COMP2007 - OSC: Coursework Specification

Dr. Geert De Maere
Dr. Alexander Turner
School of Computer Science
University of Nottingham

October 2022

1 Introduction

1.1 Overview

The goal of this coursework is to make use of operating system APIs (specifically, the POSIX API in Linux) and concurrency directives to implement a process and I/O simulator. A successful implementation will have multiple process queues (which use the principle of bounded buffers), a simulation of multiple CPUs, a readers-writers implementation with fairness for multiple devices, and a disk scheduler. The final simulator will have some principles embedded in it that you would find in common operating systems.

Completing all tasks will give you a good understanding of:

- The use of (some) operating system APIs.
- Implementations of process tables, process queues, and disk scheduling algorithms.
- The basics of concurrent/parallel programming using operating system functionality.
- Critical sections, semaphores, mutexes, and mutual exclusion.
- Bounded buffers and readers-writers.
- C-programming.

That is, successfully implementing the coursework will be a key step towards the learning outcomes for this module.

To maximise your chances of completing this coursework successfully (and to give you the best chance of getting a good grade), it is recommend that you break it down in the different stages listed in Section 3. Each step gradually adds more complexity and builds up key insights. Only the final version of your code (which will include all components of the previous versions) should be submitted in Moodle, and only the submitted version will be marked.

1.2 Coding

1.2.1 Servers

Tutorials on how to log on to the servers are available in the Lab section on the Moodle page. In short, when off campus:

- From Windows:
  - Set up an ssh tunnel using: `plink -N -L 2201:bann.cs.nott.ac.uk:22 -P 2222 <username>@canal.cs.nott.ac.uk`.
  - When using WinSCP or Putty, make sure to specify `localhost` for the hostname and port 2201 to connect to.
- From Mac:
  - For ssh use: `ssh -J <username>@canal.cs.nott.ac.uk:2222 <username>@bann`
  - For scp use: `scp -J <username>@canal.cs.nott.ac.uk:2222 file.c <username>@bann:` (this will copy the file to your root home directory)

Note that, if writing your code on the machines in the labs, the `H:` drive where you store your code is shared with the servers. That is, any code written in, e.g., Notepad++ or Visual Studio stored on the `H:` drive will be automatically visible on `bann`, and can be compiled there using an ssh connection.

Important: You must test that you are able to connect to the servers from home (using an ssh tunnel) early in the term so that any problems can be resolved at the start of term. Not being able to connect to the servers will not be a valid ground for ECs (unless the servers themselves were to fail).

1.2.2 GNU C-Compiler

Your code MUST compile and run on the School's servers (e.g., `bann.cs.nott.ac.uk`). Your submission will be tested and marked on these machines, and we cannot account for potential differences between, e.g., Apple and Linux users.

You can compile your code with the GNU C-compiler using the command `gcc -std=gnu99 <sources.c>`, adding any additional source files (e.g. `linkedlist.c` or `coursework.c`) and libraries to the end of the command. For instance, if you would like to use threads, you will have to use (on the command line):

```
gcc -std=gnu99 <sources.c> -lpthread
```

1.2.3 GNU Debugger

Code on the servers can be debugged using the GNU debugger from the command line (gdb). Tutorials on how to use the debugger are available, for instance, here: https://www.cs.cmu.edu/~gilpin/tutorial/.

1.2.4 Git

A Git repository to which you have been added - and that you must use - was created for your coursework.

Important: Under no circumstances should your code be stored in a publicly accessible repository, also after formally submitting your work.

1.3 Additional Resources

- 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. It is our understanding that

this book was published freely online by the authors and that there are therefore no copyright violations.

- Additional information on operating system data structures and concurrency / parallel programming can be found in, e.g.:
  - Tanenbaum, Andrew S. 2014 Modern Operating Systems. 4th ed. Prentice Hall Press, Upper Saddle River, NJ, USA.
  - Silberschatz, Abraham, Peter Baer Galvin, Greg Gagne. 2008. Operating System Concepts. 8th ed. Wiley Publishing.
  - Stallings, William. 2008. Operating Systems: Internals and Design Principles. 6th ed. Prentice Hall Press, Upper Saddle River, NJ, USA.

