

COMP2007 - OSC: Coursework Specification

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

October 2021

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 memory management system. A successful implementation will have multiple process queues (which use the principles of bounded buffers), a memory model, and two different page replacement algorithms (FIFO and Not Recently Used). Implementing both page replacement algorithms will enable you to simulate and compare their efficiency. 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 operating system APIs.
- The implementation of process tables, process queues, and page replacement algorithms.
- Critical sections, semaphores, mutexes, mutual exclusion, and bounded buffers.
- The basics of concurrent / parallel programming using an operating system's functionalities.
- C-programming.

To maximise your chances of completing the coursework successfully (and to give you the best chance to get a good grade) it is recommended that you break it down in the different stages listed in Section 3, each one of them gradually adding more complexity. Only the final version of your code should be submitted in Moodle and only one version will be marked.

1.2 Coding

1.2.1 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 and libraries to the end of the command. For instance, if you would like to use threads, you will have to specify `gcc -std=gnu99 <sources.c> -pthread` on the command line.

A video tutorial (and written tutorials) on how to access the school's servers and how to transfer files from your local machine to the remote file system is available in the Lab section on the COMP2007

Moodle page. This was also demonstrated in the first lecture on concurrency. Note that, if writing the 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.

1.2.2 GNU Debugger

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

1.2.3 Git

A Git repository to which you have been added - and that you must use - was created for your coursework. Under no circumstances should the solution for your coursework be stored in a publicly accessible repository.

1.3 Additional Information

- 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 no copyright violations because of this.
- 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.

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 Getting Help

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

1.6 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 has 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. 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, 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.

1.7 Copyright

Note that 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 (described in more detail below), all implemented as threads:

- A process generator.
- A process / memory access simulator.
- A paging daemon.
- A process terminator.

A full implementation will require the following data structures:

- A process table, implemented as a hash table and containing linked lists of process control blocks.
- A ready queue, implemented as a linked list and containing processes that are ready to run.
- A terminated queue, implemented as a linked list and containing processes that are finished.
- A page fault queue, implemented as a linked list and containing processes that have generated a page fault.
- A frame list, implemented as a linked list and containing frame entries corresponding to (simulated) blocks of memory.

Two key resources are provided (in Moodle) to help you with the implementation:

- Functions to simulate processes and memory access, definitions of key data structures, and definitions of constants (see `coursework.h` and `coursework.c`)
- A generic implementation of a linked list that **you must use** (see `linkedlist.h` and `linkedlist.c`)

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 found 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 and ready queue (where they are removed by the process simulator). To limit the load on the systems, the maximum number of processes in the system at any one point in time is capped. When the maximum number of processes is reached, the process generator goes to sleep. If space becomes available (i.e. one of the processes has finished), the generator is woken up (by the termination daemon).

2.3 Process Simulator

The process simulator removes processes from the head of the ready queue and simulates in a preemptive round robin fashion. If the process returns in:

- the READY state, it is re-added to the tail of the ready queue.

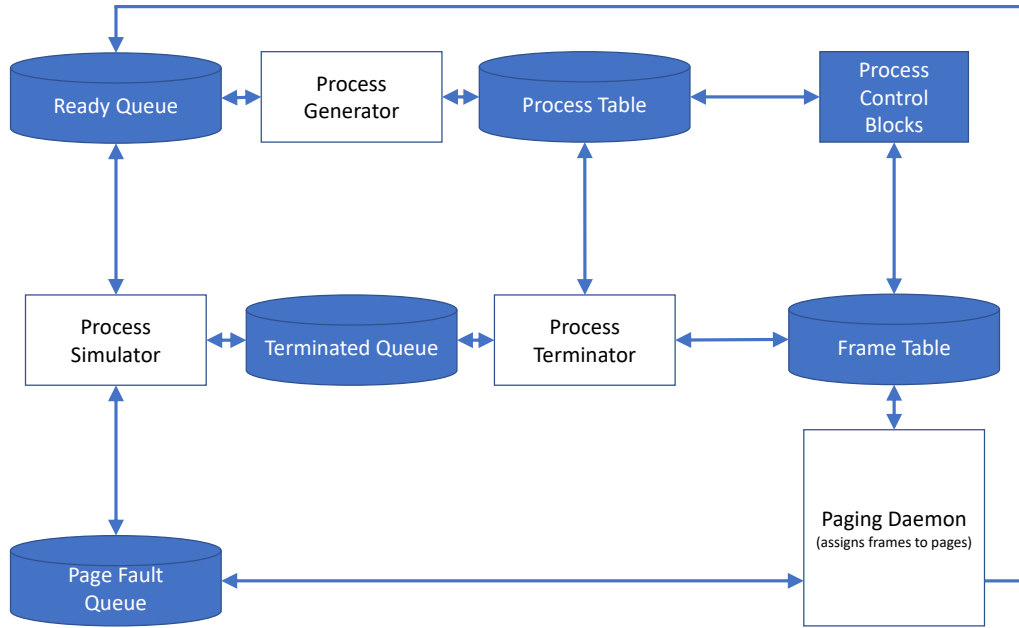


Figure 1: System Architecture

- the **TERMINATED** state, it is added to the tail of the terminated queue.
- the **PAGE_FAULTED** state, it is added to the tail of the page fault queue.

