COMP2007 - OSC: Coursework Specification

Dr. Geert De Maere
Dr. Dan Marsden
School of Computer Science
University of Nottingham

October 2023

1 Introduction

1.1 Overview

The goal of this coursework is to make use of operating system APIs, particularly the POSIX API, to implement a process simulator on Linux. Your final implementation will be based upon principles that you would find in modern operating systems, and will exploit some standard concurrency techniques.

Completing all the tasks will give you a good understanding of:

- Key concepts in operating systems.
- The use of operating system APIs.
- Implementations of process tables and process queues.
- The basics of concurrent/parallel programming using operating system functionality.
- Critical sections, semaphores, mutexes, and mutual exclusion.
- Bounded buffers.
- C programming.

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 stages, should be submitted in Moodle. Only this 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. 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 iOS/Linux:
  - 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 `file.c` to your root home directory.

The `H:` drive where you store your code is shared with the servers. Any code written in an editor such as Notepad++ or Visual Studio, and 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. Not being able to connect to the servers will not be a valid ground for ECs.

1.2.2 GNU C-Compiler

Your code **must** compile and run on `bann.cs.nott.ac.uk`. Your submission will be tested and marked on this machine, and we cannot account for potential differences with other configurations.

You can compile your code with the GNU C-compiler using the command `gcc -std=gnu99 <sources>`, where `<sources>` is a list of C source files. For example:

```
gcc -std=gnu99 file1.c file2.c
```

compiles an executable called `a.out` from two source files. If you are using pthreads, you must add the flag `-lpthread` when compiling. For example:

```
gcc -std=gnu99 file.c -lpthread
```

compiles an executable called `a.out` from one source file which might use the pthread library. If you want to specify the name of the executable file, extend your compile command with `-o <output>` where `<output>` is your choice of output file name. For example:

```
gcc -std=gnu99 file.c -o prog -lpthread
```

compiles an executable called `prog` from one source file which might use the pthread library.

1.2.3 GNU Debugger

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

1.2.4 Git

A Git repository was created for your coursework, and you should have received an invite to this respository. You are expected to use this repository to manage your code. **Under no circumstances should your code be stored in a publicly accessible repository, even after formally submitting your work.**

1.3 Additional Resources

- A tutorial on compiling source code in Linux using the GNU C compiler can be found on the course Moodle page.
- 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., in the *"Advanced Linux Programming"* book by Richard Esplin.
- Information on operating system data structures and concurrency / parallel programming can be found in:
  - 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:

- Code to simulate processes and I/O, and definitions of key data structures constants are provided in the files `coursework.h` and `coursework.c`.
- A generic implementation of a linked list is provided in the files `linkedlist.h` and `linkedlist.c` .

1.4 Workload

This coursework counts towards 50% of a 20 credit module. At 10 hours per credit, you should expect to take about 100 hours to complete it. This is 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. The submission system closes automatically at 15:00 on the day of the deadline. It is strongly recommended that you submit your coursework early and regularly to avoid last-minute difficulties.

1.6 Getting Help

You may ask Dr. Geert De Maere or Dr. Dan Marsden for help on understanding the coursework objectives if they are unclear, preferably during the OSC Lab. You may **not** get help from anybody

to do the coursework, including ourselves or the teaching assistants.

1.7 Academic Misconduct

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. 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 academic misconduct.

The coursework itself is an individual task. **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 academic misconduct. The university takes academic misconduct 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 collusion and plagiarism, and any potential cases will be followed up on through formal academic misconduct meetings.

1.8 FAQ

- *I get a message stating "file system quota exceeded", and am unable to log on*: this means that you have stored too many files on the school's file systems, for which the limit is usually set to 700MB. You may no longer be able to log on from Windows, since temporary files created when logging on cannot be stored under your home directory due to the lack of space. The best solution is to log on to the Linux servers, execute the command `du -h  sort -h—` from the command line in your home directory. This will list all the files you have stored in reverse order for size. Check and remove appropriate files, using `rm <filename>`.

1.9 Copyright

2 Requirements

2.1 Overview

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

A process generator: This thread will simulate an environment in which users are creating processes to do work. To keep things simple, it will also place these processes in the relevant data structures, rather than explicitly asking a kernel to do that work.

A Process simulator: - This thread simulates the scheduler running processes on the hardware. The simulation will be done via two provided procedures:

1. `runNonPreemptiveProcess` - This procedure should be called to simulate running a process that cannot be preempted. When it returns, the process will either be completed or blocked on I/O.
2. `runPreemptiveProcess` - This procedure should be called for processes that can be preempted, such as those running in a time sliced round robin. When it returns the process will either be completed, blocked on I/O, or ready as it was preempted and can run again.

A booster daemon: A thread that will periodically boost the priority of processes running on the system in a naive manner to prevent resource starvation.

