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

PROJECT REPORT

Subject: Operating Systems (CO2018)

Simple Operating System

Lecturer : Nguyễn Phương Duy
Class : CC02
Group : 05
Students : Nguyễn Hoàng Quốc – 2353027 quoc.nguyenhoang2305@hcmut.edu.vn
Nguyễn Vũ Long – 2352696 long.nguyenty@hcmut.edu.vn
Nguyễn Duy Thái – 2353091 thai.nguyenkhmtbk@hcmut.edu.vn
Nguyễn Đình Ngọc Cẩn – 2352130 can.nguyencan8305@hcmut.edu.vn
Lê Phước Vũ – 2353341 vu.lephuocvu13@hcmut.edu.vn

HO CHI MINH CITY, 04/2025

Contents

Chapter 1

Introduction

Operating systems are the silent architects behind modern computing, seamlessly connecting hardware and software to enable everyday tasks. They power devices ranging from laptops and smartphones to servers, transforming raw hardware into tools that drive productivity, creativity, and communication. Operating systems have revolutionized daily life, making instant communication, effortless data management, and immersive entertainment not only possible but intuitive.

Their story is one of constant evolution. From simple tools for primitive machines, operating systems have developed into powerful platforms that support multitasking, ensure data security, and synchronize across devices. They have become essential to global business, personal creativity, and innovation across industries. Companies like Microsoft, Apple, and Google continually push these systems forward, refining performance, enhancing security, and improving usability.

As part of this course, my project involves building a simple operating system using the C programming language. This hands-on experience will provide a deeper understanding of key principles such as scheduling, synchronization, and memory management. Through this project, I aim to uncover the complexities hidden beneath modern technology and appreciate the vital role operating systems play in shaping our digital world.

Chapter 2

Scheduler

2.1 Theoretical Basis - CPU Scheduling

CPU scheduling is a core function of an operating system (OS) that determines which process in the ready queue is allocated to the CPU for execution. The scheduler ensures efficient resource utilization, balancing performance metrics such as throughput, waiting time, and responsiveness. This section explores the theoretical foundations of CPU scheduling, covering its objectives, core concepts, types of algorithms, and the multilevel queue (MLQ) scheduling approach.

2.1.1 Objectives of CPU Scheduling

The primary goal of CPU scheduling is to optimize system performance based on several key criteria:

1. **CPU Utilization**: Maximize the time the CPU spends executing processes rather than remaining idle.
2. **Burst time**: Burst time is an important factor in CPU scheduling, which is the amount of time a process needs to use the CPU to run continuously. This can vary depending on the type of process and its requirements. CPU scheduling algorithms need to consider the burst time of each process to allocate CPU time effectively
3. **Throughput**: Increase the number of processes completed per unit of time.
4. **Turnaround Time**: Minimize the total time from process submission to completion, including waiting and execution periods.
5. **Waiting Time**: Reduce the time a process spends in the ready queue before execution.
6. **Response Time**: For interactive systems, minimize the delay between a request and the first response.
7. **Fairness**: Ensure equitable CPU allocation across processes based on their priorities or requirements.

These objectives often conflict, necessitating trade-offs. For example, maximizing throughput may increase waiting times for some processes, while prioritizing low response times could reduce overall CPU utilization. The scheduler's design must align with the system's purpose, such as real-time, interactive, or batch processing.

2.1.2 Scheduling Concepts

CPU scheduling operates within the process management framework of the OS. Processes transition between states—new, ready, running, waiting, and terminated—managed by the scheduler. The **ready queue** holds processes awaiting CPU allocation, while the **process control block (PCB)** stores critical information, such as the program counter, CPU registers, and priority, to enable context switching.

Scheduling decisions occur at specific events:

- When a process completes or terminates.
- When a running process is interrupted (e.g., time slice expiration in preemptive scheduling).
- When a process moves from running to waiting (e.g., for I/O operations).
- When a new process enters the ready queue.

Scheduling can be **preemptive**, where the OS can interrupt a running process to allocate the CPU elsewhere, or **non-preemptive**, where a process runs until it voluntarily yields the CPU. Preemptive scheduling enhances responsiveness and fairness, particularly in time-sharing systems.

