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

OPERATING SYSTEM (CO2018)

Assignment

"Simple Operating System"

Instructor: Diệp Thanh Đăng, *CSE-HCMUT*

Students: Ngô Trần Đình Duy - 2352179 *(Group CC10 - Team 3, **Leader**)*
Nguyễn Duy Nhật Tuệ - 2350030 *(Group CC10 - Team 3)*
Nguyễn Hoàng Danh - 2352157 *(Group CC10 - Team 3)*
Lê Hoàng Chí Vĩ - 2353336 *(Group CC10 - Team 3)*

HO CHI MINH CITY, APRIL 2025

Contents

List of Figures **4**

List of Tables **4**

Member list & Workload **4**

1 Introduction **5**

2 Scheduler **6**
2.1 Design Pattern . . . 6
2.2 Real Test & Result Interpretation . . . 6
2.3 Code Implementation . . . 10
2.4 Questions and Answers . . . 12

3 Memory Management **14**
3.1 Memory Architecture Overview . . . 14
3.2 Memory Operations . . . 14
3.2.1 Reading Memory . . . 14
3.2.2 Writing Memory . . . 14
3.2.3 Library Functions . . . 15
3.3 Calculating Frame Number . . . 15
3.3.1 Frame Number Extraction . . . 15
3.3.2 From the Source Code . . . 15
3.4 Memory Management Logic . . . 16
3.4.1 System Call Arguments . . . 16
3.4.2 Code Implementation . . . 16
3.5 Swapping Technique . . . 17
3.5.1 How It Works . . . 17
3.5.2 Implementation Details . . . 18
3.5.3 Handling Edge Cases . . . 19
3.6 Memory Status Monitoring . . . 20
3.7 Real Test & Result Interpretation . . . 21
3.7.1 Input Configuration . . . 21
3.7.2 System Execution . . . 22
3.8 Questions and Answers . . . 35
3.8.1 Question 1: . . . 35
3.8.2 Answer: . . . 35
3.8.3 Question 2: . . . 35

3.8.4 Answer: . . . 36
3.8.5 Question 3: . . . 36
3.8.6 Answer: . . . 36

4 System Call **38**
4.1 System Call Architecture . . . 38
4.1.1 System Call Table . . . 38
4.1.2 System Call Dispatcher . . . 38
4.1.3 Register Structure . . . 39
4.1.4 User-Level Interface . . . 39
4.2 Implemented System Calls . . . 40
4.2.1 Memory Mapping (syscall 17) . . . 40
4.2.2 Operation Types . . . 41
4.2.3 List System Calls (syscall 100) . . . 42
4.2.4 Kill All Processes (syscall 101) . . . 42
4.2.5 Not-Implemented System Call Handler . . . 44
4.3 Real Test & Result Interpretation . . . 45
4.3.1 Input Configuration . . . 45
4.3.2 System Execution . . . 46
4.4 Questions and Answers . . . 46

5 Put It All Together **48**

References **50**

List of Figures

List of Tables

1 Member list & workload 4
2 Various CPU address bus configurations 35

Member list & Workload

No.	Fullname	Student ID	Problems	% done
1	Ngô Trần Đình Duy	2352179	- Memory	100%
2	Nguyễn Duy Nhật Tuệ	2350030	- System Call	100%
3	Nguyễn Hoàng Danh	2352157	- Scheduling	100%
4	Lê Hoàng Chí Vĩ	2353336	- LaTeX	100%

Table 1: Member list & workload

1 Introduction

Source code: https://github.com/darealDanh/os

As technology advances, the amount of information and tasks that need to be processed gets larger and more complex, leading to hardware that cannot handle those tasks. Therefore, operating systems are designed to solve the problems of scheduling and managing resources and tasks. In this exercise, we will study the two main components to implement a simple operating system, including the Scheduler and Paging-based Memory Management. The major assignment focuses on designing a simple operating system based on these two main components:

- **Scheduler**: is a scheduling algorithm that determines which process is executed on the specified CPU.
- **Memory Management**: differentiate the physical memory spaces of processes from each other by having each process own a separate virtual memory space and the system will arrange and transfer the address in the process's virtual memory to the corresponding physical memory address.

Through the assignment, we can better understand the origin and operating principles of a simple operating system, focusing on the contents of Scheduling, Memory Management and Synchronization. With the knowledge learned through the assignment and lectures, group has successfully designed a simple operating system that will be introduced in more detail in the following contents.

2 Scheduler

2.1 Design Pattern

- **Multi-level Queue Scheduling and Round Robin**
  - Processes are organized in multiple ready queues based on their priority
  - Within each priority level, round-robin scheduling is implemented.
- **Process Priority**
  - Processes with higher priority (lower numerical value) are scheduled first
  - This ensures critical operations receive CPU time before less important ones.
- **Time Slot Allocation**
  - Each priority level has allocated slots based on the equation:
    $slot[i] = MAX\_PRIO$ - i
  - Higher priority processes (lower i) receive more consecutive time slots
  - This prevents starvation while still prioritizing important processes.
- **Multi-CPU Support**
  - Multiple CPUs can run processes concurrently
  - Improves overall system throughput and responsiveness
  - Allows parallel execution of processes from different priority levels.
- **Dual Priority Mechanism**
  - Each process has a default priority in its description file.
  - This priority can be overwritten by the configuration specified in the input file.
  - When both exist, the input configuration priority takes precedence.
  - This provides flexibility in process management without modifying process files.

2.2 Real Test & Result Interpretation

sched input:

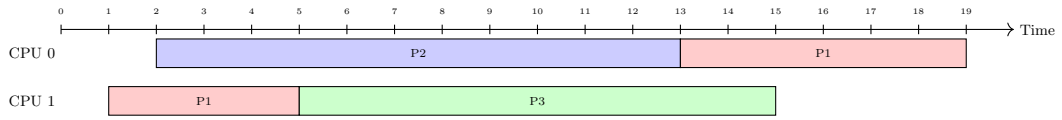
System Parameters

Parameter	Value	Description
Time slice	4	Each process executes for 4 time units
Number of CPUs	2	Two processors running in parallel
Number of processes	3	Total processes to be scheduled

Process Information

Start Time	Name	Priority	Notes
0	p1s (P1)	1	Medium priority, starts first
1	p2s (P2)	0	High priority, arrives second
2	p3s (P3)	0	High priority, arrives last

Gantt Chart 1



Explanation 1

Time 0–2: Load Phase

- **Time 0**: System initializes. Process P1 (priority 1) arrives and is loaded.
- **Time 1**: CPU 1 dispatches P1. Process P2 (priority 0) arrives and is loaded.
- **Time 2**: CPU 0 dispatches P2. Process P3 (priority 0) arrives and is loaded.