A CPU load balancer: This thread will manage load balancing by migrating processes between CPUs.

An I/O daemon This simulates I/O operations completing in a simple manner by periodically moving processes that are blocking on I/O back to the ready state.

A process terminator: A thread responsible for cleaning up terminated processes, and reporting related statistics.

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
- A set/sets of ready queues, implemented as linked lists
- A set of I/O queues, each one implemented as a linked list
- 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.

2.2 Components

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

2.2.1 Process Generator

The process generator creates a predefined number of processes and adds them to the process table and relevant ready queues. The process table is indexed by `PID` and the maximum number of processes

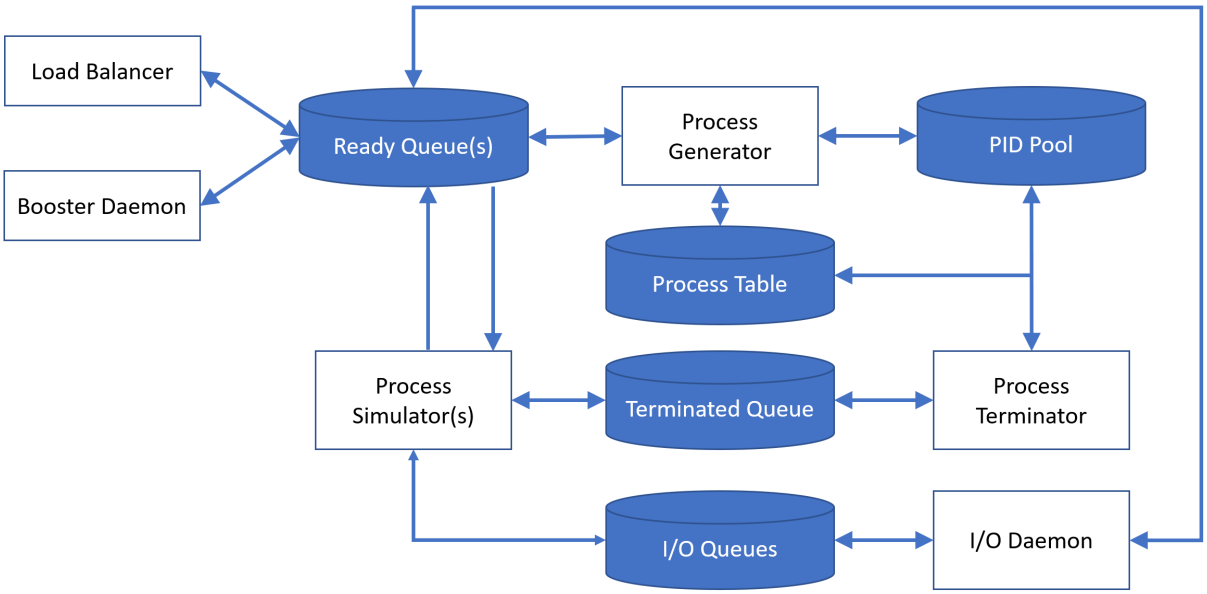

Figure 1: System Architecture

it can take, and hence the maximum number of processes that can be in the system at any one point in time, is restricted to `MAX_CONCURRENT_PROCESSES`. The process generator goes to sleep when the maximum for the number of processes currently in the system is reached and is woken up when space becomes available, e.g., due to an earlier process finishing and being removed from the system by the process terminator.

An efficient implementation will require you to use a pool for the available `PIDs`. This can be implemented as an array of integers, from which a process identifier is taken and removed when a process is created, and re-added when the process terminates. This is more efficient compared to searching through the process table to find the next available position and `PID`.

2.2.2 Process Simulators

The process simulators remove processes from the ready queues and simulate them. The implementations for the initial steps recommended in Section 3 use a single priority level, the later steps require multiple priority levels. It is assumed that lower numeric values represent higher priority, as was the convention in lectures.

In the later steps, when working with multiple priority levels, the upper half of the priority levels must be simulated in FCFS, and the `runNonPreemptiveProcess()` function in the `coursework.c` file must be called. The lower half of the priority levels run in a round robin fashion. This can be achieved using the `runPreemptiveProcess()` function.

If the `runNonPreemptiveProcess()` or `runPreemptiveProcess()` function returns a process in:

- the `BLOCKED` state, it is added to the appropriate I/O queue based on the `deviceID` in the `process struct`.
- the `TERMINATED` state, it is added to the terminated queue.

If the `runPreemptiveProcess()` returns a process in the `READY` state, it is re-added to the appropriate

ready queue.

Finally, the order in which processes run must respect fairness between I/O and CPU bound processes. That is, any process that previously blocked on I/O, and hence has not used its entire time slice, should receive priority over other comparable processes at the same priority level.