2.1.3 Types of Scheduling Algorithms

Various scheduling algorithms exist, each tailored to specific system requirements:

1. **First-Come, First-Served (FCFS)**: Executes processes in arrival order. Simple but susceptible to the convoy effect, where a long process delays others.
2. **Shortest Job Next (SJN)**: Selects the process with the shortest expected execution time, minimizing average waiting time. Requires accurate burst time estimates.
3. **Priority Scheduling**: Allocates the CPU to the highest-priority process. Priorities may be static or dynamic, but low-priority processes risk starvation.
4. **Round-Robin (RR)**: Assigns a fixed time slice (quantum) to each process in a cyclic order, ideal for time-sharing systems to ensure responsiveness.
5. **Multilevel Queue (MLQ)**: Organizes the ready queue into multiple priority-based sub-queues, each with its own scheduling policy.
6. **Multilevel Feedback Queue (MLFQ)**: Extends MLQ by allowing processes to move between queues based on execution behavior, balancing responsiveness and efficiency.

The choice of algorithm depends on the system's goals, such as low latency for real-time systems or high throughput for batch processing.

2.1.4 Multilevel Queue (MLQ) Scheduling

The multilevel queue (MLQ) scheduling algorithm divides the ready queue into multiple sub-queues, each associated with a fixed priority level. Processes are assigned to a queue based on their priority, and the scheduler allocates the CPU to the highest-priority non-empty queue. Each queue may employ a distinct scheduling policy, such as round-robin or FCFS.

Key Features:

- **Priority-Based Queues**: Priorities range from 0 (highest) to a maximum value (e.g., 139). Each queue contains processes of identical priority.
- **Fixed Slots**: Each queue receives a CPU time slot, computed as slot = (MAX_PRIO − prio), where MAX_PRIO is the highest priority (e.g., 140). Higher-priority queues receive larger slots.
- **Preemption**: If a higher-priority process becomes ready, the current process is preempted, ensuring critical tasks are prioritized.
- **Queue Traversal**: The scheduler scans queues in descending priority order, selecting the first non-empty queue.

Operation:

1. The loader creates a process, assigns a PCB, and places it in the appropriate ready queue based on priority (enqueue operation).
2. The scheduler retrieves the next process from the highest-priority non-empty queue (get process operation).

3. The CPU executes the process for its allocated time slot or until it yields (e.g., for I/O).
4. If preempted or completed, the process is returned to the ready queue or terminated (put process operation).

Advantages:

- Prioritizes high-priority processes, such as system tasks, for immediate execution.
- Flexible, as each queue can use a tailored scheduling policy.
- Effective for systems with diverse process types (e.g., interactive and batch).

Challenges:

- **Starvation**: Low-priority processes may be indefinitely delayed if high-priority queues remain populated.
- **Complexity**: Managing multiple queues increases design and maintenance overhead.
- **Priority Inversion**: A high-priority process may be delayed by a lower-priority one holding a shared resource, requiring mitigation strategies.

Question 1

What is the advantage of using priority queue in comparison with other scheduling algorithms you have learned?

Answer:
As described : "each queue have only fixed slot to use the CPU...left the remaining work for future slot even though it needs a completed round of ready queue." -> **Round Robin** The scheduling policy described — a **Multilevel Queue (MLQ)** with **Round-Robin** inside each priority level
The advantages of this policy:

- **Better Support for Multiple Priorities**: Each process is placed in a queue according to its fixed priority (prio). This ensures high-priority processes are serviced more frequently and don't get starved by low-priority ones (unlike plain FCFS or Round Robin). Compared to Priority Scheduling, MLQ improves fairness because it still uses round-robin within the same level.
- **Fairness Within Priority Levels** : Within each priority level, processes share the CPU in round-robin, avoiding starvation within that level — a problem often seen in pure Priority Scheduling.
- **Deterministic CPU Allocation (Fixed Slot Scheduling)**: The "slot = MAX_PRIO − prio" model ensures each queue gets a deterministic number of CPU turns per cycle. Unlike Shortest Job First (SJF) or Shortest Remaining Time First (SRTF), this approach doesn't need knowledge of process runtime, which is often hard to estimate.
- **Scalable to Multi-Processor Systems** : Because queues are independent, they can be distributed across multiple processors, making this approach very scalable, unlike older algorithms (FCFS or SJF) which assume a single CPU.
- **Real-Time and Background Process Separation**: You can dedicate high priority levels for time-sensitive tasks (e.g., user interactions or real-time tasks), and lower ones for background jobs — something hard to achieve with Round Robin or FCFS.
- **Multilevel Feedback Queue (MLFQ)**: Both MLQ and MLFQ use multiple queues but differ in approach. MLQ assigns processes to static, priority-based queues with tailored algorithms (e.g., Round Robin for interactive tasks, FCFS for batch), ensuring predictable scheduling as processes stay put and the highest-priority queue is prioritized. MLFQ dynamically adjusts priorities based on behavior, moving processes between queues to favor short tasks while promoting fairness. MLQ

offers predictability for critical tasks, lower overhead by avoiding MLFQ's constant adjustments, and flexibility with queue-specific algorithms. However, MLQ lacks adaptability to changing process behavior, risks starving low-priority tasks, and requires complex configuration, while MLFQ excels in dynamic responsiveness and ensures fairness.