Two key resources - **that you must use in your implementation** - are available in Moodle to help you:

- Functions to simulate processes and I/O access, definitions of key data structures, and definitions of constants (`coursework.h` and `coursework.c`)
- A generic implementation of a linked list (`linkedlist.h` and `linkedlist.c`) .

1.4 Workload

This coursework counts towards 50% of a 20 credit module. That is, at 10 hours per credit, you should expect to take about 100 hours to complete it (the equivalent of 2.5 weeks full-time work).

1.5 Submission

Your coursework must be submitted in Moodle. The submission system is set up such that you can submit as many times as you like. Any previous submission will be overwritten automatically when re-submitting.

Important: Late submissions are not allowed. That is, the submission system closes automatically at 15:00 on the day of the deadline. It is therefore strongly recommended that you submit your coursework early and regularly.

1.6 Getting Help

You may ask Dr. Geert De Maere or Dr. Alexander Turner – through the coursework forum in Moodle – for help on understanding the coursework requirements if they are unclear (i.e. what you need to achieve). Clarification will then be added to the forum so that everyone has access to the same information. You may NOT get help from anybody to do the coursework (i.e. how to do it), including ourselves or the teaching assistants.

1.7 Plagiarism

You may freely copy and adapt any code samples provided in the lab exercises or lectures. You may freely copy code samples from the Linux/POSIX websites, which have many examples explaining how to do specific tasks. This coursework assumes that you will do so and doing so is 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.

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, then you are attempting to pass someone else's work off as your own and this is plagiarism. The university takes plagiarism extremely serious and this can result in getting 0 for the coursework, the entire module, and potentially much worse. Note that all code submitted will be checked for plagiarism, and any potential cases will be followed up on through formal academic misconduct meetings.

1.8 Copyright

This coursework specification has an implicit copyright associated with it. That is, it must not be shared publicly without written consent from the authors. Doing so - without consent - could result in legal action.

2 Requirements

2.1 Overview

A full implementation of this coursework will contain the following key components, all implemented as threads:

- A process generator.
- Process simulators.
- A readers-writers implementation with fairness for multiple devices.
- A disk scheduler.
- A process terminator.

Each of these components is described in more detail in Section 2.2. In addition, the following data structures will be required:

- A pool of available PIDs.
- A process table.
- Ready queues, implemented as linked lists.
- I/O Queues, implemented as linked lists.
- A terminated queue, implemented as a linked list.

The architecture for a full implementation of the coursework, and the interaction between the different components, is shown in Figure 1.

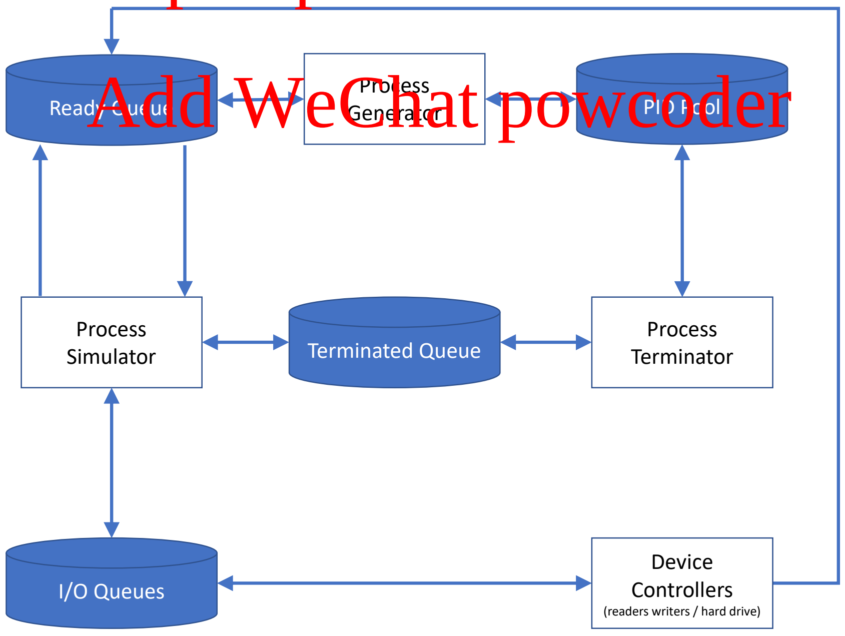

Figure 1: System Architecture

2.2 Components

This section describes the functionality of the individual components in a full implementation of this coursework.

