VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

OPERATING SYSTEMS (LAB) (CO2018)

ASSIGNMEMT SUBMISSION

REPORT OPERATING SYSTEMS ASSIGNMENT

Instructor: Le Thanh Van

Students: Phan Ngọc Lan Chi - 2352137 *(CC05 - Group 07)*
Nguyễn Mạnh Quốc Khánh - 2352525 *(CC05 - Group 07)*
Ngô Đức Tuấn - 2353269 *(CC05 - Group 07)*
Trần Anh Tuấn - 2353276 *(CC05 - Group 07)*
Vũ Quốc Việt - 2353326 *(CC05 - Group 07)*

Abstract

This assignment aims to design and implement a simplified operating system simulation that demonstrates core OS functionalities such as process scheduling, memory management, and system calls. The system utilizes a Multi-Level Queue (MLQ) scheduling algorithm to manage CPU time and employs paging-based memory management to allocate and isolate memory for multiple processes. Additionally, system calls like process termination and memory allocation are integrated to allow interaction with the operating system. The project enhances understanding of how operating systems manage resources and processes, providing a practical simulation framework for OS operations.

Keywords

Operating System, Scheduler, Memory Management, System Calls, Process Scheduling, Virtual Memory, Multi-Level Queue, Round-Robin Scheduling

Contents

List of Figures

List of Tables

Member list & Workload

No.	Fullname	Student ID	Contributions	% done
1	Nguyễn Mạnh Quốc Khánh	2352525	Report Writing (Chapter 1, 2, 3, 4)	100%
2	Phan Ngọc Lan Chi	2352137	Report Writing (Chapter 5, 6, 7, 8)	100%
3	Ngô Đức Tuấn	2353269	Scheduler Implementation (coding)	100%
4	Vũ Quốc Việt	2353326	Memory Management Implementation (coding)	100%
5	Trần Anh Tuấn	2353276	System Calls Implementation (coding)	100%

Table 1: Member list & workload

Chapter 1

Introduction

1.1 Purpose and Objective

The objective of this assignment is to design and implement a simplified operating system (OS) simulation, focusing on key functionalities such as process scheduling, memory management, and system calls, providing hands-on experience in understanding how an OS allocates CPU time, manages memory, and interacts with user programs.

1.2 Scope and System Overview

This simulation covers three core areas:

- **Scheduler**: Uses a **Multi-Level Queue (MLQ)** algorithm to manage CPU time allocation, using Round Robin and Priority.
- **Memory Management**: Allocate virtual memory to use physical memory efficiently, both RAM and SWAP, combined with **paging** for memory isolation and **swapping** to handle memory overflow.
- **System Calls**: Implements basic system calls like process termination and memory allocation, allowing interaction with the OS.

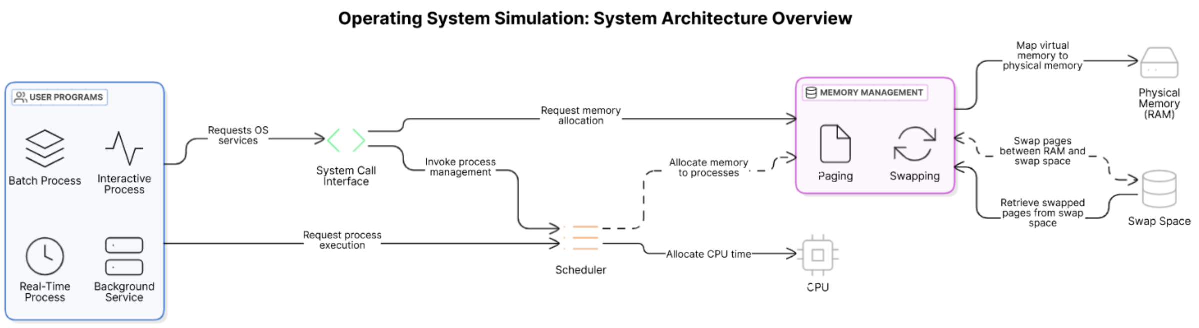
The system provides a modular design, where each component operates independently but interacts with others to simulate a complete operating environment.

Chapter 2

System Design and Architecture

This chapter describes the design and architecture of the operating system simulation. The system is composed of three key components: **Scheduler**, **Memory Management**, and **System Calls**. Each of these components interacts with the others to simulate an OS environment that efficiently manages processes, memory, and system requests.

2.1 System Architecture Overview



Operating System Simulation: System Architecture Overview
USER PROGRAMS
Batch Process
Interactive Process
Real-Time Process
Background Service
Requests OS services
System Call Interface
Request memory allocation
Invoke process management
Request process execution
Scheduler
Allocate memory to processes
Allocate CPU time
CPU
MEMORY MANAGEMENT
Paging
Swapping
Map virtual memory to physical memory
Physical Memory (RAM)
Swap pages between RAM and swap space
Retrieve swapped pages from swap space
Swap Space

Figure 2.1: System Architecture Overview

The architecture is designed to efficiently manage system resources like CPU time and memory. The key goals are to ensure **modularity**, **scalability**, and **smooth interaction** between components. The core system consists of three modules:

1. **Scheduler**: Manages CPU time allocation through the **Multi-Level Queue (MLQ)** scheduling algorithm. This algorithm prioritizes processes based on their importance and allocates CPU time using round-robin scheduling.
2. **Memory Management**: Uses **paging** to assign isolated memory regions to processes and supports **swapping** when physical memory is full.
3. **System Call Interface**: Serves as the intermediary between user programs and the OS, handling requests like process creation, termination, memory allocation, and resource management.

These components operate independently but interact seamlessly to manage processes and resources.

2.2 Key Modules

2.2.1 Scheduler (Multi-Level Queue Scheduling)

The **Scheduler** prioritizes and schedules processes for CPU execution using the **MLQ scheduling algorithm**. Processes are organized into queues based on their priority, with higher-priority processes getting more CPU time.

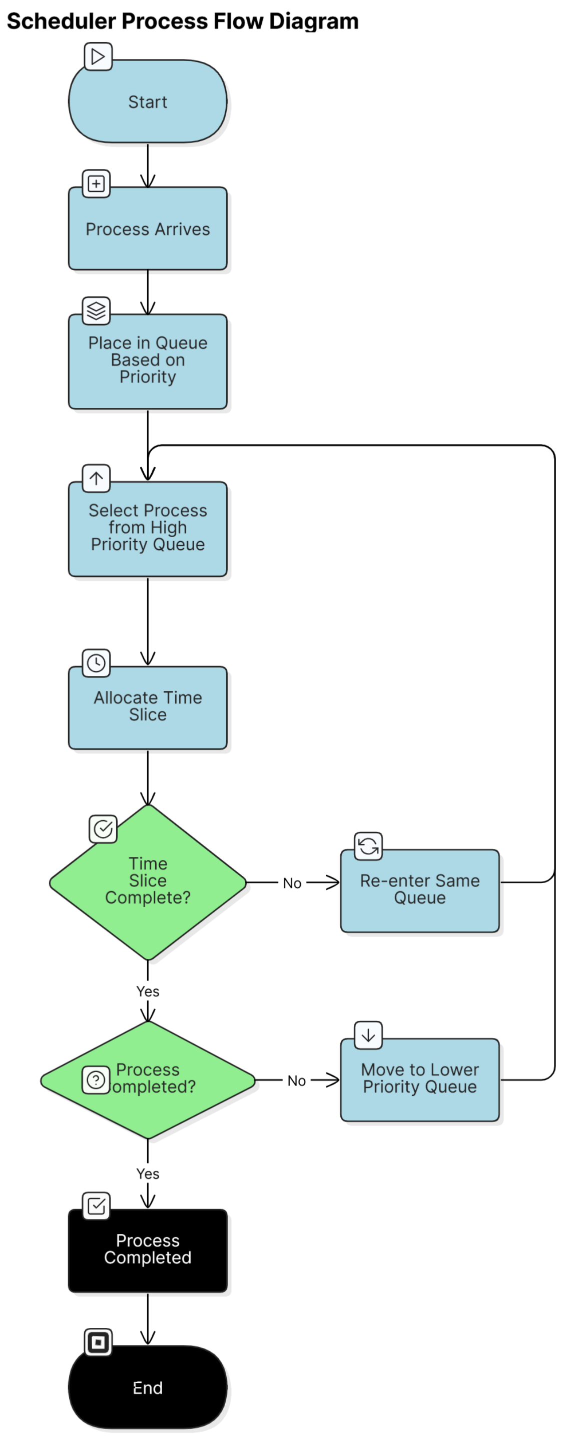

Scheduler Process Flow Diagram
Start
Process Arrives
Place in Queue Based on Priority
Select Process from High Priority Queue
Allocate Time Slice
Time Slice Complete?
No
Re-enter Same Queue
Yes
Process ompleted?
No
Move to Lower Priority Queue
Yes
Process Completed
End

Figure 2.2: Scheduler Process Flow

Process Flow:

- New processes are placed in the appropriate priority queue.
- The scheduler selects the next process with the highest priority in the MLQ ready queue.
- CPU(s) execute the process's instruction based on its Process Control Block (PCB).

- Processes either return to their original queue or are moved to a lower-priority queue after completing their time slice.