Feature	MLQ (Described)	FCFS	SJF / SRTF	Round Robin	Priority (Basic)	Multi-Processor Support
Starvation Prevention	Yes (within priority level)	No	No (can starve long jobs)	Yes	No (low prio may starve)	Yes
Fairness	Yes (within levels)	No	No	Yes	No	Yes
Real-Time Support	Yes (via priorities)	No	No	No	Yes	Yes
Preemption Support	Yes	No	Yes (SRTF)	Yes	Yes	Yes
Time Estimation Needed	No	No	Yes	No	No	No
Multi-Processor Ready	Yes	No	No	Limited	Yes	Yes

Table 2.1: Feature Comparison between Scheduling Algorithms

2.2 Algorithm and Code

2.2.1 Initialization and Queue Declaration

To implement the Multi-level Queue (MLQ) scheduling algorithm, we initialize multiple queues based on priority using the following loop:

```
for (i = 0; i < MAX_PRIO; i++) {
    mlq_ready_queue[i].size = 0;
    slot[i] = MAX_PRIO - i;
}
```

Explanation: Each queue corresponds to a specific priority level. The variable `slot[i]` determines how many times a process in the i-th priority queue can be scheduled before reallocation. Higher-priority queues (lower i) are assigned more slots to improve responsiveness.

For example: In our configuration, we have `MAX_PRIO` $= 140$. The highest priority is at $i = 0$, and the lowest priority is at $i = 140$, where `slot[140]` $= 0$.

Mathematically:

$$\text{Higher Priority (lower } i) \Rightarrow \text{More Slots}$$

$$\text{Lower Priority (higher } i) \Rightarrow \text{Fewer Slots}$$

2.2.2 Enqueue and Dequeue Functions

```
void enqueue(struct queue_t * q, struct pcb_t * proc)
{
        if(q == NULL || proc == NULL || q->size == MAX_QUEUE_SIZE)
        {
            return;
        }
        for(int i=0;i<q->size;i++){
                if (q->proc[i]==proc){
                        return;
                }
        }
        q->proc[q->size] = proc;
        ++(q->size);
}
```

The enqueue operation can be expressed mathematically as:

$$\text{If } q.\text{size} < \text{MAX\_QUEUE\_SIZE, then}$$

$$q.\text{proc}[q.\text{size}] = p \quad \text{and} \quad q.\text{size} \leftarrow q.\text{size} + 1$$

Explanation: The `enqueue()` function adds a process to the back of the priority queue in descending order if the queue is not full. Only need to add at the back of the queue because the processes have the same priority level in each queue.

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

Explanation: The `dequeue()` function retrieves and removes the first process in the queue, maintaining FIFO (First-In-First-Out) behavior.

2.2.3 Get a Process from MLQ

```
    struct pcb_t * get_mlq_proc(void) {
        static int cur_prio = 0;
        struct pcb_t *proc = NULL;
        pthread_mutex_lock(&queue_lock);

        for (int i = 0; i < MAX_PRIO; i++)
        {
         int prio = (cur_prio + i) % MAX_PRIO;
       if (!empty(&mlq_ready_queue[prio]))
         {
             slot[prio] -= i;
             if (slot[prio] > 0)
           {
                proc =dequeue(&mlq_ready_queue[prio]);
                    slot[prio]--;
           }
                else
                {
                cur_prio = (cur_prio + 1) % MAX_PRIO;
                slot[prio] = MAX_PRIO - prio;
                }
                break;
            }
        }
        pthread_mutex_unlock(&queue_lock);
        return proc;
}
```

Explanation:

The `get_mlq_proc()` function selects the next process to run based on the Multilevel Queue (MLQ) scheduling strategy. It rotates through priority levels to ensure fairness and prevent starvation of lower-priority processes.