Time 3–12: Execution Phase (Part 1)

- **Time 5**:
  - P1 completes its time slice on CPU 1 and returns to the ready queue.
  - P3 (priority 0) is dispatched on CPU 1.
  - P2 (priority 0) is redispatched on CPU 0 after completing its time slice.
- **Time 8**:
  - P3 completes its time slice on CPU 1 and returns to the ready queue.
  - P3 is redispatched on CPU 1 (round-robin among priority 0 processes).
  - P2 is redispatched on CPU 0 after another time slice.

Time 13–16: Execution Phase (Part 2)

- **Time 13**:
  - P2 finishes execution completely on CPU 0.
  - P1 (priority 1) is dispatched on CPU 0.
  - P3 continues executing on CPU 1.

- **Time 16**: P3 finishes execution completely on CPU 1. CPU 1 becomes idle.

Time 17–19: Completion Phase

- **Time 17**: P1 completes another time slice on CPU 0 and returns to the ready queue.
- **Time 17–19**: P1 is redispatched and completes execution on CPU 0.
- **Time 19**: All processes have completed. CPU 0 becomes idle.

sched_0 input

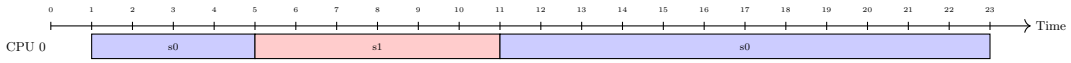
System Parameters

Parameter	Value	Description
Time slice	2	Each process executes for 2 time units
Number of CPUs	1	Single process system
Number of processes	2	Total processes to be scheduled

Process Information

Start Time	Name	Priority	Notes
0	s0	4	Lowest priority, starts first
4	s1	0	High priority, arrives second

Gantt Chart 2



Explanation 2

- **Time 0-3:**
  - Low priority process s0 starts execution at time 1 (after loading in time 0)
  - It runs for 3 time units (slots 1-3)
- **Time 4-11:**
  - High priority process s1 arrives at time 4
  - s1 preempts s0 and runs from time 5-11 (7 time units)
  - s1 completes execution at the end of time 11

- **Time 12-23:**
  - s0 returns to finish its remaining execution (12 more time units)
  - s0 completes execution at the end of time 23

sched_1 input

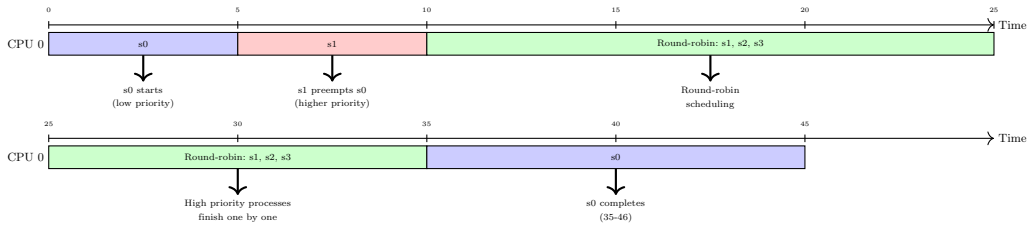
System Parameters

Parameter	Value	Description
Time slice	2	Each process executes for 2 time units
Number of CPUs	1	Single processor system
Number of processes	4	Total processes to be scheduled

Process Information

Start Time	Name	Priority	Notes
0	s0	4	Lowest priority, starts first
4	s1	0	Highest priority, arrives 2nd
6	s2	0	Highest priority, arrives 3rd
7	s3	0	Highest priority, arrives 4th

Gantt Chart 3



Explanation 3

- Time 0-4:
  - Process s0 (priority 4) starts execution.
  - Executes from time 1-3.
- Time 4-8:
  - Process s1 (priority 0) arrives at time 4.

- Process s2 (priority 0) arrives at time 6.
- Process s3 (priority 0) arrives at time 7.

- Time 8-35:
  - Round-robin scheduling among high-priority processes.
  - Each gets exactly 2 time units (the configured time slice).
  - Processes complete in order:
    - s1 completes at time 24.
    - s2 completes at time 34.
    - s3 completes at time 35.

- Time 36-46:
  - Process s0 (priority 4) resumes execution.

2.3 Code Implementation

- **Dual Mechanism:** Allows process priorities to be set in two different ways.

Dual Mechanism

```
// In loader.c - Default priority assignment
#ifdef MLQ_SCHED
proc->prio = DEFAULT_PRIO;
#endif

// In os.c - Priority override during loading
struct pcb_t *proc = load(ld_processes.path[i]);
#ifdef MLQ_SCHED
proc->prio = ld_processes.prio[i];  // Override with configuration priority
#endif
```

- **enqueue:** Put a new process at the end of the queue.

Enqueue Functions

```c
void enqueue(struct queue_t *queue, struct pcb_t *proc) {
    // Verify valid queue and process
    if (queue == NULL || proc == NULL) return;

    // Check if queue is full
    if (queue->size == MAX_QUEUE_SIZE) {
        return;
    }

    // Add process to the end of the queue
    queue->proc[queue->size] = proc;
    queue->size++;
}
```

- **dequeue:** Take out the first process at the start of the queue (highest priority).

Dequeue Function

```c
// Remove and return first process from queue
struct pcb_t* dequeue(struct queue_t *queue) {
    if (queue == NULL || queue->size == 0) {
        return NULL;
    }

    struct pcb_t* proc = queue->proc[0];
#ifdef MLQ_SCHED
    // Shift remaining processes forward
    for (int i = 0; i < queue->size - 1; i++) {
        queue->proc[i] = queue->proc[i + 1];
    }

    queue->size--;
    return proc;
#else
    ...
#endif
}
```

- **get_mlq_proc:** Get a process from PRIORITY[ready_queue]

MLQ Process Reader

```c
struct pcb_t * get_mlq_proc(void) {
    struct pcb_t * proc = NULL;
    pthread_mutex_lock(&queue_lock);
    int i;

    // Core MLQ scheduling algorithm
    for (i = 0; i < MAX_PRIO; ++i) {
        // Skip empty queues or those that used up their slots
        if (empty(&mlq_ready_queue[i]) || slot[i] == 0) {
            // Reset slot count when depleted
            slot[i] = MAX_PRIO - i;
            continue;
        }

        // Get process from this priority level
        proc = dequeue(&mlq_ready_queue[i]);
        // Decrement remaining slots for this priority
        slot[i]--;
        break;
    }

    pthread_mutex_unlock(&queue_lock);
    return proc;
}
```

2.4 Questions and Answers

Question:

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

Answer:

Multi-Level Queue (MLQ) scheduling with Round-Robin is used because of many reasons:

- **Priority-Based Process Management: MLQ** organizes processes into separate queues with fixed priority levels. Higher priority processes (lower numerical values) always execute before lower-priority ones, ensuring critical tasks receive immediate CPU attention.
- **Fair Intra-Queue Scheduling:** Within each priority level, **Round Robin (RR)** scheduling is applied, giving each process a fixed time slice before moving to the next process of equal priority. This prevents any single process from monopolizing CPU time within its priority class.