Key Benefits:

- Ensures **higher-priority processes** finish their execution before **lower-priority processes** execute.

Key scheduler functions:

- `init_scheduler()`: Initializes the system and sets up priority queues.
- `get_mlq_proc()`: Selects the next process based on priority and round-robin scheduling.
- `put_mlq_proc()`: Put (or return) a process to its queue when its time slice ends but has not yet finished.

The system uses preemptive scheduling, allocating CPU time slices based on process priority. The process is initially put in the ready queue, but after it executes and has not yet finished, it goes to another queue, the run queue.

In our designed system, we are using MLQ queue for both the ready queue and the run queue, which means that `put proc()` and `add proc()` will have the same mechanism in our design - they all redirect to `put_mlq_proc()`.

Also, to prevent the starvation if high-priority keep load into the MLQ at difference time slot, each queue will has only a fixed slot to use the CPU and when it is used up, the system must change the resource to the other process in the next queue and leave the remaining work for future slot even though it needs a completed round of ready queue.

2.2.2 Memory Management (Paging and Swapping)

Memory management allocates and isolates memory for processes using a **paging mechanism**. This system isolates each process's memory space, preventing interference between processes.

Virtual Memory and Paging

- Each process is provided with its own **virtual memory** space, which is divided into **fixed-size pages**. These virtual pages are mapped to physical memory (RAM or swap space) using a **page table**.
- **Physical memory** (RAM and swap) is divided into **fixed-size frames**. Virtual pages are mapped to these frames as needed.

Benefits of Paging

- Paging eliminates **external fragmentation** by allowing non-contiguous allocation of physical memory.
- It improves system security and stability by isolating each process's memory space.

Swapping

- When physical memory (RAM) is full, the system can **swap** pages out to secondary storage (**swap space**), making room for active pages and allowing processes to continue execution.

Memory management components

- `mm_struct`: Manages a process's virtual memory, including its page table, virtual memory areas (VMAs), and free region lists.
- `memphy_struct`: Represents a physical memory device (RAM or swap), including its storage array, size, and free frame list.

- **Page Table**: An array in each process that maps virtual pages to physical frames or swap locations.
- **Swapping Mechanism**: Handles moving pages between RAM and swap space, updating the page table accordingly.

Key Functions

- `alloc()/liballoc()`: Allocates a region of virtual memory for a process and maps it to physical frames.
- `free()/libfree()`: Frees a previously allocated virtual memory region and releases associated physical frames.
- `pg_getpage()`, `find_victim_page()`, `__mm_swap_page()`: Handle page faults, select victim pages for replacement, and perform swapping between RAM and swap space.

2.2.3 System Call Interface

The **System Call Interface** acts as the bridge between user programs and the operating system, allowing user processes to request OS-level services such as process management, memory allocation, and process termination.

System calls are managed through a **syscall table**, where each system call is assigned a unique identifier and mapped to its corresponding handler function.

Key system calls

- `killall`: Terminates all processes with a specified name (`__sys_killall()`).
- `memmap`: Allocates or frees memory regions for processes (`__sysmemmap()`).

Syscall Table

- `sys_call_table`: An array containing system call identifiers and their handler functions (`syscalltbl.lst`, `syscall.c`).
- `__sys_killall()`: Handles process termination requests.
- `__sys_memmap()`: Handles memory allocation, freeing, and mapping requests.

This interface ensures that user-level requests are safely and efficiently handled by the kernel, maintaining system stability and security.

When a process invokes a system call, the OS interacts with the appropriate subsystem (Scheduler, Memory Management) to fulfill the request.

2.3 Component Interaction

The components of the operating system interact as follows:

- **Scheduler and Memory Management**: The scheduler selects which process runs next and ensures that each process has the required memory resources. The memory management subsystem allocates and frees memory for processes, manages virtual-to-physical address translation, and handles swapping pages between RAM and swap space when physical memory is full.
- **System Call Interface**: The system call interface provides a controlled gateway for user programs to request OS services. It handles system calls such as `killall` (for process management) and `memmap` (for memory allocation and mapping), dispatching these requests to the appropriate kernel handlers.
- **OS Core**: The core of the OS coordinates all subsystems. It initializes the scheduler, memory management, and system call interface, loads processes into memory, and manages the timer system to allocate CPU time slices and support time-sharing among processes.

Chapter 3

Implementation Details

This chapter explains the detailed implementation of the key components of our operating system simulation, including the **Scheduler**, **Memory Management**, and **System Calls**. It outlines the core structures, functions, algorithms, and integration techniques used in the development of these components.

3.1 Scheduler Implementation

3.1.1 Overview of the Multi-Level Queue (MLQ) Scheduling Algorithm

The scheduler in OSSim Sierra is responsible for managing CPU time allocation among processes. We implement a **Multi-Level Queue (MLQ)** scheduling algorithm, which organizes processes into multiple priority queues. Each queue corresponds to a specific priority level, and processes in higher-priority queues are scheduled before those in lower-priority queues.

Within each queue, processes are scheduled using a round-robin mechanism: each process receives a fixed time slice (slot) before the scheduler moves to the next process in the same queue. This approach ensures both prioritization and fairness.

Starvation Avoidance: The MLQ scheduler prevents starvation by ensuring that, at the end of each scheduling round, all queues, regardless of priority, have their slot usage counters reset. This guarantees that even processes in lower-priority queues will eventually be scheduled, albeit with fewer CPU slots. For example, after all higher-priority queues have exhausted their slots, the scheduler will allocate CPU time to lower-priority queues, ensuring no process is completely ignored. In practice, if there are many high-priority processes, lower-priority queues may have to wait longer for their turn. However, the scheduler guarantees that after each round (when all slot usage counters reach zero), all queues, including the lowest priority, are reset and scheduled again. This ensures that even if high-priority queues are always full, lower-priority processes will still be scheduled, albeit with smaller CPU time slices, but with continuous execution opportunities.

3.1.2 Data Structures for the MLQ Scheduler

The MLQ scheduler uses the following core data structures:

- **Priority Queues**: An array of 140 queues, each representing a priority level.

  ```
  static struct queue_t mlq_ready_queue[MAX_PRIO];
  ```

  When a process is loaded or returns after its time slice, it is enqueued into the queue corresponding to its priority (`mlq_ready_queue[prio]`).

  Why use an array? The use of an array of queues for priority management is both simple and efficient. Arrays allow constant-time access to each priority level, making it easy to enqueue and dequeue processes based on their priority. This structure is well-suited for systems with a fixed number of priority levels. For larger or more dynamic systems, alternative data structures such as linked lists or heaps could be considered to optimize queue management and scalability. While an array of queues is efficient for a fixed number of priority levels, the system can be extended to support a dynamic or larger number of queues by using linked lists or heap-based priority queues. This would allow the scheduler to scale efficiently if the number of priority levels increases or becomes dynamic in future system designs.

- **Time Slot Allocation**: Each priority level is assigned a number of CPU slots, calculated as:

```
static int slot[MAX_PRIO];
```

  where `slot[i] = MAX_PRIO - i`, so higher-priority queues receive more CPU time.

- **Slot Usage Tracking**: To ensure fairness, each queue tracks its remaining slots per round:

```
static int slot_usage[MAX_PRIO];
```

  At the start of each round, `slot_usage[i]` is reset to `slot[i]`. This mechanism guarantees that even lower-priority queues will eventually be scheduled, preventing starvation.

3.1.3 Scheduler Initialization

The scheduler is initialized by the function `init_scheduler()` (see Appendix 8.1.2). This function sets up all MLQ queues, assigns time slots to each priority, and initializes slot usage counters.

3.1.4 Process Selection and Scheduling

The core scheduling logic is implemented in `get_mlq_proc()` (see Appendix 8.1.2). This function selects the next process to run as follows:

- At the start of each round, if all `slot_usage[i]` counters are zero, they are reset to their corresponding `slot[i]` values.
- The scheduler iterates from the highest to the lowest priority queue. For each queue, if it has remaining slots and is not empty, a process is dequeued and the slot usage is decremented.

Synchronization: Synchronization is crucial in a multi-threaded scheduler to prevent race conditions. The use of a mutex (`queue_lock`) ensures that only one thread can modify the priority queues at a time. This prevents data corruption and ensures that enqueue and dequeue operations are atomic, maintaining the integrity of the scheduling data structures. Mutexes are chosen for synchronization because they provide efficient and safe mutual exclusion in a multi-threaded environment. Unlike spinlocks, which can waste CPU cycles while waiting, mutexes allow threads to sleep while waiting for the lock, making them suitable for user-level scheduling where context switches are relatively frequent. Semaphores could be used for more complex synchronization patterns, but mutexes are optimal for protecting critical sections like queue operations.

Illustrative Example: Suppose there are three processes:

- P1 (High priority, prio = 0)
- P2 (Medium priority, prio = 1)
- P3 (Low priority, prio = 2)

In each scheduling round, P1 will be scheduled first and receive more CPU slots. Once P1's slots are used up, the scheduler will allocate CPU time to P2, and finally to P3. Even if P1 and P2 have many processes, P3 will still be scheduled once per round, ensuring fairness. If there are multiple processes with the same priority (e.g., P2 and P3 both have prio = 1), they will be scheduled in a round-robin manner within their priority queue. Each process receives a time slice in turn, ensuring fairness among processes of equal priority and preventing any single process from monopolizing the CPU at that priority level.