Example:

Assume `MAX_PRIO` = 4 and processes:

- P_0 (Priority 0): System process
- P_1 (Priority 1): Interactive process

- P_2 (Priority 2): Background process
- P_3 (Priority 3): Batch process

First Call ($cur\_prio = 0$):

$$\begin{aligned} i = 0 &\Rightarrow prio = (0+0) \mod 4 = 0 \quad \text{(check P0)} \\ i = 1 &\Rightarrow prio = (0+1) \mod 4 = 1 \quad \text{(check P1)} \\ i = 2 &\Rightarrow prio = (0+2) \mod 4 = 2 \quad \text{(check P2)} \\ i = 3 &\Rightarrow prio = (0+3) \mod 4 = 3 \quad \text{(check P3)} \end{aligned}$$

Second Call ($cur\_prio = 1$):

$$\begin{aligned} i = 0 &\Rightarrow prio = (1+0) \mod 4 = 1 \quad \text{(check P1)} \\ i = 1 &\Rightarrow prio = (1+1) \mod 4 = 2 \quad \text{(check P2)} \\ i = 2 &\Rightarrow prio = (1+2) \mod 4 = 3 \quad \text{(check P3)} \\ i = 3 &\Rightarrow prio = (1+3) \mod 4 = 0 \quad \text{(check P0)} \end{aligned}$$

Third Call ($cur\_prio = 2$):

$$\begin{aligned} i = 0 &\Rightarrow prio = (2+0) \mod 4 = 2 \quad \text{(check P2)} \\ i = 1 &\Rightarrow prio = (2+1) \mod 4 = 3 \quad \text{(check P3)} \\ i = 2 &\Rightarrow prio = (2+2) \mod 4 = 0 \quad \text{(check P0)} \\ i = 3 &\Rightarrow prio = (2+3) \mod 4 = 1 \quad \text{(check P1)} \end{aligned}$$

Benefits of this approach:

- Fair rotation between different priority levels.
- Prevents starvation for low-priority processes.
- Maintains responsiveness for high-priority processes.
- Easy to extend or shrink the system by adding or removing priority levels.

2.2.4 Run input and explain output

Although the theoretical results are completely correct, in practice there may be some confusion — for example, a process that should be dispatched at timeslot 0 might actually be dispatched at timeslot 1. This happens because, in some cases, the CPU is busy and cannot dispatch the process immediately, causing a slight delay. However, this does not significantly impact the overall results. Additionally, we assume that timeslot 0 is reserved for loading processes, so it will not appear in the Gantt chart. into **Case 1:**

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

Explanation: First line means time slice = 2, 1 CPU, 4 processes.
Next four lines meant:
Load s0 at time slot 0 with priority 4

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

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

Load s1 at time slot 4 with priority 0

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

Load s2 at time slot 6 with priority 0

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

Load s3 at time slot 6 with priority 0

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

Output:

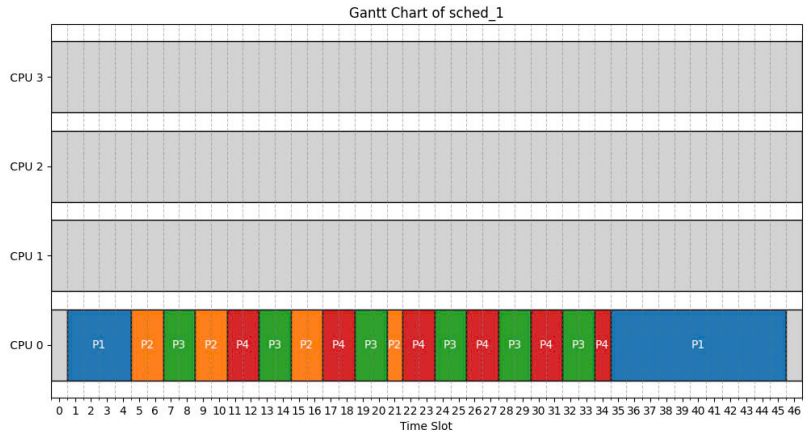
```
Time slot   0
ld_routine
        Loaded a process at input/proc/s0, PID: 1 PRIO: 4
Time slot   1
        CPU 0: Dispatched process  1
Time slot   2
Time slot   3
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
```

```
Time slot   4
        Loaded a process at input/proc/s1, PID: 2 PRIO: 0
Time slot   5
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
        Loaded a process at input/proc/s2, PID: 3 PRIO: 0
Time slot   6
Time slot   7
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  3
        Loaded a process at input/proc/s3, PID: 4 PRIO: 0
Time slot   8
Time slot   9
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  2
Time slot  10
Time slot  11
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  4
Time slot  12
Time slot  13
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  3
Time slot  14
Time slot  15
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  2
Time slot  16
Time slot  17
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  4
Time slot  18
Time slot  19
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  3
Time slot  20
Time slot  21
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  2
Time slot  22
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  4
Time slot  23
Time slot  24
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  3
Time slot  25
Time slot  26
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
Time slot  27
Time slot  28
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  3
Time slot  29
Time slot  30
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
Time slot  31
Time slot  32
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  3
```

```
Time slot   33
Time slot   34
        CPU 0: Processed  3 has finished
        CPU 0: Dispatched process  4
Time slot   35
        CPU 0: Processed  4 has finished
        CPU 0: Dispatched process  1
Time slot   36
Time slot   37
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   38
Time slot   39
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   40
Time slot   41
