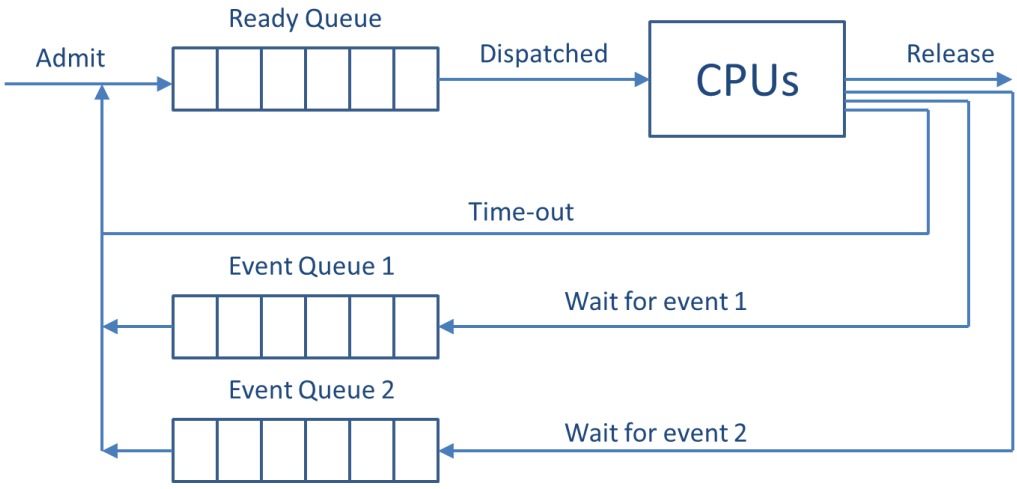
G52OSC Coursework:
Process Scheduling, Operating System APIs, Threading, and Concurrency

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 model a process scheduling system that is similar to ones that you may find in fairly simple operating systems. More specifically, we would like you implement the process scheduling system shown in the figure below, as discussed during the lectures.



A ready queue contains processes that have been admitted to the system, are in the READY state, and hence ready to run on the CPU. A process running (i.e. it is in the RUNNING state) on the CPU is simulated by calling one of the "simulateProcess()" functions. A process scheduling algorithm is used to determine which process from the ready queue is next to go onto the CPU (or one of the CPUs).

Several events can occur whilst a process is running on the CPU. Each event is defined by a unique Id. Each event type has associated with it an event queue. A process that blocks on a certain event type is placed in the corresponding event queue. A separate thread is responsible for "checking" the event queues at regular intervals and unblocks the first process in each of the queues (if one is present). A process that is unblocked is removed from the event queue, and placed back into the ready queue. If a process does not block during its execution, it either has exhausted its time slice or it finishes. In the former case, the process's state is changed from RUNNING to READY, and the process is placed back in the ready queue where it awaits its turn to use the CPU again. In the latter case, the process's state is changed from RUNNING to FINISHED, and the process is removed from the system.

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, each one of them getting gradually more difficult as you go

along. The later tasks will build upon the experience and knowledge you have gained in the earlier tasks. The last task is meant to be "a challenge" and corresponds to the full implementation of the above system.

Completing all tasks will give you a good understanding of:

- The different process states and the principles behind a possible queueing system that could be used as part of an operating system (although real world implementations are likely to be more complex).
- Basic process/thread scheduling algorithms and evaluation criteria for process scheduling.
- The use of operating system APIs in Linux.
- The implementation of linear bounded buffers of a fixed size using linked lists.
- Critical sections, semaphores, mutexes, and the principles of synchronisation/mutual exclusion.
- The basics of concurrent/parallel programming using an operating system's functionalities.

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 requirement (on occasions followed by a letter for the different sub-requirements – names are **case sensitive**). When marking your code, we may 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 loosing marks).

Coding 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 account for potential differences between, e.g., Apple and Linux users. You can compile your code using the GNU C-compiler with the command "`gcc task1a.c coursework.c`", adding any libraries that may be required to the end of the command. For instance, if you would like to use threads, you will have to specify "`gcc task2.c coursework.c -pthread`" on the command line. This will generate an executable file called `a.out`, which can be run by calling "`./a.out`" from the command line. Note that, if you automatically wanted to run your code multiple times, you could call "`for i in `seq 1 100`; do ./a.out; done`" from the command line, which will run the same executable 100 times sequentially (this is how we will test your code for deadlocks ☺)

