffer being present). This can result in deadlocks, in particular when few processes/jobs are left.
- Maintain an iItemsProduced and iItemsConsumed counter (which need to be appropriately synchronised)

Marking criteria:

The key focus of marking will be on:

- The correctness of your booster function.
- 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`" and the output "`task5.txt`". **Please stick rigorously to the naming conventions, including capitalisation.**
- Create a single .zip file containing all your source code and output files (`linkedlist.c`, `task5.c`, `task5.txt`, and `peerReviewTask5.csv`) 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 and zip file (that is, for Geert De Maere: **pszgd.zip**, for Isaac Triguero: **pszit.zip**).
- Both partners submit the same code/source files (but different peer review forms).
- Your code **must compile** using `gcc -std=c99 task5.c linkedlist.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 100 elements** (N = 100, a buffer size of 10, 2 producers, 2 consumers, and a boosting interval of 100ms.)

Submission deadline: **Friday 14th of December 3pm.**