2.4 Paging Daemon

The paging daemon reclaims memory from the list of frames, assigns them to processes that have page-faulted, removes these processes from the page fault queue, updates their process state to **READY**, and re-adds them ready queue. The paging daemon implements two different page replacement algorithms:

- FIFO
- Not Recently Used

The algorithm to use in the simulation is configurable (see below). The paging daemon is woken up when new processes are added to the page fault queue, and goes to sleep when all page faults have been processed.

2.5 Termination Daemon

The termination daemon is responsible for removing processes (that have finished) from the system. That is, it removes them from the terminated queue, frees up any frames that have been mapped in the process' page table, and ensures that the respective process control block is removed from the process table. In addition, page fault statistics and response / turnaround times are printed on the screen when freeing the process' resources. Once all processes have terminated, the total and average number of page faults (across all simulated processes) is displayed. The termination daemon goes to sleep when the terminated queue is empty, and is woken up when finished processes are added.

2.6 Output Sample

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 any 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 the coursework as accessible as possible, it is recommended to approach it in the steps outlined below. The first part of this coursework focuses on the development of a simulation framework for process scheduling, the second part on the simulation of memory. Both parts are broken down into several consecutive steps. Only the final version of your code should be submitted in Moodle for marking. Note that you are expected to use the functions and data structures defined in the files provided in Moodle (`coursework.h`, `coursework.c`, `linkedlist.h` and `linkedlist.c`)

3.1 Part 1 – Process Simulation

This part does not simulate page faults. That is, the `runPreemptiveProcess()` function - used to simulate the processes running on the CPU - should be called with the second 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, 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, and that for the purpose of this coursework, it is assumed process identifiers (PID) will only every be used once.

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 pre-defined number of processes (`NUMBER_OF_PROCESSES`) and add them to the tail of a ready queue. Once all processes are 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, remove them from the terminated queue one by one and free any resources associated with them.

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

This step introduces parallelism in your code by implementing the process generator, process simulator and termination daemon as separate threads. The process generator adds processes to the ready queue and goes to sleep when there are `MAX_CONCURRENT_PROCESSES` in the system. The process simulator is woken up when new processes are added to the ready queue and simulates them in a round robin fashion. 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 terminated queue. The termination daemon is woken up when processes are added to the terminated queue. It removes them from the queue, clears up any resources associated with them, and wakes up the process generator if there are processes remaining to be generated. The simulator finishes when all processes have been simulated.

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.

3.1.4 Process Table

In this final step for part 1, a process table is introduced in your code. The process table is implemented as a hash table of which the size is defined by the `SIZE_OF_PROCESS_TABLE` constant. The hash index for a process used to access the process table is set by the `generateProcess()` function (in the process control blocks). The process generator is now responsible for adding new processes to the process table as well as the ready queue. Once a process has finished, the termination daemon also removes it from the process table.

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.

3.2 Part 2 – Memory Simulation

This part simulates page faults. That is, the `runPreemptiveProcess()` function should be called with the second parameter set to true (or 1). In addition to `READY` and `TERMINATED` states, processes can now also be returned in the `PAGE_FAULTED` state by the `runPreemptiveProcess()` function.

3.2.1 Unlimited Memory

This step adds a paging daemon (implemented as a thread) to the code from part one and assumes that there is an unlimited amount of memory available to the simulator. When a process is returned in the `PAGE_FAULTED` state, the process simulator is responsible for adding it to the page fault queue, and wakes up the page fault daemon. The paging daemon removes page faulted processes from the head of the page fault queue, assigns them a dummy frame by calling the `mapDummyFrame()` function, changes the process' state to `READY`, and appends the process to the tail of the ready queue.

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.

3.2.2 FIFO

This step adds a FIFO page replacement algorithm to the simulator. In contrast to above, a finite amount of memory / frames is assumed. This requires you to initialise a list of `NUMBER_OF_FRAMES` frames using the `PAGE_TABLE_ENTRY_INITIALIZER` macro. Note that the `iFrame` member of the `FrameEntry` structure will need setting manually.

When a process page faults, the FIFO algorithm reclaims the first available frame by calling the `reclaimFrame()` function on it. It maps the frame the page table of the process by calling the `mapFrame()` function. The process state is then changed from `PAGE_FAULTED` to `READY`, and the processes is added to the end of the ready queue. Note that the paging daemon is woken up when processes are added to the page fault queue, and goes to sleep when the queue is empty.

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 Not Recently Used

The final version of your code implements Not Recently Used paging. The paging algorithm to use in the simulation (i.e. FIFO or NRU) is configurable by setting the `PAGING` value in `coursework.h` to

either `FIFO` paging or `NRU_PAGING`. The efficiency of both approaches can be compared through the statistics printed by the termination daemon.

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.