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   42
Time slot   43
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   44
Time slot   45
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   46
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

- **Time slot 0**: Process `s0` is loaded into the CPU with **PID: 1** and **Priority: 4**. The process is dispatched and starts executing.
- **Time slots 1-3**: Process `s0` runs for 2 time units (time slice = 2). After its time slice, it is moved to the ready queue.
- **Time slot 4**: Process `s1` is loaded and dispatched, as it has the next available priority.
- **Time slots 5-7**: Process `s1` runs for its time slice. Then, Process `s2` is loaded and dispatched.
- **Time slots 8-10**: Process `s2` runs, and Process `s3` is loaded and dispatched.
- **Time slot 22**: Process `s2` finishes execution, and Process `s3` is dispatched.
- The cycle continues with processes being moved to the ready queue and dispatched in the following time slots.
- Finally, in **time slot 46**, all processes have finished executing and the CPU stops.

Gantt chart:



[H]

Figure 2.1: *Gantt chart of this case*

Chapter 3

Memory management

3.1 Theoretical Background

3.1.1 Logical and Physical Address spaces

- **Logical Address (Virtual Address)** is the address generated by the CPU, used by programs to access memory. It is independent of the actual physical memory layout and is defined by the operating system and the program.
- **Physical Address** The actual location in RAM (main memory) where the data is stored.

3.1.2 Memory Management Unit (MMU)

The Memory Management Unit (MMU) is a hardware device inside the computer, usually part of the CPU, that serves the purpose of translating logical addresses into physical addresses during program execution.

The MMU provides benefits such as more efficient use of memory, enhanced overall system performance, and added memory protection. However, it also comes at the cost of increased complexity and can introduce some performance overhead.

[h]

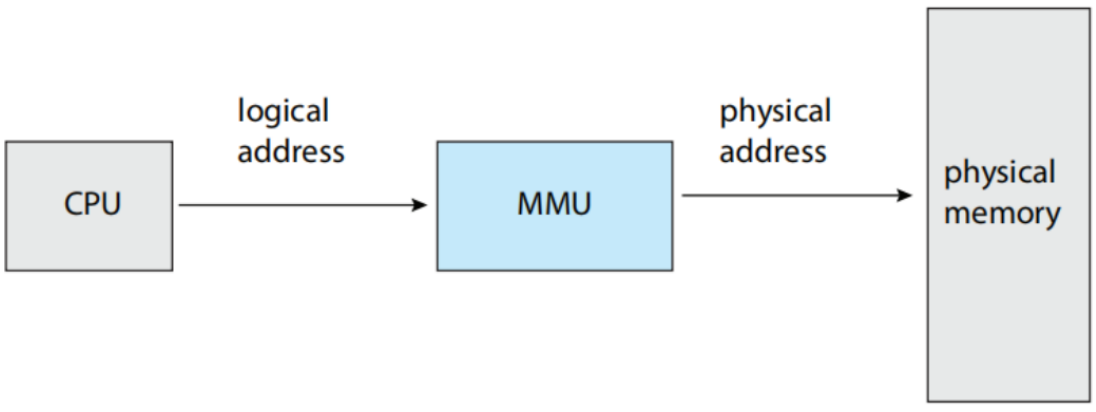