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
```

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.

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 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 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.
- 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

Files Provided:

There are two source files available on Moodle for download. 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 task1a.c coursework.c`"), and include the `coursework.h` file in your code (using `#include "coursework.h"`).

In addition, there are sample outputs for each one of the requirements on Moodle. Your code should generate outputs that look similar to those examples, but note your output may differ due to non-deterministic behaviour of processes.

Requirements

Task 1: Process scheduling

This task focusses on the implementation of two elementary process scheduling algorithms, namely **shortest job first** (SJF) and **round robin** (RR). You are asked to implement both algorithms in separate source files (`task1a.c` for SJF, `task1b.c` for RR) using linked lists. In the case of SJF, your implementation should **insert new jobs into the existing list of processes (i.e. the ready queue) at the correct location** (in increasing order of burst time). In the case of RR, new processes should be **appended to the end of the list** (you should maintain both the head and tail of the linked list for an efficient implementation in this case). Both implementations should **separate the manipulation of the linked list from the execution of the processes** (that is, you are expected to define separate "`add`" and "`remove`" methods). Note that there is no need to implement the event queues for this requirement.

In order to make your code more realistic, a `simulateSJFProcess()` and a `simulateRoundRobinProcess()` function is 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.

A successful implementation for task1a and task1b should include:

- Code to generate a pre-defined number of processes and store them in a linked list corresponding to the queue, ordered by burst time in the case of SJF, and in FCFS in the case of RR.
- A correct implementation of both the SJF and the RR algorithms.
- Correct logic to calculate the 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).
- Correct use of the appropriate `simulateProcess()` functions to simulate the processes running on the CPU.
- Correct use of dynamic memory.
- 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 50ms 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 1000 jobs and a time slice of 5ms.

Call your code "`task1a.c`" (SJF) and "`task1b.c`" (RR) and the outputs "`task1a.txt`" (SJF) and "`task1b.txt`" (RR). **Please stick rigorously to the naming conventions, including capitalisation etc.**

Task 2: SJF with Bounded Buffer

In task 1, it is assumed that all processes are available upon start-up. This is usually not the case in practice, nor can it be assumed that an infinite number of processes can simultaneously co-exist in an operating system (which typically have an upper limit on the number of processes that can exist at any one time, determined by the size of the process table). You are therefore asked to implement the SJF algorithm from task 1a using a bounded buffer (of which the size models the maximum number of processes that can co-exist). The bounded buffer must be implemented as a **linked list ordered by burst time and** with a maximum **capacity of N elements** (where N equal to the maximum number of processes and defined by `BUFFER_SIZE` in the header file `coursework.h`). Since both the producer (which creates the processes) and the consumer (which removes the processes and "runs" them by calling `simulateSJFProcess()`) manipulate the same data structure/variables, synchronisation

will be required. You are free to choose how you implement this synchronisation, but it must be as efficient as possible. A correct implementation of task 2 must:

- Contain a **producer thread** that generates processes and places them in the buffer of ready processes whenever there is space available; a **consumer thread** that removes processes from the ready list and runs them (i.e. calls `simulateSJFProcess()`). The producer can only add processes to the buffer if spaces are available, the consumer can only remove processes from the buffer if processes are available. The producer and consumer must run in separate threads and with maximum parallelism.
- Correctly use the `simulateSJFProcess()` function and correctly calculate (and display) the average response/turnaround times.
- Clearly separate the implementation of the linked lists (adding/removing elements) from the implementation of the process scheduling algorithm (i.e. do not place everything in one single method).
- Declare, initialise, and utilise all semaphores/mutexes correctly. The type of synchronisation method should be considered carefully to ensure maximum performance of your code.
- The synchronisation of your code must be correct, as efficient as possible, and no deadlocks should occur.
- Join the producer and consumer with the main thread in a correct manner, ensuring that the main thread cannot finish before the producer and consumer are both finished.
- Generate output similar to the sample file provided on Moodle for requirement 2, but for 1000 jobs.

Call your code `task2.c` and the output generated with your code for 1000 "processes" `task2.txt`. **Please stick rigorously to the naming conventions, including capitalisation etc.**

Task 3: SJF with Bounded Buffer and Multiple Consumers

You are asked to extend the code from task 2 to use multiple consumers, with each consumer modelling a different CPU/core. In addition to the requirements for task 2 (which are still applicable), a correct implementation must:

- Make use of the pre-defined constant `NUMBER_OF_CONSUMERS` in the header file `coursework.h` to create a configurable (but pre-defined) number of consumers.
- Correctly join the producer and all consumer threads with the main thread, ensuring that the main thread cannot finish before the producer and all consumers have **gracefully ended** (i.e. consumers should not be blocked on semaphores/mutexes).