- **Adaptive Time Allocation:** Higher priority queues receive more consecutive time slots based on the formula `slot[i] = MAX_PRIO - i`. This allows important processes to complete more work before yielding, while still ensuring lower-priority processes eventually execute when higher priority queues are empty.

while other scheduling algorithms, particularly conventional scheduling algorithms has many limitations:

- **First Come First Served (FCFS):** Implements a **non-preemptive scheduling model**, meaning once a process is allocated to the CPU, it runs to completion without interruption. This can result in the **"convoy effect"**, where shorter processes are forced to wait behind a long-running process, leading to poor average waiting time and reduced responsiveness for interactive tasks.
- **Shortest Job First (SJF):** While it minimizes average waiting time theoretically, it relies on accurate prediction of the next CPU burst duration—an estimate that is typically not available or reliable in real-world scenarios. Processes with longer CPU bursts may experience indefinite postponement (starvation), particularly when short jobs continuously arrive, violating fairness principles.
- **Round Robin (RR):** Employs **time-slicing** to ensure equitable CPU access across all processes, improving responsiveness for interactive tasks. However, selecting an appropriate time quantum is critical:
  - A **large time slice** causes the algorithm to behave like **FCFS**, undermining responsiveness.
  - A **very small time slice** increases context switching overhead, reducing overall CPU efficiency and throughput.
- **Basic Priority Scheduling:** Schedules processes based on fixed priorities. However, without proper mechanisms, equal-priority processes require additional strategies for fair ordering (e.g., **FIFO** within priority levels). Lower-priority processes may suffer starvation if higher-priority tasks continuously dominate the CPU, unless techniques like **aging** are employed.

3 Memory Management

3.1 Memory Architecture Overview

- Physical Memory is finite, so Virtual Memory is created to increase task capacity and OS performance.
- Physical Memory consists of frames (in both RAM and SWAP) while Virtual Memory uses pages.
- The system uses address mapping from Virtual Memory to Physical Memory.
- Two main memory operations are supported: read (*read_mem*) and write (*write_mem*).

3.2 Memory Operations

3.2.1 Reading Memory

The `read_mem` function handles reading data from memory:

- Takes a virtual address, process context, and pointer to store data
- Translates the virtual address to physical address
- Returns 0 on success, 1 on failure (invalid address)

Read_Mem Functions

```
int read_mem(addr_t address, struct pcb_t * proc, BYTE * data) {
    addr_t physical_addr;
    if (translate(address, &physical_addr, proc)) {
        *data = _ram[physical_addr];
        return 0;
    } else {
        return 1;
    }
}
```

3.2.2 Writing Memory

The `write_mem` function handles writing data to memory:

- Takes a virtual address, process context, and data to write
- Translates the virtual address to physical address
- Returns 0 on success, 1 on failure (invalid address)

Write_Mem Functions

```c
int write_mem(addr_t address, struct pcb_t * proc, BYTE data) {
    addr_t physical_addr;
    if (translate(address, &physical_addr, proc)) {
        _ram[physical_addr] = data;
        return 0;
    } else {
        return 1;
    }
}
```

3.2.3 Library Functions

The system also provides higher-level functions in `libmem.h`:

- `libread`: Reads data from a memory region, handling paging if necessary
- `libwrite`: Writes data to a memory region, handling paging if necessary
- `liballoc`: Allocates memory for a process
- `libfree`: Frees memory allocated to a process

3.3 Calculating Frame Number

3.3.1 Frame Number Extraction

```c
#define PAGING_FPN(x) GETVAL(x,PAGING_FPN_MASK,PAGING_ADDR_FPN_LOBIT)
```

- This macro extracts the Frame Physical Number (FPN) from a page table using bit operations
- The hexadecimal value (like 80000005) is the actual PTE, with the frame number encoded in it according to the defined bit masks in the systems.

3.3.2 From the Source Code

```c
for (pgit = pgn_start; pgit < pgn_end; pgit++) {
  printf("Page Number: %d -> Frame Number: %d\n", pgit, PAGING_FPN(caller->mm->pgd[pgit]));
}
```

- The Frame Number is displayed during page table printing.

3.4 Memory Management Logic

3.4.1 System Call Arguments

- a1: System Operation Type
- a2: Physical Address to write/ read
- a3: Value to write/ read

3.4.2 Code Implementation

- **pg_getpage:** Handles the core page swapping mechanism when a page is not present in physical memory

Page Swapping

```
struct sc_regs regs;
regs.a1 = SYSMEM_SWP_OP; // Operation code for swap
regs.a2 = vicfpn  ; // Source frame number (victim frame)
regs.a3 = swpfpn; // Destination frame number (swap frame)

// syscall(caller, 17, &regs) is __sys_memmap(caller, &regs);
__sys_memmap(caller, &regs); // Perform the swap operation
```

- **pg_getval:** Reads a value from memory at a given virtual address

Page Reader

```
int pg_getval(struct mm_struct *mm, int addr, BYTE *data, struct pcb_t *caller)
{
    // ...
    struct sc_regs regs;
    regs.a1 = SYSMEM_IO_READ;  // Operation type: read
    regs.a2 = phyaddr;         // Physical address to read from
    regs.a3 = (uint32_t)(*data); // Value read from memory

    // syscall(caller, 17, &regs) is __sys_memmap(caller, &regs);
    __sys_memmap(caller, &regs);
    // ...
}
```

- **pg_setval:** Writes a value to memory at a given virtual address

Page Writer

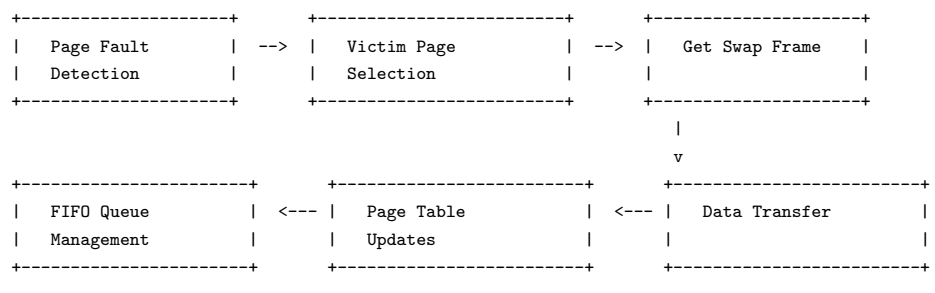
```c
int pg_setval(struct mm_struct *mm, int addr, BYTE value, struct pcb_t *caller)
{
    // ...
    struct sc_regs regs;
    regs.a1 = SYSMEM_IO_WRITE; // Operation type: write
    regs.a2 = phyaddr;         // Physical address to write to
    regs.a3 = value;           // Value to write to memory

    // syscall(caller, 17, &regs) is __sys_memmap(caller, &regs);
    __sys_memmap(caller, &regs);
    // ...
}
}
```

3.5 Swapping Technique

Swapping is a memory management technique that allows the operating system to handle situations where the physical RAM is insufficient to hold all the processes that need to be executed. This critical function enables multitasking and efficient memory utilization in operating systems.