3.1.5 Process Management Functions

The scheduler exposes several functions for process management (see Appendix 8.1.2, 8.1.2, 8.1.2):

- `get_proc()`: Retrieves the next process to run, internally calling `get_mlq_proc()`.
- `put_proc()`: Returns a process to its priority queue after its time slice expires.
- `add_proc()`: Adds a new process to the scheduler, placing it in the correct queue.
- `put_mlq_proc()`: Enqueues a process into its priority queue.

Both `put_proc()` and `add_proc()` ultimately call `put_mlq_proc()` to ensure consistent queue management.

3.1.6 Fairness and Starvation Avoidance

The slot usage mechanism ensures that all queues, including those with lower priority, will eventually be scheduled once per round. This prevents starvation and guarantees fairness, while still giving higher-priority queues more CPU time.

Starvation Avoidance in Practice: Each time a scheduling round completes (i.e., all slot usage counters reach zero), the system resets the counters, allowing all queues, including the lowest priority, to be scheduled again. This mechanism ensures that no process is permanently deprived of CPU time. In addition, if the system is heavily loaded with high-priority processes, lower-priority processes may receive smaller time slices, but they are still guaranteed to be scheduled in every round.

3.1.7 Integration with OS Core

The scheduler is tightly integrated with the OS core, CPU simulation, memory management, and timer system:

- **CPU Simulation**: Each CPU thread repeatedly calls `get_proc()` to fetch the next process. When a process's time slice expires, `put_proc()` is called to return it to the scheduler. If a process finishes, it is freed, and the CPU fetches the next process.
- **Timer System**: The timer manages time slices and triggers context switches by advancing time slots. Loader and CPU threads synchronize with the timer to maintain correct time-sharing and process loading.
- **Memory Management and System Calls**: The scheduler works closely with the memory management subsystem and the system call interface. For example, when a process in the queue requests memory allocation via a system call, the scheduler may temporarily pause the process and wait for the memory manager to complete the allocation. Only after the memory manager has successfully mapped the required memory will the process be returned to the ready queue for scheduling. This tight coupling ensures that processes are only scheduled when they have all necessary resources, improving both stability and efficiency.

3.2 Memory Management Implementation

3.2.1 Overview

The Memory Management module in OSSim Sierra is responsible for efficiently allocating, tracking, and protecting memory for processes. The design focuses on eliminating external fragmentation using paging, while internal fragmentation is minimized through the management of memory in fixed-size pages. The system also incorporates dynamic memory allocation, virtual memory, and efficient page replacement algorithms to ensure robust performance even under heavy memory usage.

3.2.2 Memory Allocation Strategies

OSSim Sierra implements dynamic memory allocation based on paging. Memory is allocated to processes in page-sized chunks. When a process requests memory, the system searches for a free area in the process's virtual memory area (VMA). If enough contiguous space is available, memory is allocated; otherwise, the system expands the VMA to accommodate the requested size.

Fragmentation Handling

External Fragmentation: Paging effectively eliminates external fragmentation by allowing non-contiguous memory allocation. Every free physical frame can be used to map any virtual page, making memory management efficient and flexible.

Internal Fragmentation: Although external fragmentation is eliminated, internal fragmentation may still occur when the allocated memory for a process does not fully utilize the last page in the VMA. For instance, a process requesting 300 bytes with a 256-byte page size results in the allocation of two pages (512 bytes), with 212 bytes left unused in the second page.

```
int liballoc(struct pcb_t *proc, uint32_t size, uint32_t reg_index) {
    // Allocates memory in page-sized chunks, may cause internal fragmentation
    return __alloc(proc, 0, reg_index, size, &addr);
}
```

Compaction: Compaction is unnecessary in paging systems because the page table maps virtual addresses to physical frames dynamically. No data movement is needed to reduce fragmentation, as any free frame can be used to store any page.

Memory Allocation Policy and Local Replacement

OSSim Sierra does not support memory overcommit: the system will never allocate more memory than the total available RAM and swap space. If a process requests more memory than is available, the allocation will fail immediately, and no "virtual" over-allocation is performed.

Furthermore, OSSim Sierra implements local page replacement: each process can only replace its own pages when handling page faults. There is no global replacement policy; pages from one process cannot be replaced by pages from another process. This design ensures strong process isolation, where each process has control over its own memory and cannot interfere with the memory allocation of other processes.

Despite these restrictions, the combination of `liballoc()` and `libfree()` allows each process to fully utilize its allocated RAM and swap space. When a process frees memory using `libfree()`, the released region is returned to the process's free list and can be reused for future allocations by the same process. This helps minimize internal fragmentation and maximizes memory utilization within the process's own quota.

However, if all physical memory and swap space are exhausted, no process can allocate additional memory, regardless of how much free memory other processes may have. The system will return an error, and the process must handle it or be terminated by the OS.

3.2.3 Paging and Virtual Memory

OSSim Sierra implements virtual memory by dividing the physical memory into fixed-sized pages and mapping these to the virtual memory address space. Each process has its own page table (represented by `mm_struct.pgd`), which tracks the mapping of virtual pages to physical frames.

Page Faults and Address Translation

When a process accesses a memory address, the system translates the virtual address to a physical address using the page table. If the page is not present in RAM, a page fault occurs, and the system will need to load the page into memory by selecting a victim page for swapping (using FIFO or other replacement algorithms).

```
int pg_getpage(struct mm_struct *mm, int pgn, int *fpn, struct pcb_t *caller) {
    if (!PAGING_PAGE_PRESENT(mm->pgd[pgn])) {
        int vicpgn;
        find_victim_page(caller->mm, &vicpgn); // Swap out victim, swap in
            required page
        // Update page table with new page mapping
    }
    *fpn = PAGING_FPN(mm->pgd[pgn]);
    return 0;
}
```

see Appendix 8.2.1

3.2.4 Swapping and Page Replacement Algorithms

When RAM is full, the system needs to swap out pages to secondary storage (swap space) to make room for new pages. OSSim Sierra uses FIFO (First-In, First-Out) by default for page replacement. However, the system is designed to be extendable to other algorithms, such as LRU (Least Recently Used) or Optimal, to improve performance under certain workloads.

FIFO (First-In, First-Out): The FIFO algorithm always selects the oldest page for replacement. The `fifo_pgn` list in `mm_struct` keeps track of the order of pages, and the oldest page is removed when a page fault occurs.

```
int find_victim_page(struct mm_struct *mm, int *retpgn) {
    struct pgn_t *pg = mm->fifo_pgn;
    *retpgn = pg->pgn;
    mm->fifo_pgn = pg->pg_next;
    free(pg);
    return 0;
}
```

LRU (Least Recently Used): LRU replaces the page that has not been accessed for the longest time. This requires tracking page access times, typically using a heap or linked list to store pages in the order of their last access.

```
void lru_page_replacement() {
    struct page *victim = heap_extract_min(lru_heap); // Extract least recently
       used page
    swap_page_to_disk(victim); // Swap out the victim page
}
```

Listing 3.1: Conceptual LRU page replacement using a heap

Optimal Page Replacement: The Optimal algorithm selects the page that will not be used for the longest period of time in the future. While this is ideal, it is not practical for real systems because it requires knowing future page access patterns.

Example: Suppose the page reference string is: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 (with 3 frames):

- **FIFO:** Always replaces the page that has been in memory the longest.
- **LRU:** Replaces the page that has not been accessed for the longest time.
- **Optimal:** Replaces the page that will not be used for the longest time in the future.

3.2.5 Out-of-Memory Error Handling

When both physical memory and swap space are full, memory allocation requests cannot be fulfilled, and the system returns an error (typically `-1` from `liballoc()` or `pg_getpage()`).

Handling Out-of-Memory Errors

The system can notify the process, allowing it to handle the error by freeing memory, retrying the allocation, or terminating.

If the process cannot handle the error, the operating system may terminate the process to prevent instability.

```
if (liballoc(proc, 4096, 0) == -1) {
    // Handle out-of-memory error: process should handle or terminate
}
```

Listing 3.2: Out-of-memory handling example

Optimization Strategy: When memory is nearly exhausted, strategies like garbage collection or aggressive memory reclamation can be applied to reclaim unused memory before the allocation fails.

3.2.6 Flexible Data Structures for Scalability

Currently, OSSim Sierra uses linked lists to manage free memory regions and the FIFO page list. For larger systems or more complex workloads, the following data structures are recommended for scalability:

Red-Black Trees: These can be used to manage free memory regions for fast allocation, improving search and update efficiency, especially in large systems with many pages or memory regions.

```
struct rb_tree *free_region_tree = create_red_black_tree();
```

Listing 3.3: Red-Black tree for fast memory allocation

Heaps: A heap can be used to efficiently manage LRU page replacement by maintaining pages ordered by their last access time.

```
void lru_with_heap() {
    struct page *victim = heap_extract_min(lru_heap);
    swap_page_to_disk(victim);
}
```

Listing 3.4: Heap-based LRU page replacement

3.2.7 Summary

- **Fragmentation:** Paging eliminates external fragmentation; only internal fragmentation remains, which is minimized by managing memory in fixed-size pages.
- **Compaction:** Not necessary due to dynamic mapping via the page table.
- **Page Replacement:** FIFO is used by default, but LRU and Optimal algorithms can be added for better performance.
- **Out-of-Memory Handling:** The system returns an error when both RAM and swap are full and allows the process to handle the error or terminates it if needed.
- **Scalability:** The use of Red-Black Trees and Heaps improves memory region management and page replacement in large systems.
- **Efficient Page Fault Handling:** Optimizing victim selection and minimizing page faults reduces overhead and improves system performance.