- Assign a unique Id to every consumer, used in the output of your code to indicate which consumer is running which job at what time.
- Generate output similar to the sample file provided on Moodle for requirement 2, but for 1000 jobs.

Call your code `task3.c` and the output generated with your code for 1000 "processes" `task3.txt`. **Please stick rigorously to the naming conventions, including capitalisation etc.**

Task 4: RR with Bounded Buffer and Multiple Consumers

You are asked to integrate the Round Robin algorithm from task 1b with a bounded buffer. In addition to the requirements under task 2 and 3, a successful implementation of this task must:

- Correctly use the `simulateRoundRobinProcess()` function and correctly calculate (and display) the average response/turnaround times.
- Ensure that all threads end gracefully once all "processes" have completed.
- Not deadlock.

- Generate output similar to the sample file provided on Moodle for requirement 4, but for 1000 jobs and a time slice of 5ms.

Call your code `task4.c` and the output generated with your code for 1000 "processes" and a time slice of 5ms `task4.txt`. **Please stick rigorously to the naming conventions, including capitalisation etc.**

Task 5: Blocking RR with Bounded Buffer and Multiple Consumers

You are asked to extend the code for requirement 4 to include event queues. The `coursework.c` file contains a `simulateBlockingRoundRobinProcess()` that should be used for this task. This function simulates a round robin process with a probabilistic possibility of blocking. If the process blocks, the process state is updated by setting the event-type-ID field in the respective process data structure to the id of the event that was generated (the event Ids are generated at random in the `simulateBlockingRoundRobinProcess()` function). Upon termination of this function, the state of the process is checked by the consumers and dealt with accordingly.

You may assume that **the event queues are unbounded**, but the total number of processes in the system should not exceed N. The unblocking of processes should be simulated using a separate "event manager" thread that periodically checks the event queues (e.g. every 20ms) and unblocks the first process in each of the queues (if one is available). The unblocked processes should be placed at the end of the ready queue by the event manager. The event queues themselves should be implemented as a FCFS linked list. I.e., new jobs are added at the end. A correct implementation of your code must meet the requirements from task 2, 3, and 4 above, which are still relevant. In addition to these requirements, a correct implementation of task 5 must:

- Correctly use the `simulateBlockingRoundRobinProcess()` function and correctly calculate (and display) the average response/turnaround times.
- Correctly define a separate `eventManager` thread that runs independently and in parallel with the producer and consumers.
- Correctly and efficiently synchronise the different queues without possibilities for deadlocks and with maximum parallelism.
- Correctly join the event manager thread with the main thread to ensure that the main thread only terminates when all separate threads have terminated.
- Correctly process the different process states.
- Be efficient with minimal unnecessary duplication of code.

Call your code `task5.c` and the output generated with your code for 1000 "processes" and a time slice of 5ms `task5.txt`. **Please stick rigorously to the naming conventions, including capitalisation etc.**

Tip:

Note that if separating the manipulation of the linked lists from the implementation of the process scheduling, the add and remove methods would only have to be defined once if "pointers to pointers" are used as the function parameters. Plenty of tutorials and examples are available online on how to use pointers to pointers.

Submission:

Create a single `.zip` file containing all your source code and output files in **one single directory** (i.e., **zip up the directory** rather than the individual files, and use your **username as the name of the directory,** e.g. psyXXX). If you have completed all parts of the coursework successfully, the directory should contain:

- task1a.c, task1a.txt, task1b.c, task1b.txt
- task2.c, task2.txt
- task3.c, task3.txt
- task4.c, task4.txt
- task5.c, task5.txt

The deadline for submission is **3pm, Friday the 1th of December.**

Marking criteria:

The focus on this coursework is on operating systems and concurrency, e.g. the use of operating system APIs and the correct synchronisation of your code. It is not on programming skills i.e., you will be mainly assessed on these concepts, not on the extra bells and whistles you have included in your code. In fact, they often make it more difficult to analyse the solution, and hence over complicating your code may actually be detrimental.

In each case, the marking will take into account how you have met the requirements for each of the tasks, your exact implementation, the correctness of the algorithms/logic, and the efficiency of your code (in particular with respect to the synchronisation). To assess these criteria, we will analyse, compile and run your submitted code and look for specific inefficiencies. Note that your coursework will not be marked automatically, and that we will examine and provide feedback for each of the individual files.

When considering how this will be marked, please be aware that it is important that:

A) We can compile and run your program **on the school's servers** without modification (i.e. if programming on an Apple device, please verify that your code compiles and runs on the school's servers).
B) Your program does not crash, go into deadlock, or cause segmentation faults (by addressing unallocated memory)
C) We can understand the code that you have written. I.e., you did not overcomplicate your code and have added sufficient documentation to enable us to understand your code.
D) We understand why you wrote the code in the way that you did. Again use the documentation to explain this.
E) Your code/logic is correct and that synchronisation is appropriate.
F) Your code does not unnecessarily complicated or inefficient.
G) You have demonstrated your understanding of threading, synchronisation, and the use of operating system APIs.

Note that not all requirements will receive an equal proportion of the marks.