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

OPERATING SYSTEMS(CO2018)

Assignment

Simple Operating System

Lecturer(s):	Diệp Thanh Đăng, *CSE-HCMUT*
Group:	CC06 - *Team 9*
Students:	Nguyễn Văn Hoàng - 2352355 Nguyễn Thanh Huy - 2352399 Trần Ngọc Thoại - 2353174 Đỗ Hữu Di - 2352162 Mai Chung Tiến - 2353177

HO CHI MINH CITY, NOVEMBER 2024

Contents

List of Symbols

List of Acronyms

Fig.	Figure
Tab.	Table
Eq.	Equation
e.g.	For Example

List of Figures

List of Tables

Member list & Workload

No.	Fullname	Student ID	Problems	% done
1	Nguyễn Văn Hoàng	2352355	- Scheduling Coding	100%
2	Nguyễn Thanh Huy	2352399	- Introduction - LaTeX	100%
3	Trần Ngọc Thoại	2353147	- Conclusion - LaTeX	100%
4	Đỗ Hữu Di	2352162	- System Call Coding	100%
5	Mai Chung Tiến	2353177	- Memory Management Coding	100%

Table 1: Member list & workload

1 Introduction

Operating systems are central to modern computing, serving as the critical layer that coordinates hardware resources and user-level applications. They handle essential tasks such as process scheduling, memory management, and communication between system components, ensuring smooth and secure operation across all processes. This assignment delves into the core principles of operating systems by guiding students through the design of a basic scheduling system using system calls.

The project focuses on the development and simulation of a custom process scheduling algorithm. This scheduler takes into account multiple factors—priority, burst time, and arrival time—to determine the execution order of processes. By considering these elements together, the goal is to enhance fairness and maximize resource efficiency. Students are expected to manage a process queue, calculate scheduling sequences, and emulate execution behavior reflective of actual operating systems.

A key component of this project is the use of system calls, which act as the interface between user applications and the kernel. These calls are essential for operations such as process creation, data exchange, and invoking scheduling routines. Through hands-on experience with system-level interactions, the assignment sheds light on how operating systems provide fundamental services to user programs.

By merging algorithmic thinking with practical coding tasks, this assignment demonstrates how operating systems work internally. It also highlights the complexities involved in achieving fair and efficient process management—laying the groundwork for exploring more sophisticated system-level concepts and real-world OS design challenges.

2 Scheduling

2.1 Quetions

Question: What are the advantages of using the scheduling algorithm described in this assignment compared to other scheduling algorithms you have learned?

Answer:

- **Disadvantages of other scheduling algorithms:**
  - **First Come First Serve (FCFS):**
    In FCFS, processes are scheduled in the order they arrive, regardless of their burst time. So if a long process arrives first, all the shorter processes must wait, even if they could finish quickly. This causes:
    - Long waiting times for shorter processes.
    - Reduced throughput.
    - Poor CPU and I/O device utilization, since short I/O-bound processes may be delayed, leaving devices idle.
  - **Shortest Job First (SJF):**
    SJF always selects the process with the shortest burst time to run next. If shorter processes keep arriving, longer processes may be postponed indefinitely. This leads to starvation.
  - **Round Robin (RR):**
    In Round Robin, each process gets a fixed time slice (quantum) to run. After that, it's preempted and the CPU switches to the next process in the queue. Causing:
    - If the time slice is too large, RR starts behaving like FCFS, losing its responsiveness.
    - If the time slice is too small, the CPU spends a lot of time switching between processes causing overhead from frequent context switching. As a result, overall CPU efficiency drops, and performance may suffer.
  - **Priority Scheduling (PS):**
    In Priority Scheduling, the CPU is assigned to the process with the highest priority. This can lead to low-priority processes waiting indefinitely if high-priority processes keep arriving, causing starvation.
- **Advantages of using the scheduling algorithm described:**
  - Combines **Fairness** and **Responsiveness**: Round Robin is applied within each queue, ensuring fair time-sharing among processes of the same priority. This prevents starvation within a priority level, unlike basic Priority Scheduling.
  - Prevents **Starvation**: By limiting each priority level's CPU time using a fixed number of slots high-priority queues can't hog the CPU forever and lower-priority queues eventually get their turn, reducing starvation (a common issue in traditional Priority Scheduling and SJF).
  - Supports **Multicore Scheduling**: The design is suited for multiple processors, allowing parallel scheduling. This improves CPU utilization and scalability, which FCFS, SJF, or even plain RR don't inherently support.

2.2 Gantt diagram

Requirement: Draw Gantt diagram describing how processes are executed by the CPU.

TEST 0: In this test, the CPU executes 2 processes in 23 time slots.

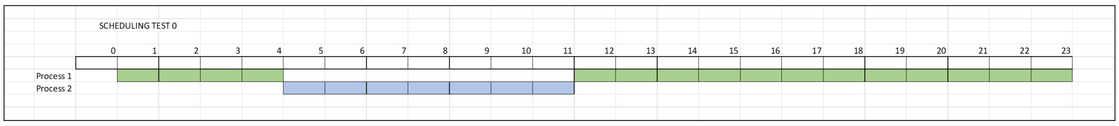

Figure 1: Sched-0

TEST 1: In this test, the CPU executes 4 processes in 46 time slots.