Page Fault Handling Flowchart



3.5.1 How It Works

1. **Basic Principle:**
    - When physical RAM is insufficient, less frequently used memory pages are temporarily moved to secondary storage (swap space).
    - This frees up RAM for more urgent processes or data.
    - Pages can be swapped back into RAM when needed again.
2. **Swap Space:**
    - Dedicated area on disk (or other storage) reserved for storing memory pages.

- Can be a separate partition or a swap file.
- In our implementation, multiple swap devices can be configured with different sizes.

3. **Page Replacement Algorithm:**
   - Our system uses a FIFO (First-In-First-Out) algorithm to select victim pages.
   - The oldest page in memory is selected as the victim to be swapped out.
   - This is tracked using the *fifo_pgn* linked list in the *mm_struct*.

3.5.2 Implementation Details

Page Table Structure

- Each process has a page table (*pgd* array in *mm_struct*)
- Page table entries (PTEs) contain information about whether a page is:
  - Present in physical memory (indicated by *PAGING_PAGE_PRESENT* bit)
  - Swapped out to disk (containing the swap frame number)

Swap Operation Sequence

1. **Page Fault Detection:**
   - When accessing a virtual address, the system checks if the corresponding page is present
   - If not present, a page fault occurs, triggering the swapping mechanism
2. **Victim Selection:**
   - System calls *find_victim_page()* to select a page to be swapped out
   - The selected victim page will give up its physical frame to the requested page
3. **Obtaining Swap Space:**
   - System calls *MEMPHY_get_freefp()* to get a free frame in swap space
   - If swap space is full or not configured, handles the error gracefully
4. **Data Transfer:**
   - Copies data from victim frame in RAM to the allocated swap frame
   - Uses *__swap_cp_page()* to perform the copy operation
5. **Page Table Updates:**

- Updates the victim's PTE to mark it as not present and records its swap location
- Updates the faulting page's PTE to mark it as present and assigns the physical frame

6. **FIFO Queue Management:**
   - Adds the newly loaded page to the FIFO queue with *enlist_pgn_node()*
   - Ensures proper tracking for future victim selection.

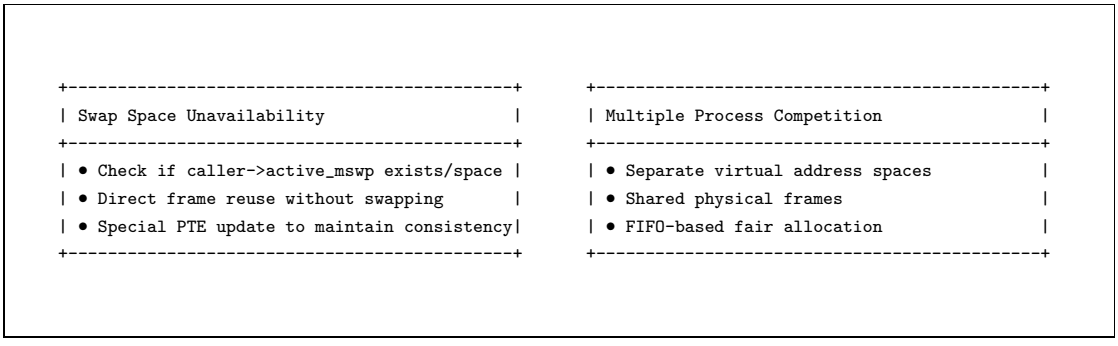
System Call Interaction

- Swapping operations use system call 17 (*sys_memmap*)
- The operation type *SYSMEM_SWP_OP* specifies a swap operation
- Arguments include source and destination frame numbers.

```c
struct sc_regs regs;
regs.a1 = SYSMEM_SWP_OP; // Operation code for swapping
regs.a2 = vicfpn;       // Source frame number (victim)
regs.a3 = swpfpn; // Destination frame number (in swap space)

// syscall(caller, 17, &regs) is __sys_memmap(caller, &regs);
__sys_memmap(caller, &regs); // Perform the swap operation
```

3.5.3 Handling Edge Cases



Swap Space Unavailability

- The implementation includes a check for swap space availability
- If swap space is not configured or is full, the system:
  1. Directly reuses a physical frame without swapping to disk
  2. Marks the victim page as not present
  3. Updates the page table entries accordingly.

Multiple Process Memory Demands

- When multiple processes compete for limited RAM:
  1. Each process gets its own virtual address space
  2. Physical frames are shared among all processes
  3. Swapping ensures fair allocation based on usage patterns.

3.6 Memory Status Monitoring

The system includes functions to monitor memory status during runtime, which are essential for debugging and evaluating memory utilization.

Memory Status Monitoring Functions

Function	Description
`dump()`	Prints detailed information about memory pages and their contents

Example of Memory Status:

Example: Memory Dump and Page Table Usage

```
Time slot   6
        Loaded a process at input/proc/p1s, PID: 4 PRIO: 0
===== 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 =====
BYTE 00000114: 100
===== PHYSICAL MEMORY END-DUMP =====
================================================================
```

The above example demonstrates several key aspects of the memory management system:

1. **Process Loading**: A process with PID 4 and priority 0 is loaded into memory
2. **Memory Writing**: A value of 100 is written to region 1, offset 20
3. **Page Table Structure**: The page table entries (PTEs) are displayed in hexadecimal format
   - `80000001` indicates page 0 is mapped to frame 1
   - The high bit (`8`) indicates the page is present in physical memory
4. **Virtual-to-Physical Mapping**: The system shows the mapping from virtual page numbers to physical frame numbers
   - Virtual page 0 → physical frame 1
   - Virtual page 1 → physical frame 0
   - Virtual page 2 → physical frame 3
   - Virtual page 3 → physical frame 2
5. **Memory Content Verification**: The physical memory dump confirms that byte at address 0x114 contains the value 100

This example illustrates how the `PAGING_FPN` macro extracts the frame number from page table entries, and how the system translates between virtual and physical addresses during memory operations.

3.7 Real Test & Result Interpretation

3.7.1 Input Configuration

The system is configured using an input file (*os_1_mlq_paging*) with the following parameters:

```
4 4 6
65536
1048576 0 0 0
0 p0s 130
1 s3 39
3 m1s 15
5 s2 120
8 m0s 120
9 p1s 15
```

This configuration specifies:

- Time slot: 4 time units
- Number of CPUs: 4
- Number of processes: 6
- Physical RAM size: 65536 bytes
- Swap space size: 1048576 bytes
- Process specifications (start time, name, priority)

3.7.2 System Execution

Initialization and Process Loading

```
Time slot   0
ld_routine
Time slot   1
        Loaded a process at input/proc/p0s, PID: 1 PRIO: 130
Time slot   2
        CPU 3: Dispatched process  1
        Loaded a process at input/proc/s3, PID: 2 PRIO: 39
        CPU 2: Dispatched process  2
```

- Loads process p0s (PID 1) with priority 130
- CPU 3 begins executing process 1
- Loads process s3 (PID 2) with priority 39
- CPU 2 begins executing process 2