2.2.1 The Process Generator

The process generator creates a pre-defined number of processes and adds them to the process table (indexed by `PID`) and relevant ready queue. The maximum number of processes in the system, at any one point in time, is restricted to `MAX_CONCURRENT_ PROCESSES`. The process generator goes to sleep when this maximum is reached, and is woken up when space becomes available (e.g., when a process has finished and is removed by the process terminator).

2.3 Process Simulator

The process simulators remove processes from the ready queues and simulate them in a preemptive round robin fashion using the `runPreemptiveProcess()` function. The simulators must be implemented such that fairness between I/O and CPU bound processes is maintained. If the `runPreemptiveProcess()` function returns a process in:

- the `READY` state, it is re-added to the appropriate ready queue.
- the `TERMINATED` state, it is added to the terminated queue.
- the `BLOCKED` state, it is added to the appropriate I/O queue.

2.4 I/O Simulators

2.4.1 Hard Drive

The hard drive simulator emulates read-write operations for a traditional rotational drive using a "look scan" algorithm. The disk simulator is woken up when new requests are added to its queue, and goes dormant when its queue is empty (i.e., all currently available requests have been processed). Once all processes have finished, the disk simulator thread ends gracefully.

2.4.2 Readers-Writers

The reader-writer implementation for multiple devices enables parallel reading and serialised writing. The reader-writer removes processes blocked on I/O from its associated device queue and processes them. To maximise fairness, requests are processed in the order in which they show up (whilst allowing parallel readers and serialised writers). To clarify, read `(R) and write (\verbW—)` requests arriving in the order `RRR W RRRRR WW RRR` are processed as:

- three readers in parallel,
- a single writer,
- five readers in parallel,
- two writers sequentially,
- three readers in parallel.

Note that the implementation should accommodate multiple - simulated - I/O devices, each one having their own readers and writers, and their own own associated queues. The number of readers and writers per device is configurable through the `NUMBER_OF_READERS` and `NUMBER_OF_WRITERS` constants in the `coursework.h` header file, as is the number of I/O devices.

2.5 Process Terminator

The process terminator removes finished processes from the system, frees up associated resources (e.g., page table entry, PID, memory), and prints the process' turnaround and response times on the screen. The "terminator" is woken up when processes are added to its queue, and goes to sleep when the queue is empty. Once all processes have terminated, the "terminator" prints the the average response, average turnaround, and average number of tracks crossed on the screen.

2.6 Output Sample

To track progress of the simulation, progress messages are printed on the screen. A sample of a successful implementation in which hard drive requests are processed in FCFS is available for download from Moodle. The output generated by your code should match the syntax of the sample provided. Numeric values can of course differ due to non-deterministic nature of multi-threaded code.

3 Breakdown

To make this coursework as accessible as possible, it is recommended to approach it in the steps outlined below. The first part focuses on the development of a simulation framework for process scheduling, the second part on the implementation of "device controllers" for I/O simulation.

Recall from above that you **must use** the functions and data structures defined in the files provided in Moodle (`coursework.h`, `coursework.c`, `linkedlist.h` and `linkedlist.c`), and that only the final version of your code should be submitted in Moodle for marking.

3.1 Part 1 – Process Simulation

This part does not simulate I/O. That is, the `runPreemptiveProcess()` function - used to simulate processes running on the CPU - should be called with the second and third parameter set to false (or 0).

3.1.1 Simulation of a Single Process

In the main function of your code, create a single process using the `generateProcess()` function and simulate it running in a round robin fashion using the `runPreemptiveProcess()` function. Note that the `generateProcess()` function returns an initialised "process control block" that is stored in dynamic memory.

Save your code as `simulator1.c`. The output generated by your code should be written to the screen and match the syntax of the output sample provided in Moodle.

3.1.2 Simulation of Multiple Processes

In the main function of your code, create a predefined number of processes (`NUMBER_OF_PROCESSES`) and add them to a ready queue (implemented as a linked list). Once all processes have been generated, simulate them in a round robin fashion using the `runPreemptiveProcess()` function provided. Processes returned in the `READY` state are re-added to the tail of the ready queue. Processes returned in the `TERMINATED` state are added to the tail of the terminated queue. Once all processes have finished running, remove them from the terminated queue one by one and free any associated resources.