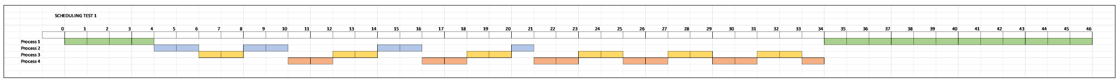

Figure 2: Sched-1

TEST: In this test, 2 CPU executes 3 processes in 20 time slots.

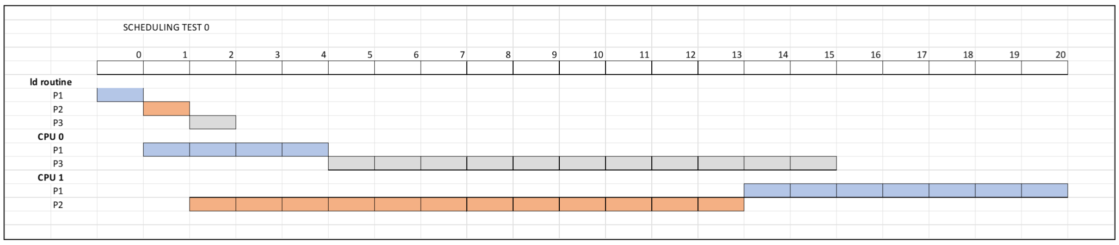

Figure 3: Sched

2.3 Implementation

2.3.1 Queue

- enqueue(): adds a process (proc) to the end of the queue (q) if it's available.
- dequeue(): Removes the first (oldest) process and then shifts all remaining elements forward.

```c
void enqueue(struct queue_t * q, struct pcb_t * proc) {
        /* TODO: put a new process to queue [q] */
        if(q == NULL || proc == NULL)
        {
                return;
        }

        if(q->size == MAX_QUEUE_SIZE)
        {
                printf("Exceed queue size (MAX: 10) !!! \n");
                return;
        }
        q->proc[q->size] = proc;
        q->size++;

}

}

struct pcb_t * dequeue(struct queue_t * q) {
        /* TODO: return a pcb whose prioprity is the highest
         * in the queue [q] and remember to remove it from q
         * */

         if(q == NULL || q->size == 0)
         {
                return NULL;
         }
        struct pcb_t* to_return = q->proc[0];
        for(int i = 0; i < q->size - 1; i++)
        {
                q->proc[i] = q->proc[i + 1];
        }
        q->size--;
        return to_return;
}
```

2.3.2 Scheduler

- `get_mlq_proc()`: Scans priority queues from prio $= 0$ (highest) to `MAX_PRIO - 1` (lowest). If slot[prio] > 0, it dequeues and runs the process. If no process can be run because slots are exhausted, all slots are reset and the first available process in the highest non-empty queue is dequeued.
- `put_mlq_proc()`: Re-enqueues the process into its appropriate priority queue.

```c
struct pcb_t * get_mlq_proc(void) {
    struct pcb_t * proc = NULL;

    // Lock the queue to ensure thread safety
    pthread_mutex_lock(&queue_lock);

    unsigned long prio;
    int first_non_empty_queue = -1;
    int process_found = 0;

    // Iterate through all priority levels
    for (prio = 0; prio < MAX_PRIO; prio++) {
        if (!empty(&mlq_ready_queue[prio])) {
            // Mark the first non-empty queue
            if (first_non_empty_queue == -1) {
                first_non_empty_queue = prio;
            }

            // Check if the current priority level has available slots
            if (slot[prio] > 0) {
                proc = dequeue(&mlq_ready_queue[prio]);
                if (proc != NULL) {
                    process_found = 1;
                    slot[prio]--;
                    break;
                }
            }
        }
    }

    // If no process is found in any queue
    if (first_non_empty_queue == -1) {
        pthread_mutex_unlock(&queue_lock);
        return NULL;
    } else {
        // If no process could run due to exhausted slots
        if (process_found == 0) {
            // Reset all slots to their initial values
            for (prio = 0; prio < MAX_PRIO; prio++) {
                slot[prio] = MAX_PRIO - prio;
            }

            // Dequeue a process from the first non-empty queue
            proc = dequeue(&mlq_ready_queue[first_non_empty_queue]);
            if (proc != NULL) {
                slot[prio]--;
            }
        }
    }

    // Unlock the queue and return the selected process
    pthread_mutex_unlock(&queue_lock);
    return proc;
}

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

2.3.3 Test Running

Input:

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

This input sets up a scheduling simulation over 4 time slots using 2 CPUs and 3 processes. Each process is loaded at a specific time: p1s at time 0 with priority 1, p2s at time 1 with priority 0, and p3s at time 2 with priority 0. Each process file contains instructions the CPU will execute:

- p1s contains 10 instructions.
- p2s contains 12 instructions.
- p1s contains 11 instructions.

Output:

```
Time slot   0
```

```
ld_routine
        Loaded a process at input/proc/p1s, PID: 1 PRIO: 1
Time slot   1
        CPU 0: Dispatched process  1
        Loaded a process at input/proc/p2s, PID: 2 PRIO: 0
Time slot   2
        CPU 1: Dispatched process  2
        Loaded a process at input/proc/p3s, PID: 3 PRIO: 0
Time slot   3
Time slot   4
Time slot   5
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  3
Time slot   6
        CPU 1: Put process  2 to run queue
        CPU 1: Dispatched process  2
Time slot   7
Time slot   8
Time slot   9
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  3
        CPU 1: Put process  2 to run queue
        CPU 1: Dispatched process  2
Time slot  10
Time slot  11
Time slot  12
Time slot  13
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  3
        CPU 1: Processed  2 has finished
Time slot  14
        CPU 1: Dispatched process  1
Time slot  15
Time slot  16
        CPU 0: Processed  3 has finished
        CPU 0 stopped
Time slot  17
Time slot  18
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  1
Time slot  19
Time slot  20
        CPU 1: Processed  1 has finished
        CPU 1 stopped
```

Explanation:

Each CPU runs 1 instruction at a time.

```
Time slot   0
ld_routine
    Loaded a process at input/proc/p1s, PID: 1 PRIO: 1
```

- At time slot 0, p1s is loaded (PID 1, priority 1).

```
Time slot   1
    CPU 0: Dispatched process  1
    Loaded a process at input/proc/p2s, PID: 2 PRIO: 0
```

- CPU 0 is free so it immediately dispatches process 1.
- At time 1, p2s (PID 2, higher priority) is loaded.

```
Time slot   2
    CPU 1: Dispatched process  2
    Loaded a process at input/proc/p3s, PID: 3 PRIO: 0
```

- CPU 1 is free so it immediately dispatches process 2.
- p3s is loaded (PID 3, also priority 0).
- All CPUs are now running: CPU 0 → PID 1, CPU 1 → PID 2.

```
Time slot   3
Time slot   4
```

- No process finishes or gets requeued yet. Both CPUs are still running their current processes.

```
Time slot   5
    CPU 0: Put process  1 to run queue
    CPU 0: Dispatched process  3
```

- Time slice of PID 1 is over.
- It's added back to the run queue.
- CPU 0 picks the highest priority process in the queue → PID 3 (priority 0) and dispatches it.

```
Time slot   6
    CPU 1: Put process  2 to run queue
    CPU 1: Dispatched process  2
```

- Time slice of PID 2 is over.
- CPU 1 still selects PID 2 again (highest priority in queue).

```
Time slot   7
Time slot   8
```

- CPUs continue running: CPU 0 → PID 3, CPU 1 → PID 2.

```
Time slot   9
    CPU 0: Put process  3 to run queue
    CPU 0: Dispatched process  3
    CPU 1: Put process  2 to run queue
    CPU 1: Dispatched process  2
```

- Time slice of PID 3 is over.
- CPU 0 still selects PID 3 again (highest priority in queue).
- Time slice of PID 2 is over.
- CPU 1 still selects PID 2 again (highest priority in queue).

```
Time slot  10
Time slot  11
Time slot  12
```

- CPUs continue running: CPU 0 → PID 3, CPU 1 → PID 2.

```
Time slot  13
    CPU 0: Put process  3 to run queue
    CPU 0: Dispatched process  3
    CPU 1: Processed  2 has finished
```

- Time slice of PID 3 is over.
- CPU 0 still selects PID 3 again (highest priority in queue).
- PID 2 has now finished all its calcs.

```
Time slot  14
    CPU 1: Dispatched process  1
```

- CPU 0 picks the highest priority process in the queue → PID 1 because it's the only one left and dispatches it.

```
Time slot  16
    CPU 0: Processed  3 has finished
    CPU 0 stopped
```

- PID 3 has now finished all its calcs.
- CPU 0 stopped working.
- For time slice 17 → 19, other processes had completed their job, so the CPU only work with PID 1 until it's end.

