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.
You will need knowledge 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.

it is recommended that you break it down in the different stages listed in Section 3, each one of them gradually adding more complexity.

You can compile your code with the GNU C-compiler using the command gcc -std=gnu99 , 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 -pthread on the command line.

You may freely copy code samples from the Linux/POSIX websites, which has many examples explaining how to do specific tasks.
However, you must Not copy code samples from any other source/Websites.

1 Requirements 1.1 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 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 di↵erent

components, is shown in Figure 1.

1.2 Components

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

1.1.2 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).

1.3 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.

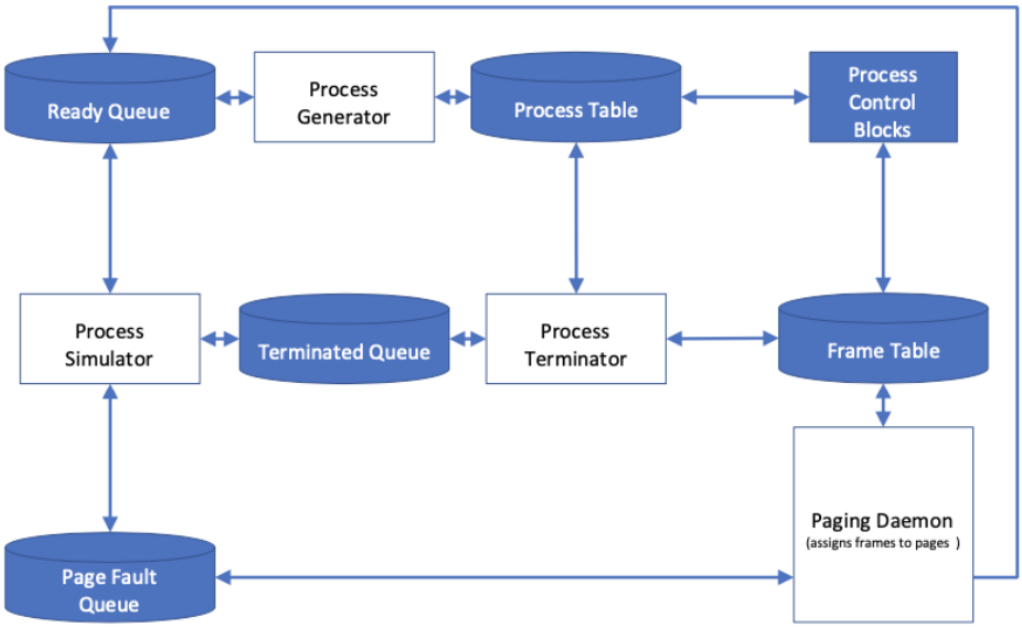

Figure 1: System Architecture

- 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.

1.4 Paging Daemon

The paging daemon reclaims memory from the list of frames, assigns them to processes that have pagefaulted, 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 di↵erent 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.

2.1.2 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.

2.1.3 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 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.

2.1.4 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.

2.2 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.

2.2.1 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.

2.2.2 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.

2.2.3 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 e ciency 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.