3.3 System Call Implementation

System calls provide the interface through which user programs request OS services such as memory management and process control. This section describes the architecture and implementation of system calls in OSSim Sierra.

3.3.1 System Call Architecture

System calls are managed via a **syscall table**, which maps syscall numbers to their corresponding handler functions. The dispatcher receives syscall requests from user programs and routes them to the appropriate handler based on the syscall number.

3.3.2 System Call Dispatcher

The dispatcher function, `syscall()`, uses a switch-case structure (auto-generated from the syscall table) to invoke the correct handler:

```
int syscall(struct pcb_t caller, uint32_t nr, struct sc_regs regs) {
    switch (nr) {
        #include "syscalltbl.lst"
        default: return __sys_ni_syscall(caller, regs);
    }
}
```

3.3.3 Key System Call Handlers

Important system call handlers include:

- `__sys_memmap()`: Handles memory management operations such as memory mapping, region expansion, swapping, and direct memory I/O.
- `__sys_killall()`: Terminates all processes with a matching name in the ready or running queues.
- `__sys_listsyscall()`: Lists all available system calls for debugging or introspection.

3.3.4 System Call Registration

System call registration is automated by the `syscalltbl.sh` script, which parses a syscall definition file and generates the necessary C-compatible dispatch entries. This ensures that each syscall number is correctly mapped to its handler function.

```
#define __SYSCALL(nr, sym) extern int __##sym(struct pcb_t*, struct sc_regs*);
#include "syscalltbl.lst"
#undef __SYSCALL
```

This mechanism allows for easy extension and maintenance of the system call interface in the OS.

3.4 Summary

This chapter covered the implementation details of the key components: the Scheduler, Memory Management, and System Calls. We discussed the core data structures, algorithms, and interactions that enable efficient process scheduling, memory allocation, and system service management.

Chapter 4

Integration of Components

In this chapter, we describe how the key components of the operating system—**Scheduler**, **Memory Management**, **System Calls**, **OS Core**, and the **Timer System**—interact to provide a functional and efficient operating system. The interaction between these components is essential for smooth process execution, efficient memory management, and proper handling of system calls.

4.1 Inter-Module Interaction

The **Scheduler** and **Memory Management** components work closely to ensure processes are efficiently scheduled and have access to the required memory resources. Their seamless interaction guarantees that processes are executed on time and allocated memory dynamically.

4.1.1 Scheduler and Memory Management

- **Scheduler's Role**: The scheduler determines which process should run next by calling `get_proc()` (8.1.2), which implements the Multi-Level Queue (MLQ) scheduling policy. Each process is placed in a priority queue (`mlq_ready_queue`) according to its priority. When a process finishes its time slice, it is returned to the appropriate queue using `put_proc()` (8.1.2).
- **Memory Management's Role**: Memory management is handled per process via the `mm_struct` structure, which maintains the page directory (`pgd`), virtual memory areas (VMAs), and free region lists. When a process requests memory (e.g., via an `alloc` instruction), the system calls `liballoc()` (8.2.1) to allocate a virtual memory region. Physical frames are mapped to virtual pages on demand, and address translation is performed using the page table.
- **Interaction Example**: When a process is loaded, the loader calls `add_proc()` (8.1.2), which initializes the process control block (`pcb_t`) and its memory management structure (`mm_struct`). During execution, instructions such as `alloc`, `free`, `read`, and `write` are handled by the memory management subsystem through functions like `liballoc()` and `libread()` (see Appendix 8.2.1). If a page fault occurs (the required page is not in RAM), the memory manager invokes `pg_getpage()` (8.2.1), which may call `find_victim_page()` (8.2.1) and `__mm_swap_page()` to perform page replacement and swapping.

This integration ensures dynamic resource allocation and efficient management of CPU time and memory.

4.2 OS Core and Timer Integration

The **OS Core** is the central system component that coordinates interactions between the scheduler, memory management, system calls, and other subsystems. It ensures efficient process execution and resource management.

4.2.1 OS Core

The OS core is responsible for:

- **Initializing Components**: It initializes the scheduler, memory management, and system calls, ensuring that all components are ready when the system starts.
- **Process Execution Flow**: It manages process flow, ensuring that processes are scheduled and run in sequence, including invoking system calls when needed.

4.2.2 Timer System

The **Timer System** is critical for allocating CPU time slices and supporting **preemptive scheduling**. It triggers context switches, calling the scheduler to select the next process.

Time Slices: The timer ensures each process receives a time slice. When the time slice expires, the scheduler re-evaluates the queue and selects the next process based on priority and available CPU time.

```
// In timer.c
void timer_routine() {
    // Manage time slices for processes
    scheduler();  // Call the scheduler to select the next process
}
```

By managing time slices, the timer system ensures fairness in CPU time distribution, supporting multitasking.

4.3 System Call Interaction

System calls were detailed in the **Implementation Details** chapter, but it is important to highlight their interaction with other components, particularly **Memory Management** and **Scheduler**.

- **Memory Management and System Calls**: System calls like `memmap` (for memory mapping) and `killall` (for process management) interact with the memory management and scheduler subsystems. For example, `memmap` may trigger memory allocation or swapping, while `killall` might remove processes from the scheduler's queue.
- **Process Management and System Calls**: System calls like `killall` interact directly with the scheduler. When a process is terminated, the scheduler removes it from the ready queue, ensuring it no longer consumes CPU time.

This interaction ensures that system calls can modify the system state—such as memory usage or process execution—by interacting with the appropriate subsystems.

4.4 Potential Synchronization Issues

In a multi-process system, synchronization is crucial to avoid issues like race conditions and deadlocks, especially when processes access shared resources concurrently. In **OSSim Sierra**, synchronization is handled with locking mechanisms (e.g., mutexes), ensuring that only one process can access a resource at a time.

- **Race Conditions**: Without proper locking, processes might simultaneously modify shared data, leading to inconsistent results.
- **Deadlocks**: If two processes hold locks on resources that the other needs, the system could enter a state where no process can proceed.

Careful design of locking mechanisms, particularly with mutexes in critical sections, helps mitigate these risks.

4.5 Summary

This chapter discussed how the key components of the operating system—scheduler, memory management, OS core, timer system, and system calls—interact to provide efficient process execution, memory allocation, and system resource management. By ensuring tight integration between these components, the system achieves dynamic scheduling, memory management, and system call handling, supporting multitasking and efficient resource usage.

Chapter 5

Result Analysis

5.1 Scheduler Output

5.1.1 How CPU time was shared among processes

There are two main scheduling modes: Basic Queue-based Scheduling and Multi-Level Queue Scheduling (MLQ).

- Basic Queue-based Scheduling (FIFO)
  - Processes are executed in the order they arrive in the `ready_queue`.
  - Each process gets a time slice before moving to the next.
  - CPU time is shared equally in a round-robin manner.
- Multi-Level Queue Scheduling (MLQ)
  - Processes are sorted by priority, with higher-priority processes getting CPU time first.
  - Within each priority level, round-robin allocates CPU time.
  - Processes at higher priorities (lower prio values) are executed before lower-priority ones.
  - Time slots are allocated per priority queue, and once slots are exhausted, the scheduler moves to the next lower-priority queue.

In FIFO, all processes share CPU time equally, while in MLQ, higher-priority processes get more frequent CPU time, potentially starving lower-priority ones.

5.1.2 Gantt diagram describing how processes are executed by the CPU

- Input:

```
4 2 3
0 p1s 1
1 p2s 0
2 p3s 0
```

p1s

```
1 10
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
```

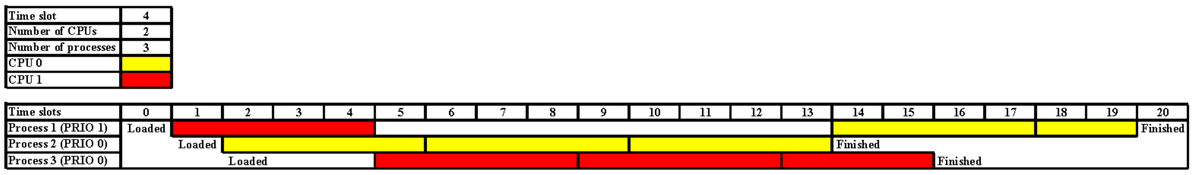
p2s

```
20 12
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
```

p3s

```
7 11
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
```

- Output:



  - **Dual-CPU System**: The log shows a **dual-CPU system**, which improves process throughput as processes can run in parallel on different CPUs.
  - **Process Scheduling**: The system uses **priority-based round-robin scheduling** to fairly allocate CPU time among processes. The higher-priority processes (like **Process 1**) are executed more frequently, while lower-priority processes (like **Process 2, 3, and 4**) are preempted and run intermittently.
  - **Process 3** (PID: 3) was dispatched and finished earlier in Time Slot 16.
  - **Efficient CPU Time Allocation**: Time slots are effectively managed, ensuring no CPU idle time as processes are continually being dispatched and executed.