```
Time slot  20
    CPU 1: Processed  1 has finished
    CPU 1 stopped
```

- PID 1 has now finished all its calcs.
- CPU 1 stopped working.

3 Memory Management

3.1 Quetions

Question:

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?

Answer:
The proposed design of using multiple memory segments in this os offers several advantages for memory management:

- **Isolation of Memory**

  Each process has its own set of memory segments (e.g., code, stack, heap), which are managed independently. This isolation ensures that processes cannot inadvertently access or modify each other's memory, enhancing security and stability.

- **Memory Safety**

  Since each segment has its own defined range (`vm_start` to `vm_end`), the system can catch errors like accessing memory out of bounds. This helps prevent crashes, memory corruption, and other bugs.

- **Efficient Memory Use**

  Each segment keeps track of free memory using a free list (`vm_freerg_list`). When a program needs memory, the system can reuse available space instead of always expanding the segment. This reduces memory fragmentation and improves performance.

- **Compatibility with paging**

  The use of multiple segments facilitates the implementation of a paging-based virtual memory system. Each segment can be mapped to physical frames independently, allowing the OS to swap out unused or less critical segments to the SWAP device while keeping frequently accessed segments in RAM. This enhances the efficiency of memory swapping and page table management.

Question:

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

Answer:

- **Before Dividing the Address (Single-Level Paging)**

  The virtual address is divided into two parts:

  - Page number – used to index directly into a single, flat page table.
  - Page offset – used to locate the exact byte inside the physical frame.

  What happens in this case: The page table must have an entry for every page in the virtual address space. If the address space is large, the page table may need millions of entries, taking up lots of memory even if the process uses only a small portion. It's simple and fast (only one lookup), but inefficient in terms of memory usage for sparse processes. Every process needs its own full-size page table, which increases overhead.

- **After Dividing the Address (More-Than-2-Levels Paging)**

  The address is divided into more than two parts. Now, instead of one big page table, we have a hierarchy of page tables.

  For example, in 2-level paging:

  - First part - indexes the first-level page table (page directory).
  - Second part - indexes the second-level page table (page table).
  - Third part (offset) - locates the exact byte in the physical frame.

  What changes: The system no longer needs to allocate one giant page table at once. Instead, page tables are created only as needed, leading to more efficient memory usage. This approach is particularly beneficial when processes utilize only a small portion of their address space, as it avoids wasting memory on unused entries. However, each memory access may require multiple lookups through different levels of page tables, which can slow down performance.

Question:

What are the advantages and disadvantages of segmentation with paging?

Answer:

To understand the pros and cons of segmentation with paging, we need to know how it manages process memory. As described, each process's memory space is divided into multiple segments (such as code, heap, and stack), represented by `vm_area_struct`. Each of these segments spans a virtual address range (`vm_start` to `vm_end`) and tracks usable used and unused space. Instead of allocating large blocks of continuous memory, each segment is broken into small fixed-size pages. These virtual pages are then mapped to physical frames using a page table (pgd), which translates virtual addresses to physical addresses. If a page is not currently in RAM, it can be swapped in from the SWAP device, allowing the system to use more memory than physically available.

- **Advantages of This Approach**
  - Management: Segmentation separates memory by function (code, heap, stack), making it easier to manage and apply access rules.
  - Memory Use: Paging avoids the need for large contiguous memory blocks by allocating memory only as needed.
  - Memory Protection: The OS can detect illegal memory access since each segment has clear boundaries and each page is individually controlled.
  - Support for Virtual Memory: Pages not currently in RAM can be swapped in from secondary storage, allowing processes to use more memory than physically available.
- **Disadvantages**
  - Increased Complexity: Managing both segments and pages requires more data structures and logic, making the system harder to implement and debug.
  - Slow Address Translation: Translating a virtual address involves multiple steps, including segment lookup and page table access, which adds overhead.
  - Page Fault Overhead: If a page is not in RAM, the system must swap it in, which causes a delay. Frequent page faults can degrade performance.

  - Internal Fragmentation: While paging removes external fragmentation, internal fragmentation can still occur within segments if small allocations are made or memory is frequently freed.

3.2 Show the status of RAM

Input:

6	2	4		
1048576	16777216	0	0	0
0	*p0s*	0		
2	*p1s*	15		
4	*p1s*	0		
6	*p1s*	0		

- Simulation runs for 6 time units, on 2 CPUs, with a max of 4 processes.
- 1048576 16777216 0 0 0 → RAM = 1048576 bytes (1MB), `SWAP_0` = 16 MB, `SWAP_1`, `SWAP_2` and `SWAP_3` are not used
- Processes arriving:
  - At time 0: load p0s with priority 0
  - At time 2: load p1s with priority 15
  - At time 4: load p1s again with priority 0
  - At time 6: load p1s again with priority 0

Output:

```
    Time slot   0
ld_routine
        Loaded a process at input/proc/p0s, PID: 1 PRIO: 0
Time slot   1
        CPU 1: Dispatched process  1
Examining free region: 0 - 0 (size 0)
Time slot   2
        Loaded a process at input/proc/p1s, PID: 2 PRIO: 15
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=1 - Region=0 - Address=00000000 - Size=300 byte
print_pgtbl: 0 - 512
00000000: 80000001
00000004: 80000000
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
================================================================
Examining free region: 0 - 0 (size 0)
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=1 - Region=4 - Address=00000000 - Size=300 byte
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
```