Tip: note that a macro to initialise a linked list structure is provided and can be used as:
`LinkedList oProcessQueue = LINKED_LIST_INITIALIZER`.

Save your code as `simulator2.c`. The output generated by your code should be written to the screen and match the syntax of the output sample provided in Moodle.

3.1.3 Parallelism

Single Process Simulator

This step introduces parallelism into your code by implementing the process generator, process simulator and process terminator as threads. The process generator adds processes to the ready queue, goes to sleep when there are `MAX_CONCURRENT_PROCESSES` in the system, and is woken up when space for new processes becomes available again. The process simulator removes processes from the ready queues and runs them in a round robin fashion using the `runPreemptiveProcess()` function. Processes returned in the `READY` state are re-added to the ready queue, processes returned in the `TERMINATED` state are added to the terminated queue, and the process terminator is woken up. The simulator finishes when all processes have finished.

Save your code as `simulator3.c`. The output generated by your code should be written to the screen and match the syntax of the output sample provided in Moodle.

Parallel Process Simulators

Extend the code above to have a configurable number of process simulators. Note that all simulators must terminate gracefully once all processes have finished.

Save your code as `simulator4.c`. The output generated by your code should be written to the screen and match the syntax of the output sample provided in Moodle.

3.1.4 Process Table

Add a process table to your code, implemented as an array of size `SIZE_OF_PROCESS_TABLE` and indexed by PID. The process generator is now responsible for adding new processes to the process table, in addition to the ready queue. The termination daemon is now also required to remove finished processes from the process table.

Note that the implementation above requires to add a "pool" (implemented as an array) of available PIDs. That is, when a process is created using the `generateProcess()` function, a PID is removed from the pool. When it is cleaned up (by the process terminator), the PID is re-added to the pool. Although this can be implemented using counting semaphores, your implementation must use **binary semaphores** only to manage the PID pool.

Save your code as `simulator5a.c`. The output generated by your code should be written to the screen and match the syntax of the output sample provided in Moodle.

3.2 Part 2 – I/O Simulation

This part simulates I/O operations. It distinguishes between emulating I/O controllers for traditional rotational hard drives, and an implementation of a reader-writer approach for a generic device. The `runPreemptiveProcess()` function in this part should be called with the second and third parameter set to true (or 1) as appropriate, and processes can now also be returned in the `BLOCKED` state. Note that the the device number and device type in the process control block enable you to identify the type of I/O request and to which device it applies.

3.2.1 Hard Drive Access: FCFS

Add an appropriate queuing structure to your code to simulate disk access. Disk requests are added to the queue when they become available, and processed in FCFS order. Similar to, e.g., the process terminator, the "disk controller" goes to sleep when there are no requests available, and is woken up when new requests are added. To emulate disk access times, the `simulateIO()` function should be called as requests are processed. Once the I/O request for a given process is dealt with, the process enters the `READY` state and is added to the appropriate ready queue. Your implementation must ensure fairness between I/O and CPU bound processes.

Save your code as `simulator5b.c`. The output generated by your code should be written to the screen and match the syntax of the output sample provided in Moodle.

3.2.2 Hard Drive Access: LOOK-SCAN

Modify the code above to implement a LOOK-SCAN algorithm.

Save your code as `simulator6.c`. The output generated by your code should be written to the screen and match the syntax of the output sample provided in Moodle.

3.2.3 Generic I/O Device

Single Device

Add a "device controller" and an appropriate queue structure to your code to simulate a single generic I/O device. Requests must be processed using a reader-writer approach with fairness (see above), allowing for parallel leading and serialised writing. Call the `simulateIO()` function to emulate I/O access times.

Save your code as `simulator7.c`. The output generated by your code should be written to the screen and match the syntax of the output sample provided in Moodle.

Multiple Devices

Extend the code above to handle multiple independent I/O devices. Every device should have its own queue structure and act independent of one another. For instance, read and write requests on different devices can happen with full parallelism, however, write requests on the same device are serialised.

Save your code as `simulator8.c`. The output generated by your code should be written to the screen and match the syntax of the output sample provided in Moodle.