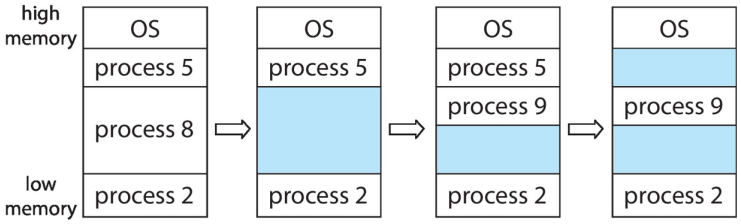


Figure 3.1: *Management Memory Unit*

3.1.3 Memory Allocation Variable Partition

Variable partitioning is a memory management technique where memory is divided into partitions of different sizes, with each partition assigned to a single process. The operating system maintains records of both free and allocated memory areas. Initially, all memory is available as one large free partition. When a process requests memory, the OS searches for a free space (hole) large enough to fit the process. If a suitable hole is found, memory is allocated; otherwise, the process may be delayed or denied. When a process finishes, its allocated memory is returned to the pool of free memory.

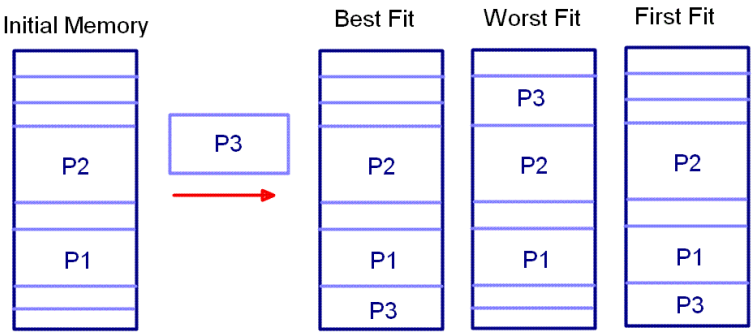


[h]

Figure 3.2: *Variable Partition*

Some allocation strategies:

- **First-fit:** Will allocated at the first free region that is large enough. Fast, but can lead to external fragmentation.
- **Best-fit:** Will allocated at the free region that had the smallest space possible. More memory usage efficient but slow.
- **Worst-fit** Will allocated at the free region that is the largest. Slowest and wasteful of memory.



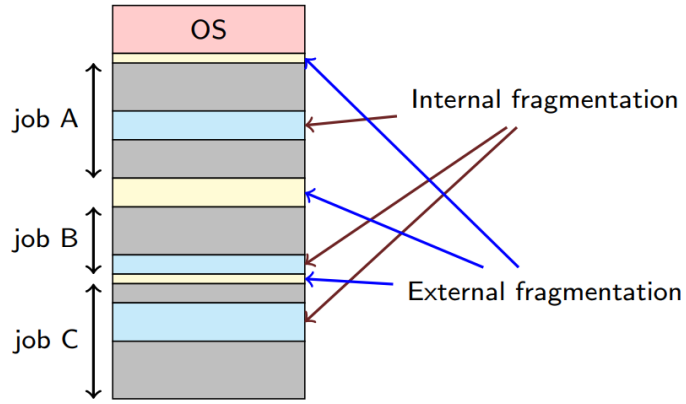
[h]

Figure 3.3: *Best-fit, Worst-fit, First-fit allocation*

3.1.4 Memory Fragmentation

Memory fragmentation occurs due to the inefficient utilization of memory over time.

- **External Fragmentation** occurs when free memory is separated into small non-contiguous blocks over time, making it difficult to allocate large contiguous memory spaces even if enough total free memory exists.
- **Internal Fragmentation** happens when memory is allocated in fixed-size blocks and the allocated block is larger than the requested memory, leaving unused space inside the block.



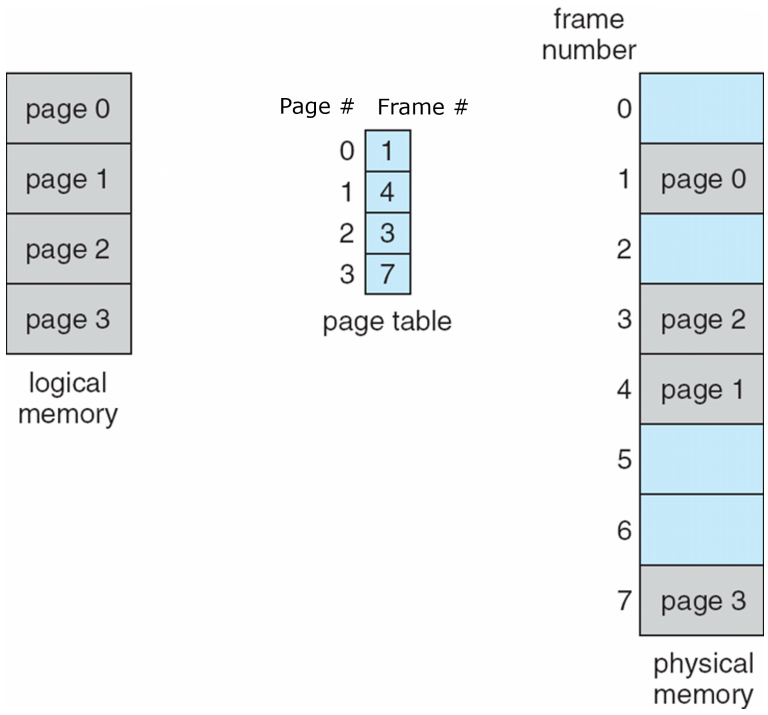
[h]

Figure 3.4: *Illustrate for Fragmentation*

3.1.5 Paging