```
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
        CPU 0: Dispatched process  2
Time slot   3
Time slot   4
        Loaded a process at input/proc/p1s, PID: 3 PRIO: 0
===== PHYSICAL MEMORY AFTER DEALLOCATION =====
PID=1 - Region=0
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
Examining free region: 0 - 300 (size 300)
Allocated region: 0 - 100 (size 100)
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=1 - Region=1 - Address=00000000 - Size=100 byte
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
Time slot   5
Time slot   6
===== PHYSICAL MEMORY AFTER WRITING =====
write region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
        Loaded a process at input/proc/p1s, PID: 4 PRIO: 0
Time slot   7
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  3
Time slot   8
Time slot   9
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  4
Time slot  10
Time slot  11
Time slot  12
        CPU 1: Put process  3 to run queue
        CPU 1: Dispatched process  1
===== PHYSICAL MEMORY AFTER READING =====
read region=1 offset=20 value=100
print_pgtbl: 0 - 1024Time slot  13

00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000114: 100
Time slot  14
===== PHYSICAL MEMORY AFTER WRITING =====
write region=2 offset=20 value=102
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000114: 100
Time slot  15
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  3
```

```
===== PHYSICAL MEMORY AFTER READING =====
read region=2 offset=20 value=102
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000114: 102
Time slot  16
===== PHYSICAL MEMORY AFTER WRITING =====
write region=3 offset=20 value=103
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000114: 102
Time slot  17
===== PHYSICAL MEMORY AFTER READING =====
read region=3 offset=20 value=103
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000114: 103
Time slot  18
Time slot  19
        CPU 0: Processed  3 has finished
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  1
```

```
===== PHYSICAL MEMORY AFTER DEALLOCATION =====
PID=1 - Region=4
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
        CPU 0: Dispatched process  4
Time slot  20
        CPU 1: Processed  1 has finished
        CPU 1: Dispatched process  2
Time slot  21
Time slot  22
Time slot  23
        CPU 0: Processed  4 has finished
        CPU 0 stopped
Time slot  24
Time slot  25
        CPU 1: Processed  2 has finished
        CPU 1 stopped
```

Explanation:

- System Startup (Time Slot 0): The system initializes, and process PID 1 is loaded into memory.
- CPU Dispatch (Time Slot 1): Process 1 (PID 1) is dispatched to CPU 1 for execution, and process PID 2 is loaded into memory with higher priority.
- Load Process and Dispatch CPU (Time Slot 2): Process 2 (PID 2) is loaded, and dispatched to CPU 0 for execution.
- Execute Memory Instructions for PID 1 (Time Slots 3-4): The system starts executing memory instructions for Process 1 (PID 1), performing read/write operations in memory.
- Execute Memory Instructions for PID 1 (Time Slots 5-6): Process 1 continues executing memory instructions, involving additional memory writes/reads.
- CPU Dispatch (Time Slots 7-8): Process 1 is put back into the run queue, and Process 3 (PID 3) is dispatched to CPU 1 for execution.
- Execute Memory Instructions for PID 1 (Time Slots 9-13): Process 1 resumes execution with additional memory instructions being executed.
- Execute Memory Instructions for PID 1 (Time Slots 14-17): Process 1 continues executing its memory instructions, with read/write operations happening at specified offsets.
- Process Termination:

- Process 3 (PID 3) finishes execution at Time Slot 18.
- Process 1 (PID 1) finishes at Time Slot 21.
- Process 4 (PID 4) finishes at Time Slot 22.
- Process 2 (PID 2) finishes at Time Slot 25.

4 SystemCall

In operating systems, a system call serves as the fundamental mechanism that allows user-space programs to request services provided by the kernel. These may include operations such as memory allocation, process control, or file access. In the “Simple Operating System” described in the assignment, system calls are invoked indirectly via wrapper functions in the standard library, which prepare the arguments and execute a predefined interface to cross the user-kernel boundary safely. Once inside the kernel, a corresponding syscall handler performs the requested operations based on the syscall index. System calls are vital in enforcing protection and abstraction. Since the kernel resides in a privileged mode, allowing user programs direct access would break system isolation and compromise security. Therefore, system calls act as controlled gateways, ensuring that requests are validated and processed safely. This becomes a core mechanism in designing any robust operating system.

4.1 Quetions

Question:
What is the mechanism to pass a complex argument to a system call using the limited registers?
Answer:
As most computer architectures provide only a limited number of CPU registers for passing system call arguments (typically 3 to 6 general-purpose registers), a challenge arises when a system call must receive complex or large arguments, such as arrays, structures, or dynamic strings. To handle such cases, modern operating systems—and the Simple Operating System in this assignment—follow a standard convention: instead of passing the full argument directly through registers, a pointer (memory address) to the complex object in user space is passed via a single register. Inside the kernel, the system call handler uses this pointer to dereference and access the actual data. This pointer-passing mechanism reduces register pressure and enables the transfer of arbitrarily complex structures. However, it necessitates that the kernel performs validation checks on the memory address to ensure that it references a valid and authorized region, thus preventing illegal memory access and maintaining system security. In the assignment, this is mirrored in the way registers carry references to regions in memory where process ID, file paths, or command names are stored, such as when using the killall system call.

Question:
What happens if the syscall job implementation takes too long execution time?
Answer:
If the execution of a system call takes excessively long, it can lead to performance degradation, system unresponsiveness, and fairness issues in process scheduling. In the context of a single-threaded or cooperative kernel as in the Simple Operating System simulation, long-running system calls may block the entire kernel thread or CPU from serving other processes. This monopolization prevents the timely execution of other ready processes, which is especially harmful in multi-process or multi-user environments. Additionally, if the system call interacts with shared resources, prolonged execution may heighten the risk of race conditions or deadlocks, especially when no synchronization mechanisms (such as locking) are in place. In real-world operating systems, this issue is often mitigated by designing system calls to be preemptible, interruptible, or split into smaller phases that allow temporary yielding of CPU control. However, in this simple simulation model, if a syscall such as memory swapping or process termination consumes significant time, there is a lack of built-in mechanisms to detect or manage the time cost dynamically.

Consequently, care must be taken by the developer to ensure that syscall implementations run efficiently and complete promptly to preserve system responsiveness and fairness in scheduling.

4.2 Implementation

4.2.1 System Call for Terminating Processes by Name