- Input_0:

```
2 1 2
0 s0 4
4 s1 0
```

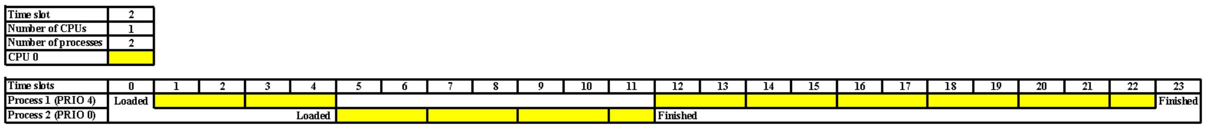
s0

```
12 15
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
```

s1

```
20 7
calc
calc
calc
calc
calc
calc
calc
```

- Output_0:



Time slot	2
Number of CPUs	1
Number of processes	2
CPU 0	🟨

Time slots	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Process 1 (PRIO 4)	Loaded	🟨	🟨	🟨	🟨								🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨	Finished
Process 2 (PRIO 0)					Loaded	🟨	🟨	🟨	🟨	🟨	🟨	🟨	Finished											

  - **Single-CPU System**: All processes are executed on a single CPU, and they are dispatched based on their priority and round-robin scheduling.
  - **Efficient CPU Time Management**: The system effectively uses time slots to allocate CPU time to each process, preventing any process from running indefinitely and ensuring that all processes are eventually executed.
  - **Priority Handling**: The system gives preference to **higher-priority processes**, with Process 1 running the most, while **lower-priority processes** are executed more intermittently.

- Input_1:

```
2 1 4
0 s0 4
4 s1 0
6 s2 0
7 s3 0
```

s0

```
12 15
calc
calc
calc
calc
calc
```

```
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
```

s1

```
20 7
calc
calc
calc
calc
calc
calc
calc
```

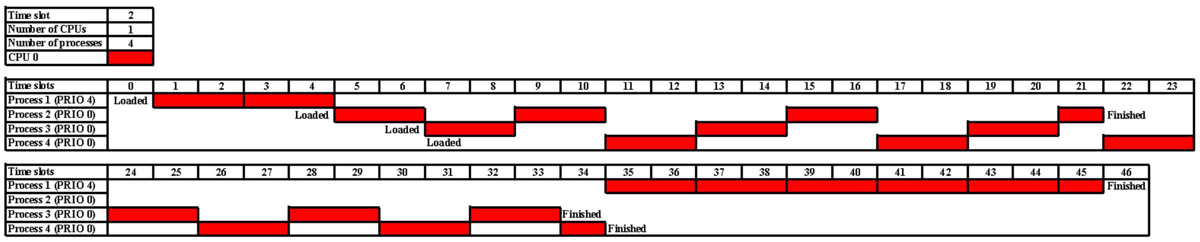
s2

```
20 12
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
```

s3

```
7 11
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
```

- Output_1:



- **Single-CPU System**: All processes are handled by a single CPU, and the system switches between processes based on time slices and priority.
- **Time Slot Utilization**: Each time slot represents an opportunity for a process to execute, and the system efficiently uses time slots to handle multiple processes in a round-robin fashion.
- **Preemption and Priority Handling**g: The scheduling ensures that processes are handled fairly, with **high-priority processes** (like **Process 1**) being favored in terms of CPU time, while still allowing lower-priority processes a chance to run.

5.2 Memory Output

5.2.1 Discuss page faults, swapping, and memory usage.

Implementing paging-based memory management, which involves translating virtual addresses to physical addresses and managing memory through page tables.

- Page Faults
  - Occur when a process accesses a page not in memory.
  - The translate() function checks if the page is present; if not, a page fault happens and the page is fetched from disk or swap space.
- Swapping
  - Swapping moves pages between RAM and swap space to manage limited memory.
  - `MEMPHY_swap_page()` and related functions handle swapping pages in and out.
  - When no free pages are available, a victim page is selected to be swapped out `find_victim_page()`.
- Memory Usage
  - Memory is allocated in pages. The `alloc_mem()` function allocates pages for a process and updates the page table.
  - Pages are tracked with the `_mem_stat[]` array, showing which pages belong to which process.
  - Memory is managed by translating virtual addresses to physical addresses using page tables.

In summary, the system handles page faults by swapping pages in and out of memory, using a paging mechanism to efficiently manage memory across processes.

5.2.2 The status of the memory allocation in heap and data segments

- Heap Segment
  - Memory Allocation: When a process requests memory via `malloc` or similar functions, it typically allocates memory from the heap segment.
  - Memory Deallocation: When `free` is called, the allocated region is deallocated, and the memory is returned to the heap.
- Data Segment

  - Memory Allocation: The data segment holds global and static variables that are initialized by the program. Allocation is done when the program starts, and the data for these variables is mapped to memory.
  - Memory Deallocation: Static and global variables do not typically get deallocated explicitly by the program (since they persist for the duration of the program). However, they can be affected by the operating system during process termination or context switching.

- Additional Insights
  - Page Faults and Memory Mapping
    * The system also tracks page faults, which occur when a process tries to access memory that isn't currently mapped to physical memory. This will trigger the operating system to load the required pages into memory.
    * Swapping
      · If the system runs out of physical memory (RAM), pages from the heap or data segment can be swapped out to swap space on disk. When a page is swapped back into memory, it is marked as "Present" in the page table.
      · The status of the swapped-in or swapped-out pages will be visible in the page table entries.

- Summary: In short, the memory allocation and deallocation statuses in the heap and data segments are managed via the virtual-to-physical memory mapping process. The system tracks these allocations via the page table, marking virtual pages as "Present" or "Not Present" depending on whether the page is loaded into physical memory

5.3 System Call Output

5.3.1 What happened when system calls were used

- When the system call is invoked
  - **Fetch Process Name**: It reads the target process name from a specified memory region using `libread()` until it encounters the end-of-string marker `(-1)`.
  - **Traverse Queues**: The system iterates through process queues (`mlq_ready_queue[MAX_PRIO]`) for each priority level.
  - **Match and Terminate**: For each process, it compares the process name to the target name. If they match, the process is "terminated" by setting its priority to `-1` and removing it from the queue.
  - **Return Killed Count**: The system counts and returns the number of processes terminated.

  The system call retrieves the process name, searches through process queues, terminates matching processes, and returns the number of processes killed.

- In the System
  - **Memory Access**: The system accesses memory regions to retrieve the target process name.
  - **Queue Manipulation**: The system iterates over priority queues to find processes and modify the process list by removing terminated processes.
  - **Process Termination**: Processes are effectively removed from the scheduling queues, and their execution is stopped by setting their priority to `-1`.

  This system call provides a way to terminate multiple processes with the same name, which can be useful in a variety of scenarios where processes need to be killed programmatically based on certain criteria.

5.3.2 The inter-module interactions

- Memory Storing Process Name
  - `sys_killall.c` directly interacts with the process name, reading it from memory `libread()` and comparing it to the processes in the **ready queue**.
  - `sys_listsyscall.c` does not interact with process names.
  - `sys_mem.c` doesn't directly deal with process names but modifies the process's memory.
  - `syscall.c` routes system calls that may indirectly affect process name handling through function calls like `__sys_killall`.
- OS Process Control
  - `sys_killall.c` uses the **PCB** to manipulate process priority and status.
  - `sys_mem.c` uses the **PCB** to access and modify the process's `physical memory`.
  - `syscall.c` passes the **PCB** to the relevant system calls, which may modify process control information.
- Scheduling Queue Management
  - `sys_killall.c` interacts directly with the **multi-level queue** (`mlq_ready_queue`) to search for processes by name and remove them from the queue.
  - Other modules do not directly interact with scheduling queues, but they may indirectly affect them (e.g., `sys_mem.c` can impact process memory usage, potentially influencing process scheduling).

Chapter 6

Question answering

6.1 Scheduling

6.1.1 What are the advantages of using the scheduling algorithms you have learned?

The scheduling algorithm used in this assignment is the **Multilevel Queue (MLQ)** algorithm. It has several advantages:

- **Priority-based Process Management:** MLQ assigns each process a priority and places it in a corresponding queue. This allows high-priority processes to be scheduled before lower-priority ones, ensuring responsiveness for time-sensitive tasks.
- **Separation of Process Classes:** By organizing processes into distinct queues based on priority, the system can handle different types of workloads more effectively (e.g., interactive vs. background jobs).
- **Fairness Within Queues:** Each queue typically uses round-robin scheduling. This ensures that all processes with the same priority receive equal CPU time, promoting fairness within each class.
- **Efficient CPU Utilization:** The scheduler can always pick the next ready process from the highest-priority non-empty queue, reducing CPU idle time and increasing throughput.
- **Scalability and Flexibility:** The structure of MLQ makes it easy to scale to more processes and priority levels. It can also be extended to support additional features such as aging or dynamic priority adjustment.
- **Modularity for Future Improvements:** MLQ is modular, allowing different queues to use different scheduling policies or for future enhancements (like feedback scheduling) to be added with minimal disruption.