Memory Allocation and Page Table Management

```
Time slot   3
Free region: 0 - 0
Free region: 0 - 0
Free region: 0 - 512
===== 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
================================================================
```

- Process 1 allocates 300 bytes in memory region 0
- The page table shows virtual-to-physical mapping:
  - Page 0 maps to Frame 1
  - Page 1 maps to Frame 0
- The high bit (0x8) in page table entries indicates pages are present in physical memory

Memory Allocation for New Processes

```
Time slot   4
        Loaded a process at input/proc/m1s, PID: 3 PRIO: 15
        CPU 3: Put process  1 to run queue
        CPU 3: Dispatched process  3
Free region: 0 - 0
Free region: 0 - 0
Free region: 0 - 512
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=3 - Region=0 - Address=00000000 - Size=300 byte
print_pgtbl: 0 - 512
00000000: 80000003
00000004: 80000002
Page Number: 0 -> Frame Number: 3
Page Number: 1 -> Frame Number: 2
================================================================
        CPU 2: Put process  2 to run queue
        CPU 2: Dispatched process  2
        CPU 0: Dispatched process  1
```

- Process m1s (PID 3) is loaded with priority 15
- Process 1 is returned to the run queue after its time slice expires
- CPU 3 begins executing higher-priority process 3
- Process 3 allocates 300 bytes in memory region 0
- A new page table shows process 3's virtual pages mapped to physical frames 3 and 2

Multiple Region Allocation

```
Free region: 0 - 0
Free region: 300 - 512
Free region: 0 - 0
Free region: 300 - 512
Free region: 512 - 1024
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=1 - Region=4 - Address=00000200 - Size=300 byte
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000005
00000012: 80000004
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 5
Page Number: 3 -> Frame Number: 4
================================================================
```

- The system displays free regions in physical memory
- Process 1 allocates 300 bytes in region 4, starting at address 0x200
- The page table for process 1 is extended to include pages 2 and 3
- These pages map to frames 5 and 4 respectively

Memory Allocation and Deallocation

```
Time slot   5
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=3 - Region=1 - Address=0000012c - Size=100 byte
print_pgtbl: 0 - 512
00000000: 80000003
00000004: 80000002
Page Number: 0 -> Frame Number: 3
Page Number: 1 -> Frame Number: 2
================================================================
===== PHYSICAL MEMORY AFTER DEALLOCATION =====
PID=1 - Region=0
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000005
00000012: 80000004
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 5
Page Number: 3 -> Frame Number: 4
================================================================
```

- Process 3 allocates 100 bytes in region 1 at address 0x12c
- Process 1 deallocates region 0
- Both processes maintain their page table mappings

New Process Loading and Memory Deallocation

```
Time slot   6
        Loaded a process at input/proc/s2, PID: 4 PRIO: 120
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  4
        CPU 3: Put process  3 to run queue
        CPU 3: Dispatched process  3
===== PHYSICAL MEMORY AFTER DEALLOCATION =====
PID=3 - Region=0
print_pgtbl: 0 - 512
00000000: 80000003
00000004: 80000002
Page Number: 0 -> Frame Number: 3
Page Number: 1 -> Frame Number: 2
================================================================
        CPU 2: Put process  2 to run queue
        CPU 2: Dispatched process  2
        CPU 1: Dispatched process  1
```

- Process s2 (PID 4) is loaded with priority 120
- CPUs redistribute processes based on priority and availability
- Process 3 deallocates region 0 but maintains its page table mappings

Memory Allocation and Writing

```
Free region: 0 - 0
Free region: 0 - 512
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=1 - Region=1 - Address=00000000 - Size=100 byte
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000005
00000012: 80000004
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 5
Page Number: 3 -> Frame Number: 4
================================================================
Time slot   7
Free region: 0 - 0
Free region: 0 - 300
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=3 - Region=2 - Address=00000000 - Size=100 byte
print_pgtbl: 0 - 512
00000000: 80000003
00000004: 80000002
Page Number: 0 -> Frame Number: 3
Page Number: 1 -> Frame Number: 2
================================================================
===== PHYSICAL MEMORY AFTER WRITING =====
write region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000005
00000012: 80000004
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 5
Page Number: 3 -> Frame Number: 4
================================================================
```

- Process 1 allocates 100 bytes in region 1 at address 0x0
- In time slot 7, process 3 allocates 100 bytes in region 2 at address 0x0

- Process 1 writes value 100 to region 1, offset 20
- Each process can use the same virtual addresses (0x0) without conflict

Memory Dump and New Process Loading

```
===== PHYSICAL MEMORY DUMP =====
BYTE 00000114: 100
        Loaded a process at input/proc/m0s, PID: 5 PRIO: 120
===== PHYSICAL MEMORY END-DUMP =====
================================================================
Time slot   8
        CPU 3: Put process  3 to run queue
        CPU 3: Dispatched process  3
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  5
```

- Memory dump confirms the value 100 has been written to address 0x114
- Process m0s (PID 5) is loaded with priority 120
- CPUs continue round-robin scheduling within priority levels
- CPU 0 dispatches newly loaded process 5

Memory Allocation for New Process

```
Free region: 0 - 0
Free region: 0 - 0
Free region: 0 - 512
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=5 - Region=0 - Address=00000000 - Size=300 byte
print_pgtbl: 0 - 512
00000000: 80000007
00000004: 80000006
Page Number: 0 -> Frame Number: 7
Page Number: 1 -> Frame Number: 6
================================================================
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  4
        CPU 2: Put process  2 to run queue
        CPU 2: Dispatched process  2
```

- Process 5 allocates 300 bytes in region 0 at address 0x0

- The page table for process 5 maps virtual pages 0 and 1 to physical frames 7 and 6
- CPUs continue scheduling processes based on priority and round-robin within levels

Memory Deallocation and Process Loading

```
===== PHYSICAL MEMORY AFTER DEALLOCATION =====
PID=3 - Region=2
print_pgtbl: 0 - 512
00000000: 80000003
00000004: 80000002
Page Number: 0 -> Frame Number: 3
Page Number: 1 -> Frame Number: 2
================================================================
Time slot   9
Free region: 0 - 0
Free region: 300 - 512
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=5 - Region=1 - Address=0000012c - Size=100 byte
print_pgtbl: 0 - 512
00000000: 80000007
00000004: 80000006
Page Number: 0 -> Frame Number: 7
Page Number: 1 -> Frame Number: 6
================================================================
        Loaded a process at input/proc/p1s, PID: 6 PRIO: 15
```

- Process 3 deallocates region 2 but maintains its page table mappings
- Process 5 allocates 100 bytes in region 1 at address 0x12c
- Process p1s (PID 6) is loaded with priority 15 (higher priority)