```
    int __sys_killall(struct pcb_t *caller, struct sc_regs* regs)
{
    printf("===== EXECUTING SYS_KILLALL =====\n");

    char proc_name[100];
    uint32_t data;
    uint32_t memrg = regs->a1;

    // Get name of the target proc
    int i = 0;
    data = 0;
    while(data != -1){
        libread(caller, memrg, i, &data);
        proc_name[i]= data;
        if(data == -1) proc_name[i]='\0';
        i++;
    }
    proc_name[i] = '\0';
    printf("The procname retrieved from memregionid %d is \"%s\"\n", memrg,
        proc_name);

    pthread_mutex_lock(&lock);
    int terminated_count = 0; // Count of terminated processes
    int list_visited[1000] = {-1}; // Track already processed PIDs in running list
    int queue_visited[1000] = {-1}; // Track already processed PIDs in queue

    if (caller->running_list) {
        struct queue_t *running = caller->running_list;

        if (running->size > 0) {
            for (i = running->size - 1; i >= 0; i--) {
                struct pcb_t *proc = running->proc[i];

                // Validate process
                if (proc == NULL || proc->pid >= 1000 ||
                    list_visited[(int)proc->pid] == 1) continue;

                // printf("[CHECKING IN RUNNING LIST] Process with PID: %d -
                    Name: \"%s\"\n", proc->pid, get_proc_name(proc->path));

                // Now check for name match
                if (strcmp(get_proc_name(proc->path), proc_name) == 0) {
                    // Mark as visited
                    list_visited[proc->pid] = 1;

                    // Remove from queue
                    int j;
                    for (j = i; j < running->size - 1; j++) {
                        running->proc[j] = running->proc[j + 1];
                    }
                    running->size--;
                    terminated_count++;

                    printf("[TERMINATED FROM RUNNING LIST] Process with PID: %d -
                        Name: \"%s\"\n", proc->pid, get_proc_name(proc->path));
                    // free(proc);
                }

                // printf("List size: %d\n", running->size);
            }
        }
    }

#ifdef MLQ_SCHED
    if(caller->mlq_ready_queue) {
        for(int prio = 0; prio < MAX_PRIO; prio++) {
            struct queue_t *mlq_queue = &(caller->mlq_ready_queue[prio]);

            for(int m = mlq_queue->size - 1; m >= 0; m--) {
                struct pcb_t *proc = mlq_queue->proc[m];

                if (!proc || queue_visited[(int)proc->pid] == 1) continue;

                // printf("[CHECKING IN MLQ_READY_QUEUE] Process with PID: %d -
                    Name: \"%s\"\n", proc->pid, get_proc_name(proc->path));

                if(strcmp(get_proc_name(proc->path), proc_name) == 0) {
                    // Mark as terminated
                    proc->is_terminated = 1;
                    queue_visited[proc->pid] = 1;
                    terminated_count++;
                    mlq_queue->size--;

                    printf("[TERMINATED FROM MLQ_READY_QUEUE] Process with PID:
                        %d - Name: \"%s\"\n", proc->pid,
                        get_proc_name(proc->path));
                    free(proc);
                }

                // printf("Queue size: %d\n", mlq_queue->size);
            }
        }
    }

#else
    // For non-MLQ scheduler
    if(caller->ready_queue) {
        for(int m = caller->ready_queue->size - 1; m >= 0; m--) {
            struct pcb_t *proc = caller->ready_queue->proc[m];

            if (!proc || queue_visited[proc->pid]) continue;

            printf("[CHECKING IN READY_QUEUE] Process with PID: %d - Name:
                \"%s\"\n", proc->pid, get_proc_name(proc->path));

            if(strcmp(get_proc_name(proc->path), proc_name) == 0) {
                proc->is_terminated = 1;
                queue_visited[(int)proc->pid] = 1;
                terminated_count++;
                caller->ready_queue->size--;

                printf("[TERMINATED FROM READY_QUEUE] Process with PID: %d -
                    Name: \"%s\"\n", proc->pid, get_proc_name(proc->path));
                free(proc);
            }

            printf("Queue size: %d\n", mlq_queue->size);
        }
    }
#endif

    pthread_mutex_unlock(&lock);
    printf("Terminated %d processes with name: \"%s\"\n", terminated_count,
        proc_name);

    return terminated_count;
}
```

4.2.2 Test Running

4.2.2.a Test case 1: os_sc

Input:

```
  2       1        1
2048  16777216     0    0    0
  9      sc3       15
```

In this simulation input, we verify the basic functionality of the newly implemented system call handler (sys_xxxhandler) at syscall number 440. The system is configured with a 2-second time slice, 1 CPU, and 1 process. The physical memory includes 2048 bytes of RAM and 16MB of SWAP space. At simulation time 8 seconds, a process based on the program "sc3" is loaded with a live priority of 15. The sc3 program is designed to invoke syscall 440, and its purpose is to test whether the system properly routes the syscall and handles the parameter passing inside the custom handler code, not to validate real system call services. Success is determined if the handler processes parameters and prints them correctly without error.

Output:

```
    Time slot   0
ld_routine
Time slot   1
Time slot   2
Time slot   3
Time slot   4
Time slot   5
Time slot   6
Time slot   7
Time slot   8
        Loaded a process at input/proc/sc3, PID: 1 PRIO: 15
        CPU 0: Dispatched process  1
libsyscall: (null)
Syscall handler: nr=440
The first system call parameter 1
Time slot   9
Time slot  10
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Explanation:

- At time slot 0:
  - The system initializes and enters the loading routine (`ld_routine`).
- At time slots 1–7:
  - The system waits; no process is loaded yet.
- At time slot 8:
  - A new process is loaded from `input/proc/sc3`.
  - The process is assigned PID 1 and a priority of 15.
  - CPU 0 dispatches process 1 for execution.
- During process execution:
  - The process invokes a system call with number 440 (`sys_killall`).
  - The syscall handler displays "The first system call parameter 1", indicating successful parameter reading.
- At time slot 9:
  - Process 1 continues running.
- At time slot 10:
  - Process 1 finishes its execution.
  - CPU 0 indicates the process has finished.
- Afterward:
  - CPU 0 stops since there are no more processes to run.

4.2.2.b Testcase 2: os_syscall

Input:

2	1	1		
2048	16777216	0	0	0
10	*sc*2	15		

In this testcase, we set up a test case to validate the behavior of the sys_killall system call when no matching processes exist. The input defines a simulation environment with a 2-second time slice, 1 CPU, and 1 process. The physical memory is configured with 2048 bytes of RAM and 16MB of SWAP space. At simulation time 10 seconds, a process based on the program "sc2" is loaded with a live priority of 15. The sc2 program is designed to invoke sys_killall in order to attempt to terminate processes with the specified name "P0". This verifies the system's ability to handle the case where no processes match the termination criteria.

Output:

```
Time slot   0
ld_routine
Time slot   1
Time slot   2
Time slot   3
Time slot   4
Time slot   5
Time slot   6
Time slot   7
Time slot   8
Time slot   9
Time slot  10
        Loaded a process at input/proc/sc2, PID: 1 PRIO: 15
Time slot  11
        CPU 0: Dispatched process  1
Examining free region: 0 - 0 (size 0)
Time slot  12
Time slot  13
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  14
Time slot  15
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
libsyscall: (null)
Syscall handler: nr=101
===== EXECUTING SYS_KILLALL =====
The procname retrieved from memregionid 1 is "P0"
Terminated 0 processes with name: "P0"
Time slot  16
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Explanation:

- At time slot 0:
  - The system initializes and enters the loading routine (`ld_routine`).
- At time slots 1–9:
  - System continues running; no process has been loaded yet.
- At time slot 10:
  - The process from `input/proc/sc2` is loaded into memory.
  - Assigned PID 1 and priority 15.
- At time slot 11:
  - CPU 0 dispatches process 1.
  - Memory management inspects an initially empty free region (size 0).

- At time slots 12–13:
  - Process 1 continues execution.
  - It is placed back into the run queue and then dispatched again.
- At time slots 14–15:
  - Process 1 is processed in another cycle.
  - The process is again queued and dispatched for execution.
- During time slot 15:
  - Process 1 invokes system call number 101 (`sys_killall`).
  - The syscall handler retrieves the target process name as "P0".
  - No processes matching the name "P0" are found.
  - The kernel reports "Terminated 0 processes".
- At time slot 16:
  - Process 1 finishes its execution.
  - CPU 0 stops after completing all tasks.

4.2.2.c Testcase3: os_syscall_list

Input:

2	1	1		
2048	16777216	0	0	0
9	*sc*1	15		

In this simulation input, we configure the system to test the sys_listsyscall system call, which is responsible for listing all available system calls in the OS. The environment is set up with a 2-second time slice, 1 CPU, and 1 process. The physical memory allocation includes 2048 bytes of RAM and 16MB of SWAP. At simulation time 9 seconds, a process based on the program "sc1" is loaded with a priority of 15. The sc1 program is specifically designed to invoke syscall number 0 (sys_listsyscall) to print out the registered system calls. This test case helps validate that the system call listing function operates correctly and that basic syscall invocation and process scheduling work without issues.

Output:

```
    Time slot   0
ld_routine
Time slot   1
Time slot   2
Time slot   3
Time slot   4
Time slot   5
Time slot   6
Time slot   7
Time slot   8
        Loaded a process at input/proc/sc1, PID: 1 PRIO: 15
```

```
Time slot   9
Time slot  10
        CPU 0: Dispatched process  1
libsyscall: 0-sys_listsyscall
Syscall handler: nr=0
0-sys_listsyscall
17-sys_memmap
101-sys_killall
440-sys_xxxhandler
Time slot  11
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Explanation:

- At time slot 0:
  - The system initializes and enters the loading routine (`ld_routine`).
- At time slots 1–7:
  - The system remains idle; no process is loaded yet.
- At time slot 8:
  - The process `sc1` is loaded from `input/proc/sc1`, assigned PID 1 and priority 15.
- At time slots 9–10:
  - CPU 0 dispatches process 1 for execution.
- During execution:
  - The process invokes syscall number 0 corresponding to `sys_listsyscall`.
  - The system call displays the list of registered system calls including:
    - 0 - sys_listsyscall
    - 17 - sys_memmap
    - 101 - sys_killall
    - 440 - sys_xxxhandler
- At time slot 11:
  - Process 1 completes its execution.
  - CPU 0 stops, as there are no more processes left in the system.

5 Analyze results

5.1 Quetions

Question:

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.

Answer:

- If synchronization is not handled in the Simple OS, the system becomes vulnerable to race conditions and inconsistent behavior due to concurrent access to shared resources. Since the system is designed to run on multiple processors, it may allow multiple processes to operate in parallel. During this parallel execution, different CPUs might attempt to read or write to the same shared data structures—such as ready queues, memory page tables, process control blocks (PCBs), or free frame lists in the memory manager—without any form of mutual exclusion.

- Without synchronization, two CPUs might concurrently attempt to dequeue processes from the same priority queue or modify its structure. If locking mechanisms like spinlocks or mutexes are not used, one CPU might modify the queue while another is in the middle of reading or writing to it, possibly leaving the queue in an invalid or corrupted state. This could result in a process being skipped, scheduled multiple times, or lost—creating inaccurate CPU behavior and affecting fairness.

- A concrete example can be illustrated through the interaction between memory allocation and scheduling. Suppose that two processes are allocated CPU time on separate processors and both issue an ALLOC instruction to request memory. Since memory allocation involves accessing and modifying the system-wide free frame list (or memory region list), the lack of synchronization can lead to both processes being assigned the same physical memory address. This results in overlapping memory usage, where one process could overwrite data belonging to another—a direct violation of process isolation. Such behavior is difficult to detect and can break the correctness of both simulations and results.

- In another case, if the killall system call is invoked to terminate processes based on a particular program name, and this call accesses or modifies PCB structures concurrently with the scheduler or CPU dispatcher, unsynchronized access might lead to use-after-free or null pointer dereference errors. This could cause the simulation to crash or behave inconsistently when comparing against reference outputs.

6 Conclusion

Through the process of completing this assignment, we gained a deeper understanding of how a basic operating system functions. Specifically, we learned about the key components that form the foundation of an OS: process scheduling, memory management, and system call handling. Implementing the Multilevel Queue (MLQ) scheduling policy allowed us to explore how the operating system selects which process to run next, while managing different priority levels. This gave us insight into how real-world systems achieve fairness and responsiveness across multiple CPUs. Furthermore, working on paging-based virtual memory management helped us understand how operating systems allocate and translate memory addresses in a way that isolates processes while efficiently using physical RAM. Finally, we learned how system calls act as a communication bridge between user-level programs and the kernel, providing controlled access to system resources.

However, during the implementation, we encountered several challenges that required careful thought and debugging. One of the earliest difficulties was managing the scheduler queues in a multilevel structure while adhering to the time slice rules. To solve this, we had to understand how to properly enqueue and dequeue processes and ensure the CPU switched tasks correctly. This part laid the groundwork for the rest of the simulation. The next challenge was in memory management, especially during memory allocation and paging. Here, the complexity of simulating virtual to physical address mapping required precise tracking of page tables and frame allocations. We resolved this by breaking down each operation—such as ALLOC and READ—into clear steps and testing them separately before integrating them.

In conclusion, this assignment not only improved our coding and debugging skills but also provided meaningful experience in understanding how different OS subsystems interact. The knowledge we gained reinforces core operating system concepts while teaching us the practical difficulties of building a real-time, multi-CPU system. Despite the challenges, by working step-by-step, testing logically, and applying theoretical knowledge to design decisions, we were able to build a working and stable simulation of a simple but functional operating system.