Overall, MLQ strikes a good balance between simplicity, efficiency, and fairness, making it suitable for educational operating systems and a stepping stone to more advanced schedulers.

6.2 Memory management

6.2.1 In this simple OS, we implement a design of multiple memory segments or memory areas in source code declaration. What is the advantage of the proposed design of multiple segments?

The design of multiple memory segments (or memory areas) has several advantages in the context of an operating system:

- **Separation of Concerns**

- **Memory Segmentation** allows different types of data or program components to be placed in different segments.
    - **Code Segment** for program instructions.
    - **Data Segment** for global and static variables.
    - **Heap Segment** for dynamically allocated memory.
    - **Stack Segment** for function call data.
- This separation ensures that data types and structures are managed efficiently and are isolated from each other, preventing unintended access or modification.

- **Efficient Memory Allocation**
    - With multiple segments, the OS can allocate memory more efficiently by keeping each type of data (code, data, stack, heap) in its own segment.
    - The heap can grow or shrink dynamically without affecting other segments like the stack or data segment. This helps manage memory in a more flexible way.
- **Memory Protection**
    - Multiple segments allow for better memory protection. Each segment can have different access controls:
        - The code segment might be marked as read-only to prevent accidental modification of instructions.
        - The stack and heap can have different read/write permissions.
    - This helps in preventing bugs or security issues, such as buffer overflows, by isolating the stack and heap from other regions.
- **Optimized Performance**
    - Memory segments allow the OS to use paging or segmentation techniques to optimize performance. For example, different segments can be swapped independently to disk if memory is tight, or specific segments can be cached.
    - Code and read-only data may be placed in memory that is cached more aggressively, improving execution speed, while dynamically allocated heap space can be managed more efficiently.
- **Ease of Debugging** By organizing memory into multiple segments, it becomes easier to identify and debug issues related to specific areas of the program. For instance:
    - Stack overflows can be easily detected if the stack has its own separate memory area.
    - Memory leaks in dynamic memory allocation (heap) can be traced more easily when it's kept isolated.
- **Security** Segment-based memory management enhances security by isolating critical segments (e.g., code) from user data and stack. For example:
    - The stack cannot overwrite code in the code segment, preventing certain types of attacks, like return-oriented programming (ROP) attacks.
- **Dynamic Memory Management**
    - With separate segments for heap, stack, and data, dynamic memory management becomes more predictable and efficient. Each segment has its own allocation strategy:
        - The stack is typically allocated using a LIFO (Last In First Out) scheme.
        - The heap is dynamically managed, allowing for flexible memory allocation and deallocation.
    - This design allows the OS to better handle growing data structures and program execution.
- **Better Control Over Memory Layout** The operating system has more control over memory layout, which can be optimized for specific needs:

  - Stack growth direction can be managed to prevent collisions with other segments.
  - The heap can be allocated with specific memory pools or chunk sizes to optimize usage.

The advantage of using multiple memory segments in an OS design is **efficient management, enhanced security, better performance, and easier debugging**. Each type of memory (code, data, stack, heap) is isolated and managed separately, providing both flexibility and protection for the system and its processes.

6.2.2 What will happen if we divide the address to more than 2 levels in the paging memory management system?

In a multi-level paging system, the address is divided into multiple parts, each indexing into a different level of the page table. For example, in a 3-level system:

- The first part indexes the top-level page table.
- The second part indexes the second-level page table.
- The remaining part is the page offset.

This approach provides:

- **Flexibility:** Better memory management with fewer page table entries for sparsely used memory.
- **Memory Overhead:** More levels require more memory to store additional page tables.
- **Access Time:** More levels increase the time to resolve the address due to multiple lookups.

6.2.3 What are the advantages and disadvantages of segmentation with paging?

- Advantages of Segmentation with Paging:
  - **Improved Memory Management:** Segmentation provides logical divisions (e.g., code, data), while paging eliminates external fragmentation with fixed-size pages.
  - **Flexibility:** Segmentation allows variable-sized segments; paging manages memory efficiently with fixed-size pages.
  - **Protection:** Different access rights can be assigned to segments; paging isolates memory between processes.
  - **Reduced Fragmentation:** Segmentation reduces internal fragmentation; paging eliminates external fragmentation.
- Disadvantages of Segmentation with Paging:
  - **Increased Overhead:** Requires extra structures (e.g., segment and page tables) and complex address translation.
  - **Memory Overhead:** Additional memory is required to store segment and page tables.
  - **Internal Fragmentation:** Pages can still suffer from internal fragmentation, especially if they are not fully used.
  - **Increased Latency:** Multi-level address translation can increase access time.

6.3 System call

6.3.1 What is the mechanism to pass a complex argument to a system call using the limited registers?

Mechanism to Pass Complex Arguments Using Limited Registers:

- **Store Data in Memory:** Instead of passing large structures or arrays, store the data in memory.
- **Pass Pointer in Register(s):** Pass the address (pointer) of the data structure in the register(s).
- **System Call Accesses Data:** The system call uses the pointer to access the data in memory.

Advantages:

- Efficient use of registers.
- Allows passing large data structures without register limitations.

Disadvantages:

- Requires memory management (allocation and deallocation).
- The system call must validate the pointer.

6.3.2 What happens if the syscall job implementation takes too long execution time?

- Consequences of Long Syscall Execution:
  - **Blocking:** The calling process is delayed until the syscall finishes.
  - **Reduced Responsiveness:** Long syscalls can affect system performance, delaying other tasks.
  - **Starvation/Deadlock:** Delays in resource allocation may cause starvation or deadlocks.
  - **Timeouts:** Some syscalls may time out, leading to errors or aborted operations.
  - **High CPU Usage:** Long computation-heavy syscalls can increase CPU load.
  - **Context Switching:** Frequent switching increases overhead.
- Mitigation:
  - Use **non-blocking** or **asynchronous** syscalls.
  - Set **timeouts** and handle **errors**.
  - Offload tasks using **concurrency**.

6.4 Synchronization

6.4.1 What happens if the synchronization is not handled in your Simple OS? Illustrate the problem of your simple OS (assignment outputs) by example if you have any.

If synchronization is not properly handled, the following issues can occur:

- **Race Conditions:** Multiple threads access **shared resources** (like queues, buffers, or task lists) **at the same time**, leading to **unpredictable results**.
- **Data Corruption:** Shared data structures could be **overwritten, lost,** or contain **invalid values** because operations like push/pop are **interleaved incorrectly.**
- **Deadlock:** If two or more threads are waiting for each other's resources indefinitely, the system can **freeze** (deadlock).

For example: If synchronization is not properly handled in `queue.c` (normal version in Appendix 8.1.1). If no `pthread_mutex_lock/pthread_mutex_unlock`, the operations on `q->size` and `q->proc[]` are unsafe when accessed by multiple threads.

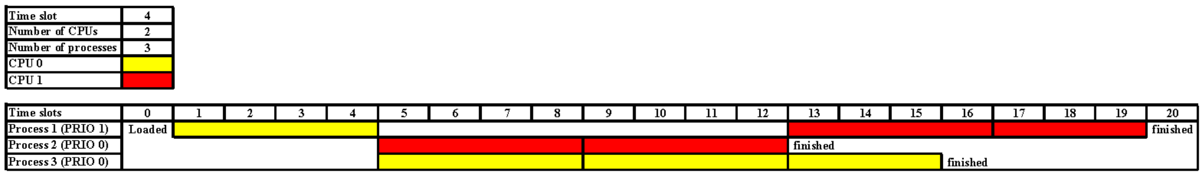
```
void enqueue(struct queue_t* q, struct pcb_t* proc) {
    if (q == NULL || proc == NULL) return;
    if (q->size >= MAX_QUEUE_SIZE) return;

    q->proc[q->size] = proc;
    q->size++;   // <-- RACE CONDITION if 2 threads update q->size together
}

struct pcb_t* dequeue(struct queue_t* q) {
    if (q == NULL || q->size <= 0) return NULL;

    // Assume we simply pop the last inserted process
    struct pcb_t* proc = q->proc[q->size - 1];
    q->size--;   // <-- RACE CONDITION here too

    return proc;
}
```



The correct sched output is in 5.1.1

Comparison between Buggy Output and Correct Output

Buggy Output	Correct Output	Comments
Time slot 0: CPU 0 dispatches process 1	**Time slot 1:** CPU 1 dispatches process 1	CPU dispatching is inconsistent (CPU 0 vs CPU 1). Task assignment is not synchronized.
Time slot 5: CPU 0 puts process 1 and dispatches process 3 at the same time	**Time slot 4:** CPU 1 puts process 1 first, then dispatches process 3	Bug: simultaneous put and dispatch, wrong order of operations.
Time slot 5: CPU 1 also puts process 2 and dispatches process 2	**Time slot 6:** CPU 0 correctly puts process 2 then dispatches it	Race condition: two CPUs are competing to put or dispatch at the same time.
Time slot 9: CPU 0 puts 3 and dispatches 3, CPU 1 puts 2 and dispatches 2	**Time slot 10:** CPU 0 properly puts and dispatches 2	No correct sequencing: Buggy version overlaps put and dispatch actions.
Time slot 13: CPU 0 puts 3, dispatches 3, CPU 1 finishes process 2 and dispatches process 1	**Time slot 14:** CPU 0 finishes process 2 first, then dispatches 1	Process finish and dispatch are mixed incorrectly across CPUs.
Time slot 16-17: CPU 0 stops, CPU 1 puts and dispatches process 1	**Time slot 18:** CPU 0 puts and dispatches process 1	CPU roles are swapped wrongly (CPU 1 doing what CPU 0 should).