Process Deallocation and CPU Redistribution

```
===== PHYSICAL MEMORY AFTER DEALLOCATION =====
PID=3 - Region=1
print_pgtbl: 0 - 512
00000000: 80000003
00000004: 80000002
Page Number: 0 -> Frame Number: 3
Page Number: 1 -> Frame Number: 2
================================================================
Time slot  10
        CPU 3: Processed  3 has finished
        CPU 3: Dispatched process  6
        CPU 1: Put process  4 to run queue
        CPU 1: Dispatched process  4
        CPU 0: Put process  5 to run queue
        CPU 0: Dispatched process  5
```

- Process 3 deallocates region 1 before completion
- Process 3 finishes execution in time slot 10
- CPU 3 now dispatches process 6 (priority 15)
- CPUs 1 and 0 continue round-robin scheduling for their processes

Memory Deallocation and New Process Loading

```
===== PHYSICAL MEMORY AFTER DEALLOCATION =====
PID=5 - Region=0
print_pgtbl: 0 - 512
00000000: 80000007
00000004: 80000006
Page Number: 0 -> Frame Number: 7
Page Number: 1 -> Frame Number: 6
================================================================
        CPU 2: Put process  2 to run queue
        CPU 2: Dispatched process  2
Time slot  11
Free region: 0 - 0
Free region: 0 - 300
        Loaded a process at input/proc/s0, PID: 7 PRIO: 38
```

- Process 5 deallocates region 0 but maintains its page table mappings

- The system reports free memory regions available for allocation
- Process s0 (PID 7) is loaded with priority 38

Memory Allocation for Process 5

```
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=5 - Region=2 - Address=00000000 - Size=100 byte
print_pgtbl: 0 - 512
00000000: 80000007
00000004: 80000006
Page Number: 0 -> Frame Number: 7
Page Number: 1 -> Frame Number: 6
================================================================
Time slot  12
        CPU 2: Put process  2 to run queue
        CPU 2: Dispatched process  7
        CPU 1: Put process  4 to run queue
        CPU 0: Put process  5 to run queue
        CPU 3: Put process  6 to run queue
```

- Process 5 allocates 100 bytes in region 2 at address 0x0
- The system maintains process 5's page table mappings to frames 7 and 6
- In time slot 12, CPU 2 begins executing newly loaded process 7 (priority 38)
- All processes take turns based on priority and round-robin scheduling

CPU Dispatching Based on Priority

```
        CPU 3: Dispatched process  6
        CPU 0: Dispatched process  4
        CPU 1: Dispatched process  2
Time slot  13
        CPU 1: Processed  2 has finished
        CPU 1: Dispatched process  5
```

- Process 2 completes execution in time slot 13
- CPU 1 immediately dispatches process 5 to maintain CPU utilization

Page Swapping and Memory Operations

```
Time slot  14
Memop code: 7
Memop code: 0
===== PHYSICAL MEMORY AFTER WRITING =====
write region=2 offset=1000 value=1
print_pgtbl: 0 - 512
00000000: c0000000
00000004: 80000006
Page Number: 0 -> Frame Number: 0
Page Number: 1 -> Frame Number: 6
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000114: 100
BYTE 00000640: 102
BYTE 000007e8: 1
===== PHYSICAL MEMORY END-DUMP =====
================================================================
```

- Write operation to region 2, offset 1000, value 1
- Page table entry changed to `c0000000` - the high bit (c) indicates this page has been swapped

Reading from Memory

```
Time slot  15
        CPU 1: Processed  5 has finished
        CPU 1: Dispatched process  1
===== PHYSICAL MEMORY AFTER READING =====
read region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000005
00000012: 80000004
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 5
Page Number: 3 -> Frame Number: 4
================================================================
```

- Process 1 reads from region 1, offset 20, obtaining value 100

Writing to Memory

```
Time slot  16
        Loaded a process at input/proc/s1, PID: 8 PRIO: 0
        CPU 2: Put process  7 to run queue
        CPU 2: Dispatched process  8
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  7
        CPU 3: Put process  6 to run queue
        CPU 3: Dispatched process  6
===== PHYSICAL MEMORY AFTER WRITING =====
write region=2 offset=20 value=102
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000005
00000012: 80000004
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 5
Page Number: 3 -> Frame Number: 4
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000114: 102
BYTE 00000640: 102
BYTE 000007e8: 1
===== PHYSICAL MEMORY END-DUMP =====
================================================================
```

- Process s1 (PID 8) is loaded with highest priority (0)
- Multiple CPUs redistribute their workload
- A write operation changes address 0x114 to value 102

Process Completion and Memory Deallocation

```
Time slot  22
Time slot  23
        CPU 3: Put process  8 to run queue
        CPU 3: Dispatched process  8
        CPU 1: Put process  7 to run queue
        CPU 2: Put process  1 to run queue
        CPU 2: Dispatched process  1
        CPU 1: Dispatched process  7
===== 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
================================================================
```

- Process 8 completes execution and CPU 2 stops
- Process 1 deallocates memory region 4
- Page table entries remain mapped as the frames may still contain valid data
- The system maintains page-to-frame mappings for future access operations

System Shutdown

```
Time slot  24
        CPU 3: Processed  8 has finished
        CPU 3 stopped
Time slot  25
        CPU 1: Put process  7 to run queue
        CPU 2: Processed  1 has finished
        CPU 2 stopped
        CPU 1: Dispatched process  7
Time slot  26
        CPU 1: Processed  7 has finished
        CPU 1 stopped
```

3.8 Questions and Answers

3.8.1 Question 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?

CPU bus	PAGE size	PAGE bit	No pg entry	PAGE Entry sz	PAGE TBL	OFFSET bit	PGT mem	MEMPHY	fram bit
20	256B	12	~4000	4byte	16KB	8	2MB	1MB	12
22	**256B**	**14**	**~16000**	**4byte**	**64KB**	**8**	**8MB**	**1MB**	**12**
22	512B	13	~8000	4byte	32KB	9	4MB	1MB	11
22	512B	13	~8000	4byte	32KB	9	4MB	128KB	8
16	512B	8	256	4byte	1KB	9	128KB	128KB	4

Table 2: Various CPU address bus configurations

3.8.2 Answer:

Advantages of the Proposed Multi-Segment Memory Design:

- **Optimized Memory Utilization:** Segmentation allows for targeted memory allocation strategies tailored to the unique behavior of each segment. For instance, the stack is used for managing function calls and local variables, the heap supports dynamic memory allocation, and the code segment holds executable instructions. This specialization helps minimize internal and external fragmentation, enhancing overall memory efficiency.
- **Structured Memory Layout:** Segmenting memory into distinct regions—such as code, data, stack, and heap—provides a logical and organized framework for memory management. Each segment serves a specific function, facilitating easier access control, debugging, and system-level memory operations.
- **Improved Context Switching:** During a context switch, segmented memory enables selective saving and restoring of only relevant memory regions. Instead of swapping the entire address space, the system can preserve and load only the active segments required by the process, thereby reducing overhead and speeding up context-switch operations.