2.2.3 I/O Daemon

The I/O daemon runs periodically, and checks the I/O queues for processes blocked on I/O. The daemon removes processes from the I/O queues, and re-adds them to the relevant ready queue.

There is one I/O queue corresponding to each device. The device on which a process is blocked, and hence which I/O queue it is in, is determined by the `deviceID` in the `process struct`.

2.2.4 Booster Daemon

The booster daemon periodically increases the priority of round robin jobs to the highest round robin level, e.g. level 8 in the case of 16 priority levels. This prevents starvation, improves response times and can, in real operating systems, help to prevent deadlocks.

2.2.5 Load Balancer

The load balancer runs periodically, and monitors the average CPU load. The latter is, in the case of this coursework, approximated as the average response time for the last 20 processes on each CPU. If the CPU load is unbalanced, the load balancer selects a random process from the CPU with the highest load, and adds it to the ready queues for the CPU with the lowest load. This assumes that every CPU has its own set of private ready queues, which is the case for the final steps of this coursework described in Section 3.

2.2.6 Process Terminator

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

2.3 Output Samples

To track progress of the simulation, progress messages are printed on the screen. A sample of a successful implementation is available for download from Moodle, and 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. 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 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. The memory is cleared when the `destroyProcess()` function is called.

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.

3.2 Simulation of Multiple Processes

In the main function of your code, create a pre-defined number of processes (`NUMBER_OF_PROCESSES`) and add them to a ready queue implemented as a linked list (using the implementation provided). 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.

3.3 Parallelism - Single CPU

This step introduces parallelism into your code by implementing the process generation, process simulation and process termination 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 as soon as a new process can be added to the system. 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 that are returned in the `TERMINATED` state are added to the terminated queue, after which the process terminator is woken up. The simulator finishes when all processes have been simulated and 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.

3.4 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 terminator is now also required to remove finished processes from the process table. Note that an efficient implementation of the above above requires to add a "pool"

of PIDs, as described in Section 2.2.1. The process generator removes PIDs from the pool, the process terminator adds them again.

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.

3.5 Process Priorities

Extend the code above to account for different priority level. The maximum number of levels is defined by `NUMBER_OF_PRIORITY_LEVELS`. The upper half (`[0, NUMBER_OF_PRIORITY_LEVELS[`) is simulated in a FCFS fashion using the `runNonPreemptiveProcess()` function. The lower half (`[NUMBER_OF_PRIORITY_LEVELS / 2, NUMBER_OF_PRIORITY_LEVELS[`) runs in a round robin fashion using the `runPreemptiveProcess()` function. Both functions can be found in the `coursework.c` file, the second parameter should be set to `false` to disable I/O simulation.

Save your code as `simulator5.c`. The output generated by your code should be written to the screen and match the syntax of the output sample provided in Moodle.

3.6 Booster Daemon

Implement a booster daemon that runs at regular intervals (defined by `BOOST_INTERVAL`). The booster increases the priority of round robin jobs at lower levels periodically to the highest round robin level. This can help to improves response times, prevent starvation, and prevent potential deadlocks. Note that recent Windows schedulers use a similar approach for variable jobs.

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.

3.7 I/O Simulation

This requirement adds I/O simulation. To enable this, the process simulation functions functions should be called with the second parameter set to `true`. You are required to add multiple I/O queues to your implementation, one for every device. The number of devices is determined by the `NUMBER_OF_IO_DEVICES` constant in the `coursework.h` file. The device that has "generated" the blocking call is determined by the value of `iDeviceID` in the `process struct`. Note that I/O queue to which the process must be added is determined by the value of `iDeviceID`. The "blocked" processes are removed from the I/O queues and added to the corresponding ready queues at regular intervals by the the I/O daemon. The length of the interval is determined by the value of the `IO_DAEMON_INTERVAL` parameter in the `coursework.h` file. Processes that were blocked on I/O should be given priority over similar processes when re-adding them to the ready queues.

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.

3.8 Parallelism - Multiple CPUs

Extend the code above to have `NUMBER_OF_CPUS` process simulators. Note that all simulators must terminate gracefully once all processes have been simulated. **Tip:** You may want to do a trial implementation in the code for simulator 3 first, before extending the code for simulator 7.

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.

3.9 Private Queues

Extend the code above to have private ready queues for every processor.

Save your code as `simulator9.c`. The output generated by your code should be written to the screen and match the syntax of the output sample provided in Moodle.

3.10 Load balancing

Extend the code above to implement a load balancing daemon that runs at regular intervals (determined by `LOAD_BALANCING_INTERVAL`). If the load (response time) is unbalanced, the load balancer removes a random process form busiest CPU and adds it to the queues for the least busy CPU.

Save your code as `simulator10.c`. The output generated by your code should be written to the screen and match the syntax of the output sample provided in Moodle.