Buggy Output	**Correct Output**	**Comments**
Final: CPU 1 stops after CPU 0	**Final:** CPU 1 stops before CPU 0	CPU shutdown sequence is incorrect.

Chapter 7

Conclusion

The development of the **OSSim Sierra** operating system provided a comprehensive hands-on experience in building and integrating core OS components. Through this project, I developed a deeper understanding of how the **Scheduler**, **Memory Management**, **System Calls**, **OS Core**, and **Timer System** collaborate to manage processes and system resources efficiently.

Implementing each module from scratch enhanced my practical knowledge in areas such as:

- Multi-level queue (MLQ) scheduling
- Paging and memory management
- Context switching and preemptive scheduling
- Synchronization and mutual exclusion

Challenges Encountered:

- Ensuring proper synchronization across modules, particularly during concurrent access and context switching.
- Debugging race conditions and avoiding deadlocks required careful use of mutexes and locking mechanisms.
- Managing memory efficiently using paging and swap space under limited physical memory.
- Maintaining system stability during stress testing with high process loads and memory usage.

Possible Improvements:

- Optimize the page replacement algorithm to reduce latency under memory pressure.
- Introduce support for more advanced system calls and inter-process communication (IPC).
- Add dynamic monitoring tools to observe system performance in real-time.
- Refactor and modularize code further to improve scalability and maintainability.

Overall, this assignment reinforced my understanding of core operating system concepts and highlighted the complexity and interdependence of OS components. It was a rewarding experience that bridged theory and practice, deepening my appreciation for OS design and implementation.

Chapter 8

Appendix

- Sample input/output screenshots (attach images or use `lstlisting` for code)
- Key code snippets (if needed)

8.1 Scheduling

8.1.1 src/queue.c (Queue Operations)

`void enqueue(struct queue_t *q, struct pcb_t *proc)`

```
void enqueue(struct queue_t* q, struct pcb_t* proc) {
    if (q == NULL || proc == NULL) return;
    if (q->size >= MAX_QUEUE_SIZE) {
        fprintf(stderr, "Queue is full, cannot enqueue process %d\n", proc->pid)
            ;
        fflush(stderr);
        return;
    }

    if (q->size < 0)
        q->size = 0;

    q->proc[q->size] = proc;
    q->size++;
}
```

`struct pcb_t* dequeue(struct queue_t* q)`

```
struct pcb_t* dequeue(struct queue_t* q) {
    if (q == NULL || q->size <= 0)
        return NULL;

    if (q->size > MAX_QUEUE_SIZE)
        q->size = MAX_QUEUE_SIZE;

    int highest_priority_idx = 0;
    for (int i = 1; i < q->size; i++) {
        if (q->proc[i] == NULL)
            continue;

        if (q->proc[highest_priority_idx] == NULL) {
            highest_priority_idx = i;
            continue;
        }

        #ifdef MLQ_SCHED
            if (q->proc[i]->prio < q->proc[highest_priority_idx]->prio)
                highest_priority_idx = i;
        #else
            if (q->proc[i]->priority < q->proc[highest_priority_idx]->priority)
                highest_priority_idx = i;
        #endif
    }

    if (q->proc[highest_priority_idx] == NULL)
        return NULL;

    struct pcb_t* highest_proc = q->proc[highest_priority_idx];

    for (int i = highest_priority_idx; i < q->size - 1; i++)
        q->proc[i] = q->proc[i + 1];

    q->size--;
    return highest_proc;
}
```

int empty(struct queue_t* q)

```
int empty(struct queue_t* q) {
    if (q == NULL)
        return 1;

    return (q->size == 0);
}
```

8.1.2 src/sched.c (Scheduler)

void init_scheduler(void)

```
void init_scheduler(void) {
    #ifdef MLQ_SCHED
        for (int i = 0; i < MAX_PRIO; i++) {
            mlq_ready_queue[i].size = 0;
            slot[i] = MAX_PRIO - i;
            slot_usage[i] = slot[i];
        }
    #endif
    ready_queue.size = 0;
    run_queue.size = 0;
    pthread_mutex_init(&queue_lock, NULL);
}
```

struct pcb_t * get_mlq_proc(void)

```
struct pcb_t * get_mlq_proc(void) {
    pthread_mutex_lock(&queue_lock);

    bool all_zero = true;
    for (int i = 0; i < MAX_PRIO; i++) {
        if (slot_usage[i] > 0) {
            all_zero = false;
```

```
                break;
            }
        }
        if (all_zero) {
            for (int i = 0; i < MAX_PRIO; i++) {
                slot_usage[i] = slot[i];
            }
        }

        struct pcb_t * proc = NULL;
        for (int pr = 0; pr < MAX_PRIO; pr++) {
            if (slot_usage[pr] > 0 && !empty(&mlq_ready_queue[pr])) {
                proc = dequeue(&mlq_ready_queue[pr]);
                slot_usage[pr]--;
                pthread_mutex_unlock(&queue_lock);
                return proc;
            }
        }
        pthread_mutex_unlock(&queue_lock);
        return NULL;
}
```

void put_mlq_proc(struct pcb_t * proc)

```
void put_mlq_proc(struct pcb_t * proc) {
    if(proc == NULL) return;
    pthread_mutex_lock(&queue_lock);
        enqueue(&mlq_ready_queue[proc->prio], proc);
    pthread_mutex_unlock(&queue_lock);
}
```

struct pcb_t * get_proc(void)

```
struct pcb_t * get_proc(void) {
    return get_mlq_proc();
}
```

void put_proc(struct pcb_t * proc)

```
void put_proc(struct pcb_t * proc) {
    if(proc == NULL) return;
    proc->ready_queue = &ready_queue;
    proc->mlq_ready_queue = mlq_ready_queue;
    proc->running_list = &running_list;
    put_mlq_proc(proc);
}
```

void add_proc(struct pcb_t * proc)

```
void add_proc(struct pcb_t * proc) {
    if(proc == NULL) return;
    put_proc(proc);
}
```

8.2 Memory

8.2.1 src/libmem.c (Memory Management - User-level API & Helpers)

`int liballoc(struct pcb_t *proc, uint32_t size, uint32_t reg_index)`

```
int liballoc(struct pcb_t *proc, uint32_t size, uint32_t reg_index) {
  int addr;
  return __alloc(proc, 0, reg_index, size, &addr);
}
```

`int libfree(struct pcb_t *proc, uint32_t reg_index)`

```
int libfree(struct pcb_t *proc, uint32_t reg_index) {
  return __free(proc, 0, reg_index);
}
```

`int libread(struct pcb_t *proc, uint32_t source, uint32_t offset, uint32_t *destination)`

```
int libread(struct pcb_t *proc, uint32_t source, uint32_t offset,
            uint32_t *destination) {
    BYTE data;
    int result = __read(proc, 0, source, offset, &data);

    if (result == 0 && destination != NULL)
        *destination = data;

    #ifdef IODUMP
        printf("===== PHYSICAL MEMORY AFTER READING =====\n");  // Added Header
        printf("read region=%d offset=%d value=%d\n", source, offset, data);
    #ifdef PAGETBL_DUMP
        print_pgtbl(proc, 0, (uint32_t)-1);  // print max TBL (cast -1 for
           clarity)
    #endif
        MEMPHY_dump(proc->mram);
    #endif

    return result;
}
```