3.8.3 Question 2:

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

3.8.4 Answer:

In the paging memory management system, dividing the address to more than two levels will have many impacts:

- **Dynamic and Scalable Page Table Allocation:** Multi-level paging enables demand-driven allocation of page tables. Page tables are instantiated only when needed by a process, which minimizes memory waste and enhances scalability for processes with sparse address space utilization.
- **Lower Page Table Memory Overhead:** Adding additional levels to the paging hierarchy results in smaller individual page tables at each level. This hierarchical breakdown reduces the total memory footprint required to maintain page tables, which is particularly advantageous in systems with large virtual address spaces.
- **Improved Memory Utilization and Fragmentation Handling:** Unlike single-level paging, which may require large contiguous memory blocks for page tables, multi-level paging allows smaller tables to be allocated non-contiguously. This leads to more flexible memory management and better accommodation of fragmented physical memory.
- **Faster Table Traversal and Translation Efficiency:** Although more levels introduce additional steps during address translation, each table involved is smaller and more cache-friendly. This can lead to quicker lookups within each level and can offset the performance degradation associated with managing a single, large, monolithic page table.

3.8.5 Question 3:

What are the advantages and disadvantages of segmentation with paging?

3.8.6 Answer:

Segmentation with paging is a memory management technique that combines the strengths of both segmentation and paging techniques by dividing the memory space into segments, and dividing the segments into pages, making memory allocation more flexible because each segment can have a different size. Though, this technique has many pros and cons.

- Pros:
  - Segments have arbitrary sizes, making allocation more flexible and at the same time reducing external fragmentation.
  - Optimizing memory usage and system performance because it does not need to load all pages. Segmentation with paging only loads the pages that need to be executed, thereby reducing swap time and handling multiple processes at the same time.

- Cons:
  - Implementation is very complicated.
  - Managing multiple segments and pages consumes CPU resources because there can be many pages stored in RAM but only one page is active at a certain time.
  - Pages have a fixed size so there is still internal fragmentation.

4 System Call

System calls are the interface between user applications and the kernel. They allow processes to request services from the operating system kernel.

4.1 System Call Architecture

4.1.1 System Call Table

The system call table is defined in `syscall.c` and populated from `syscalltbl.lst`, which is generated by `syscalltbl.sh`. This table maps system call numbers to their corresponding handler functions.

System Call Table Definition

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

/*
 * The sys_call_table[] is used for system calls
 */
#define __SYSCALL(nr, sym) #nr "-" #sym,
const char* sys_call_table[] = {
#include "syscalltbl.lst"
};
#undef __SYSCALL

const int syscall_table_size = sizeof(sys_call_table) / sizeof(char*);
```

4.1.2 System Call Dispatcher

The system call dispatcher is the entry point for all system calls. It routes each call to the appropriate handler based on the system call number.

System Call Dispatcher

```
#define __SYSCALL(nr, sym) case nr: return __##sym(caller, regs);

int syscall(struct pcb_t *caller, uint32_t nr, struct sc_regs* regs)
{
    regs->orig_ax = nr;

    switch (nr) {
    #include "syscalltbl.lst"
    default: return __sys_ni_syscall(caller, regs);
    }

    return 0;
};
```

4.1.3 Register Structure

The `struct sc_regs` defines the register set used for system call arguments:

System Call Register Structure

```
struct sc_regs {
    uint32_t a1;
    uint32_t a2;
    uint32_t a3;
    uint32_t a4;
    uint32_t a5;
    uint32_t a6;

    /*
     * orig_ax is used for:
     * - the syscall number
     * - error_code stored by the CPU on exceptions
     * - the interrupt number for device interrupts
     */
    uint32_t orig_ax;

    int32_t flags;
};
```

4.1.4 User-Level Interface

The `libsyscall` function provides a convenient wrapper for application code to make system calls:

User-Level System Call Interface

```
int libsyscall(struct pcb_t *caller,
               uint32_t syscall_idx,
               uint32_t a1,
               uint32_t a2,
               uint32_t a3)
{
    struct sc_regs regs;

    regs.a1 = a1;
    regs.a2 = a2;
    regs.a3 = a3;

    return syscall(caller, syscall_idx, &regs);
}
```

4.2 Implemented System Calls

4.2.1 Memory Mapping (syscall 17)

The memory mapping system call handles various memory operations, including page swapping and memory I/O.

Memory Mapping System Call

```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;
}
```

4.2.2 Operation Types

The memory mapping system call supports several operation types:

- `SYSMEM_MAP_OP (1)`: Reserved for future use
- `SYSMEM_INC_OP (2)`: Increases virtual memory area limit
- `SYSMEM_SWP_OP (3)`: Handles page swapping for virtual memory
- `SYSMEM_IO_READ (4)`: Reads a value from physical memory
- `SYSMEM_IO_WRITE (5)`: Writes a value to physical memory

4.2.3 List System Calls (syscall 100)

This utility system call lists all available system calls:

List System Calls

```
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;
}
```

4.2.4 Kill All Processes (syscall 101)

Kill All Processes System Call

```
int __sys_killall(struct pcb_t *caller, struct sc_regs *regs)
{
    ...
    int terminated_count = 0;
    struct pcb_t *proc = NULL;

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

        if (running->size > 0)
        {
            for (i = running->size - 1; i >= 0; i--)
            {
                proc = running->proc[i];
                if (proc != NULL && strcmp(proc->path, proc_name) == 0)
                {
                    int j;
                    for (j = i; j < running->size - 1; j++)
                    {
                        running->proc[j] = running->proc[j + 1];
                    }
                    running->size--;
                    // free(proc);

                    terminated_count++;
                }
            }
        }
    }
```

Kill All Processes System Call

```
#ifdef MLQ_SCHED
    // MLQ scheduler has multiple priority queues
    if (caller->mlq_ready_queue != NULL)
    {
        for (int prio = 0; prio < MAX_PRIO; prio++)
        {
            struct queue_t *ready_q = &(caller->mlq_ready_queue[prio]);
            if (ready_q != NULL && ready_q->size > 0)
            {
                // Iterate backwards for safe removal
                for (i = ready_q->size - 1; i >= 0; i--)
                {
                    proc = ready_q->proc[i];
                    if (proc != NULL && strcmp(proc->path, proc_name) == 0)
                    {
                        // Remove the process from the queue
                        int j;
                        for (j = i; j < ready_q->size - 1; j++)
                        {
                            ready_q->proc[j] = ready_q->proc[j + 1];
                        }
                        ready_q->size--;
                        // free(proc);
                        terminated_count++;
                    }
                }
            }
        }
    }
```

Kill All Processes System Call

```
#else
    // Single ready queue
    if (caller->ready_queue != NULL)
    {
        struct queue_t *ready_q = caller->ready_queue;
        if (ready_q->size > 0)
        {
            // Iterate backwards for safe removal
            for (i = ready_q->size - 1; i >= 0; i--)
            {
                proc = ready_q->proc[i];
                if (proc != NULL && strcmp(proc->path, proc_name) == 0)
                {
                    // Remove the process from the queue
                    int j;
                    for (j = i; j < ready_q->size - 1; j++)
                    {
                        ready_q->proc[j] = ready_q->proc[j + 1];
                    }
                    ready_q->size--;
                    free(proc);
                    terminated_count++;
                }
            }
        }
    }
#endif
    printf("Total of %d processes named '%s' terminated\n", terminated_count, proc_name);
    return terminated_count;
}
```

Kill All Process Flow

The flow of the `killall` system call is:

1. Read the target process name from memory using `libread`
2. Searches the running list for processes with matching names
3. Checks ready queues (either single queue or MLQ depending on configuration) for matches
4. Removes all matching processes from their respective queues
5. Returns the count of terminated processes

4.2.5 Not-Implemented System Call Handler

For invalid system call numbers, the system provides a helpful error handler:

Not-Implemented System Call Handler

```c
int __sys_ni_syscall(struct pcb_t *caller, struct sc_regs *regs)
{
    printf("ERROR: Called non-implemented system call (nr=%d)\n", regs->orig_ax);
    printf("Available system calls:\n");

    for (int i = 0; i < syscall_table_size; i++)
        printf("%s\n", sys_call_table[i]);

    return 0;
}
```

4.3 Real Test & Result Interpretation

4.3.1 Input Configuration

The system is configured using an input file (*os_ syscall*) with the following parameters:

```
2 1 1
2048 16777216 0 0 0
9 sc2  15
10 P0 15
```

This configuration specifies:

- Time slot: 2 time units
- Number of CPUs: 1
- Number of processes: 1
- Physical RAM size: 2048 bytes
- Process sc2 will start at time 9 with priority 15
- Process P0 will start at time 10 with priority 15

4.3.2 System Execution

```
./os os_syscall
...
Time slot  18
        CPU 0: Dispatched process  1
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 80
BYTE 00000001: 48
BYTE 00000002: -1
===== PHYSICAL MEMORY END-DUMP =====
The procname retrieved from memregionid 1 is "input/proc/P0"
Total of 4 processes named 'input/proc/P0' terminated
Time slot  19
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Explanation

1. **Executing Killall System Call (Time Slot 18):**
   - The killall system call reads the process name "P0"
   - The system searches for processes named "input/proc/P0"
   - 4 matching processes are found and terminated, returning 4 as the count
   - These processes were likely found across both running lists and ready queues
2. **Process Completion (Time Slot 19):**
   - After all "P0" processes are terminated, Process 1 (which was executing the system call) completes its execution
   - CPU 0, being the only CPU, stops as there are no more processes to execute

4.4 Questions and Answers

Question 1:

What is the mechanism to pass a complex argument to a system call using the limited registers? Illustrate by example the problem of your simple OS if you have any.

Answer:

Since the number of available registers is limited (typically a few general-purpose registers like **a0**, **a1**, **a2**, etc.), complex arguments such as structures, large arrays, or strings cannot be passed

directly through registers. Instead, the standard approach is:

1. Pass a pointer to the complex object in one of the registers (**a1**)
2. The pointer must reference a memory location in user space that is:
   - Allocated and valid
   - Mapped correctly so that the kernel has permission to read/write to it
3. The kernel will dereference the pointer during the system call to access the actual data (e.g. structure fields or string contents).

Question 2:

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

Answer:

If a system call job takes too long in its execution, it can negatively impact system performance and stability, particularly in a simplistic OS like the Simple Operating System that likely lacks advanced kernel preemption or task scheduling mechanisms.
Problems:

1. **CPU Starvation:** other processes may be blocked from running while the kernel is busy handling one long-running syscall, especially in a single-core or cooperative multitasking model
2. **Kernel Blocking:** since system calls typically run in kernel mode, a long execution can prevent the OS from handling other important kernel events like interrupts, timers, or I/O
3. **User Experience Lag:** from the application's point of view, the system may appear frozen or unresponsive during the syscall's execution
4. **Priority Inversion & Scheduling Issues:** long syscalls from low-priority processes may block higher-priority tasks if shared resources or locks are involved.

Best feasible practice: Long-running tasks should be split into smaller jobs and executed incrementally, or offloaded to a background worker thread/process to avoid holding the kernel for too long.

5 Put It All Together

Questions and Answers

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 a simple operating system does not properly manage synchronization, it can lead to serious issues such as race conditions, data inconsistency, and deadlocks.

Race Condition

Example: In our scheduler implementation, we use mutex locks to prevent race conditions:

Put Process Function

```
1 void put_proc(struct pcb_t * proc)
2 {
3     pthread_mutex_lock(&queue_lock);
4     pthread_mutex_unlock(&queue_lock);
5 }
```

Without synchronization, if two CPUs try to add processes simultaneously:

- Both CPUs could access the queue at the same time.
- Queue pointers might become corrupted.
- Some processes could be lost or duplicated.

Observed behavior when running with multiple CPUs:

```
Time slot 4
        CPU 0: Dispatched process  1
        CPU 1: Dispatched process  1 <- Same process on two CPUs!
```

Risk of Deadlock

Our `add_proc` function carefully manages lock order to prevent deadlocks:

Add Process Function

```c
void add_proc(struct pcb_t * proc)
{
    if(proc == NULL) return;
    proc->ready_queue = &ready_queue;
    proc->running_list = &running_list;
    /* First critical section */
    pthread_mutex_lock(&queue_lock);
    enqueue(&running_list, proc);
    pthread_mutex_unlock(&queue_lock);

    /* Second critical section */
    pthread_mutex_lock(&queue_lock);
    enqueue(&ready_queue, proc);
    pthread_mutex_unlock(&queue_lock);
}
```

Deadlock scenario that would occur with improper locking:

- CPU 0 acquires `queue_lock` for scheduling
- CPU 1 acquires `mem_lock` for memory allocation
- CPU 0 tries to get `mem_lock` (blocked, waiting)
- CPU 1 tries to get `queue_lock` (blocked, waiting)
- Both CPUs deadlock, system hangs indefinitely

Memory Corruption Example

Our memory management requires synchronization to prevent corruption:

Get Free Frame Function

```c
int get_free_frame(int* idx_list, int npages, struct pcb_t* proc)
{
    pthread_mutex_lock(&mem_lock); // Critical section protection
    int i;
    int count = 0;
    for (i = 0; i < MEMPHY_NFRAMES; i++) {
        if (_mem_stat[i].proc == 0) {
            idx_list[count] = i;
            count++;
            if (count == npages) {
                pthread_mutex_unlock(&mem_lock);
                return 0;
            }
        }
    }
    pthread_mutex_unlock(&mem_lock);
    return -1;
}
```

Without synchronization, when multiple processes request memory:

- Multiple CPUs might allocate the same physical frame to different processes.
- Memory corruption occurs when both processes write to the same physical memory.
- System crashes due to corrupted memory management structures.

Memory dump showing potential corruption:

```
===== PHYSICAL MEMORY DUMP =====
BYTE 00000114: 102 <- Could be written by two different processes
===== PHYSICAL MEMORY END-DUMP =====
```