`int libwrite(struct pcb_t *proc, BYTE data, uint32_t destination, uint32_t offset)`

```
int libwrite(struct pcb_t *proc, BYTE data, uint32_t destination,
             uint32_t offset) {
    #ifdef IODUMP
        printf("===== PHYSICAL MEMORY AFTER WRITING =====\n");
        printf("write region=%d offset=%d value=%d\n", destination, offset, data
           );
    #ifdef PAGETBL_DUMP
        print_pgtbl(proc, 0, (uint32_t)-1);
    #endif
        MEMPHY_dump(proc->mram);
    #endif

    return __write(proc, 0, destination, offset, data);
}
```

int __alloc(struct pcb_t *caller, int vmaid, int rgid, int size, int *alloc_addr)

```
int __alloc(struct pcb_t *caller, int vmaid, int rgid, int size,
            int *alloc_addr) {
    pthread_mutex_lock(&mmvm_lock);
    struct vm_rg_struct rgnode;

    if (get_free_vmrg_area(caller, vmaid, size, &rgnode) == 0) {
        caller->mm->symrgtbl[rgid].rg_start = rgnode.rg_start;
        caller->mm->symrgtbl[rgid].rg_end = rgnode.rg_end;
        *alloc_addr = rgnode.rg_start;
        pthread_mutex_unlock(&mmvm_lock);
        return 0;
    }

    struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);
    int inc_sz = PAGING_PAGE_ALIGNSZ(size);
    int inc_limit_ret;

    if (cur_vma == NULL) {
        pthread_mutex_unlock(&mmvm_lock);
        return -1;
    }

    inc_limit_ret = inc_vma_limit(caller, vmaid, inc_sz);
    if (inc_limit_ret < 0) {
        pthread_mutex_unlock(&mmvm_lock);
        return -1;
    }

    if (get_free_vmrg_area(caller, vmaid, size, &rgnode) == 0) {
        caller->mm->symrgtbl[rgid].rg_start = rgnode.rg_start;
        caller->mm->symrgtbl[rgid].rg_end = rgnode.rg_end;
        *alloc_addr = rgnode.rg_start;
        pthread_mutex_unlock(&mmvm_lock);
        return 0;
    }

    return -1;
}
```

int __free(struct pcb_t *caller, int vmaid, int rgid)

```
int __free(struct pcb_t *caller, int vmaid, int rgid) {
    if (rgid < 0 || rgid >= PAGING_MAX_SYMTBL_SZ) return -1;

    struct vm_rg_struct *rgnode = get_symrg_byid(caller->mm, rgid);
    if (rgnode == NULL) return -1;

    enlist_vm_freerg_list(caller->mm, rgnode);

    return 0;
}
```

int pg_getpage(struct mm_struct *mm, int pgn, int *fpn, struct pcb_t *caller)

```
int pg_getpage(struct mm_struct *mm, int pgn, int *fpn, struct pcb_t *caller) {
    if(mm == NULL || fpn == NULL || caller == NULL)
        return -1;
    int ret = 0;

    uint32_t pte = mm->pgd[pgn];

    if (!PAGING_PAGE_PRESENT(pte)) {
        int vicpgn, swpfpn, tgtfpn;

        ret = find_victim_page(caller->mm, &vicpgn);
        if(ret == -1)
            return -1;

        ret = MEMPHY_get_freefp(caller->active_mswp, &swpfpn);
        if(ret == -1)
            return -1;

        tgtfpn = PAGING_PTE_SWP(pte);

        __mm_swap_page(caller, vicpgn, swpfpn);
        __mm_swap_page(caller, tgtfpn, vicpgn);

        pte_set_swap(&mm->pgd[vicpgn], 0, swpfpn);
        pte_set_fpn(&mm->pgd[pgn], vicpgn);

        ret = enlist_pgn_node(&caller->mm->fifo_pgn, pgn);
        if(ret == -1)
            return -1;
    }
    *fpn = PAGING_FPN(mm->pgd[pgn]);
    return 0;
}
```

int pg_getval(struct mm_struct *mm, int addr, BYTE *data, struct pcb_t *caller)

```
int pg_getval(struct mm_struct *mm, int addr, BYTE *data,
              struct pcb_t *caller) {
    int pgn = PAGING_PGN(addr);
    int off = PAGING_OFFST(addr);
    int fpn;

    if (pg_getpage(mm, pgn, &fpn, caller) != 0)
        return -1;

      int phyaddr = (fpn << PAGING_ADDR_SHIFT) | off;

    if (MEMPHY_read(caller->mram, phyaddr, data) != 0)
        return -1;

    return 0;
}
```

int pg_setval(struct mm_struct *mm, int addr, BYTE value, struct pcb_t *caller)

```
int pg_setval(struct mm_struct *mm, int addr, BYTE value,
              struct pcb_t *caller) {
    int pgn = PAGING_PGN(addr);
    int off = PAGING_OFFST(addr);
    int fpn;
    if (pg_getpage(mm, pgn, &fpn, caller) != 0)
        return -1;

    int phyaddr = (fpn << PAGING_ADDR_SHIFT) | off;

    if (MEMPHY_write(caller->mram, phyaddr, value) != 0)
        return -1;

    return 0;
}
```

int find_victim_page(struct mm_struct *mm, int *retpgn)

```
int find_victim_page(struct mm_struct *mm, int *retpgn) {
    if (mm == NULL || retpgn == NULL)
        return -1;

    struct pgn_t *pg = mm->fifo_pgn;

    if (pg == NULL)
        return -1;

    *retpgn = pg->pgn;
    mm->fifo_pgn = pg->pg_next;

    free(pg);
    return 0;
}
```

int get_free_vmrg_area(struct pcb_t *caller, int vmaid, int size, struct vm_rg_struct *newrg)

```
int get_free_vmrg_area(struct pcb_t *caller, int vmaid, int size,
                       struct vm_rg_struct *newrg) {
    if (caller == NULL || newrg == NULL || size < 0)
        return -1;

    struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);
    if (cur_vma == NULL)
        return -1;

    struct vm_rg_struct *rgit = cur_vma->vm_freerg_list;
    struct vm_rg_struct *prev = NULL ;

    while (rgit != NULL) {
        if ((rgit->rg_end - rgit->rg_start) >= size) {
            newrg->rg_start = rgit->rg_start;
            newrg->rg_end = rgit->rg_start + size;

            rgit->rg_start += size;

            if (rgit->rg_start == rgit->rg_end) {
                if(prev == NULL)
                    cur_vma->vm_freerg_list = rgit->rg_next;
                else
                    prev->rg_next = rgit->rg_next;

                free(rgit);
            }
            return 0;
        }
        prev = rgit;
        rgit = rgit->rg_next;
    }
    return -1;
}
```

8.2.2

8.3 SystemCall

8.3.1 sys_killall.c

```
int __sys_killall(struct pcb_t *caller, struct sc_regs* regs){
    char proc_name[100];
    uint32_t data;
    uint32_t memrg = regs->a1;

    int i = 0;
    data = 0;
    while (data != -1 && i < 99) { // Add limit
        libread(caller, memrg, i, &data);
        proc_name[i] = (char)data;
        if (data == -1 || data == 0) {
            proc_name[i] = '\0';
            break;
        }
        i++;
    }
    proc_name[i] = '\0';
    printf("The procname retrieved from memregionid %d is \"%s\"\n", memrg,
       proc_name);

    for (int lvl = 0; lvl < MAX_PRIO; lvl++) {
        struct queue_t *queue = &caller->mlq_ready_queue[lvl];
        if (!queue || empty(queue)) continue;

        int idx = 0;
        while (idx < queue->size) {
            struct pcb_t *proc = queue->proc[idx];
            if (proc && strcmp(proc->path, proc_name) == 0) {
                printf("Terminated process PID %d with name \"%s\"\n", proc->pid
                    , proc->path);

                free(proc->code);
#ifdef MM_PAGING
                if (proc->mm) free(proc->mm);
#endif
                for (int k = idx; k < queue->size - 1; k++) {
                    queue->proc[k] = queue->proc[k + 1];
                }
                queue->proc[queue->size - 1] = NULL;
                queue->size--;
            } else {
                idx++;
            }
        }
    }

    struct queue_t *run_queue = caller->running_list;
    if (run_queue && !empty(run_queue)) {
        int idx = 0;
        while (idx < run_queue->size) {
            struct pcb_t *proc = run_queue->proc[idx];
            if (proc && strcmp(proc->path, proc_name) == 0) {
                printf("Terminated running process PID %d with name \"%s\"\n",
                    proc->pid, proc->path);

                free(proc->code);
#ifdef MM_PAGING
                if (proc->mm) free(proc->mm);
#endif
                for (int k = idx; k < run_queue->size - 1; k++) {
                    run_queue->proc[k] = run_queue->proc[k + 1];
                }
                run_queue->proc[run_queue->size - 1] = NULL;
                run_queue->size--;
            } else {
                idx++;
            }
        }
    }

    return 0;
}
```

8.3.2 sys_listsyscall.c

```
int __sys_listsyscall(struct pcb_t *caller, struct sc_regs* reg){
   for (int i = 0; i < syscall_table_size; i++)
       printf("%s\n",sys_call_table[i]);

   return 0;
}
```

8.3.3 sys_mem.c

```
int __sys_memmap(struct pcb_t *caller, struct sc_regs* regs){
   int memop = regs->a1;
   BYTE value;

   switch (memop) {
   case SYSMEM_MAP_OP:
            /* Reserved process case*/
            break;
   case SYSMEM_INC_OP:
            inc_vma_limit(caller, regs->a2, regs->a3);
            break;
   case SYSMEM_SWP_OP:
            __mm_swap_page(caller, regs->a2, regs->a3);
            break;
   case SYSMEM_IO_READ:
            MEMPHY_read(caller->mram, regs->a2, &value);
            regs->a3 = value;
            break;
   case SYSMEM_IO_WRITE:
            MEMPHY_write(caller->mram, regs->a2, regs->a3);
            break;
   default:
```

```
            printf("Memop code: %d\n", memop);
            break;
   }

   return 0;
}
```

8.3.4 syscall.c

```
int syscall(struct pcb_t *caller, uint32_t nr, struct sc_regs* regs){
    switch (nr) {
    #include "syscalltbl.lst"
    default: return __sys_ni_syscall(caller, regs);
    }
};
```