Paging is a memory management scheme that divides both logical memory (process) and physical memory into fixed-sized blocks called pages and frames, respectively. The operating system uses a page table to map pages to available frames. This eliminates external fragmentation and allows noncontiguous memory allocation, but can lead to internal fragmentation if the last page doesn't fully fill its frame.

Page Table is the data structure used to translate the process's logical page numbers into physical frame numbers.

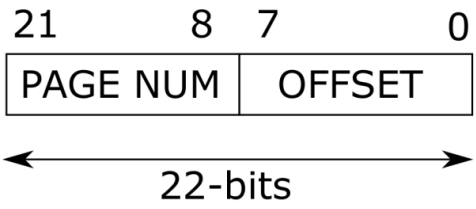


[h]

Figure 3.5: *Paging Model of logical and physical memory*

Logical Adresses (CPU Addresses) are split into.

- **page number - p** : Index into the page table.
- **page offset - d** : Relative position within the page.



[h]

Figure 3.6: *Logical addresses scheme for 22-bits cpu*

The Memory Management Unit (MMU) will translates logical addresses to physical addresses by.

1. Extracting the page number from the logical addresses.
2. Using the number extracted to look up the frame correspond to that page in the page table.
3. Replacing the page number with the frame number obtained.
4. Combining with the given offset in the page to get the correct physical address.

Page size affects both internal fragmentation and memory management overhead. Smaller pages help reduce internal fragmentation but add complexity to memory management. To optimize performance, many systems support multiple page sizes (e.g., 4KB and 2MB on Linux or Windows).

The operating system tracks the usage of frames via a **Frame Table** and maintains separate page tables for each process to manage address mappings during system calls and context switches.

[h]

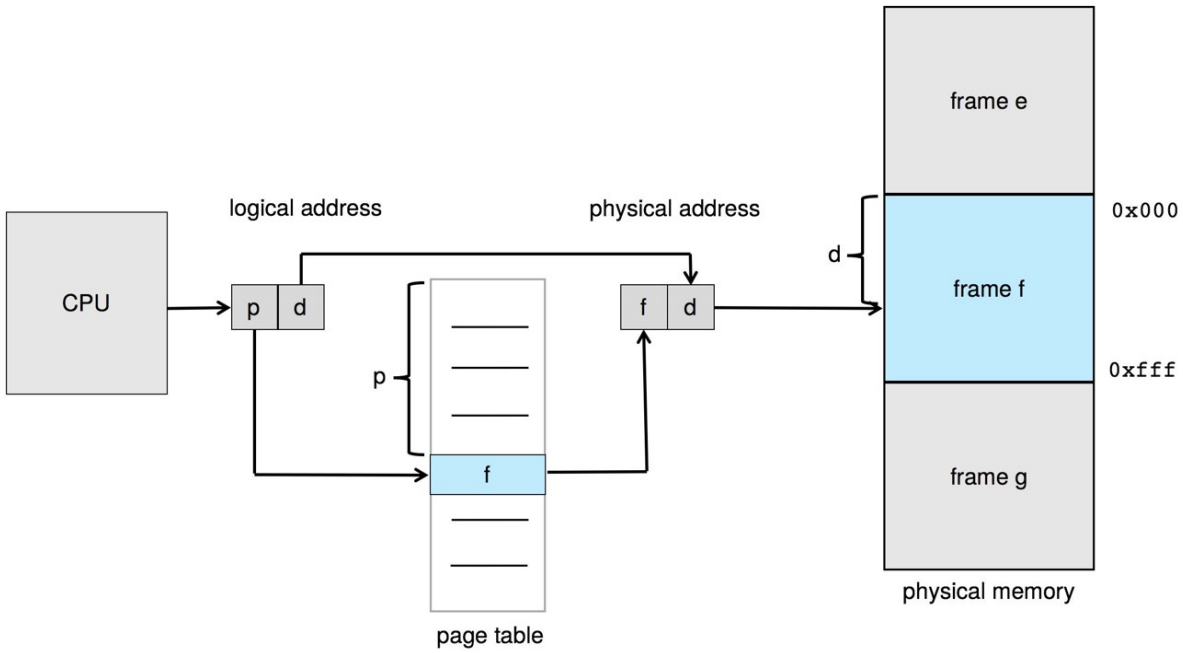
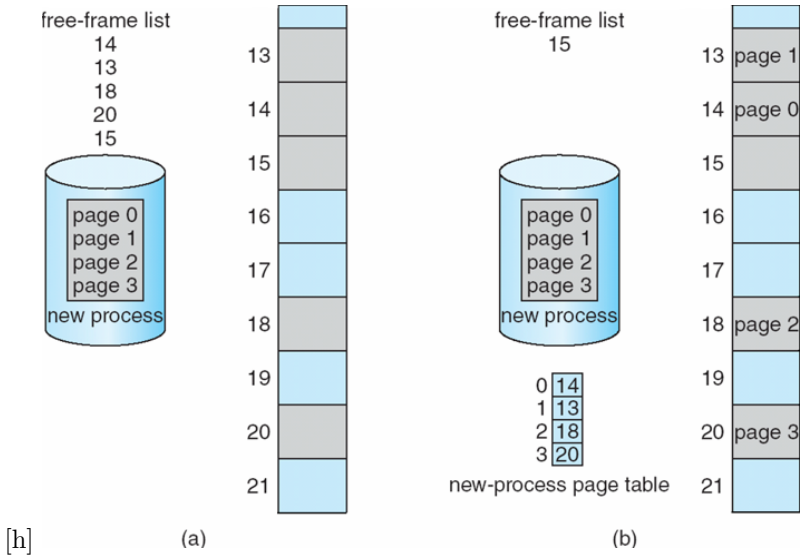


Figure 3.7: *Illustrate for the operation of MMU*



[h]

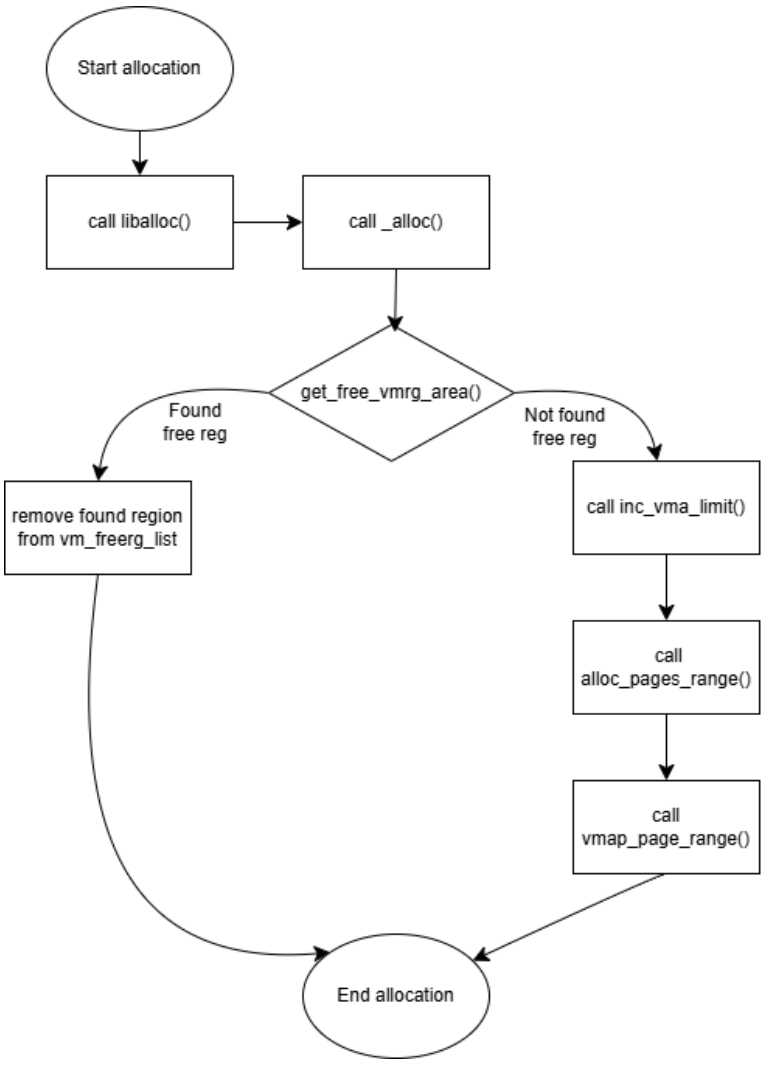
Figure 3.8: *Free frame before allocation and after allocation*

3.2 Implementation

For this Simple OS assignment, in memory management there are 4 main parts that need to implement is alloc, free, read and write.

3.2.1 Alloc.

Function: Allocate a region of size BYTE for the process and store the allocated address in the desinated register.



[h]

Figure 3.9: *Flow chart for allocation.*

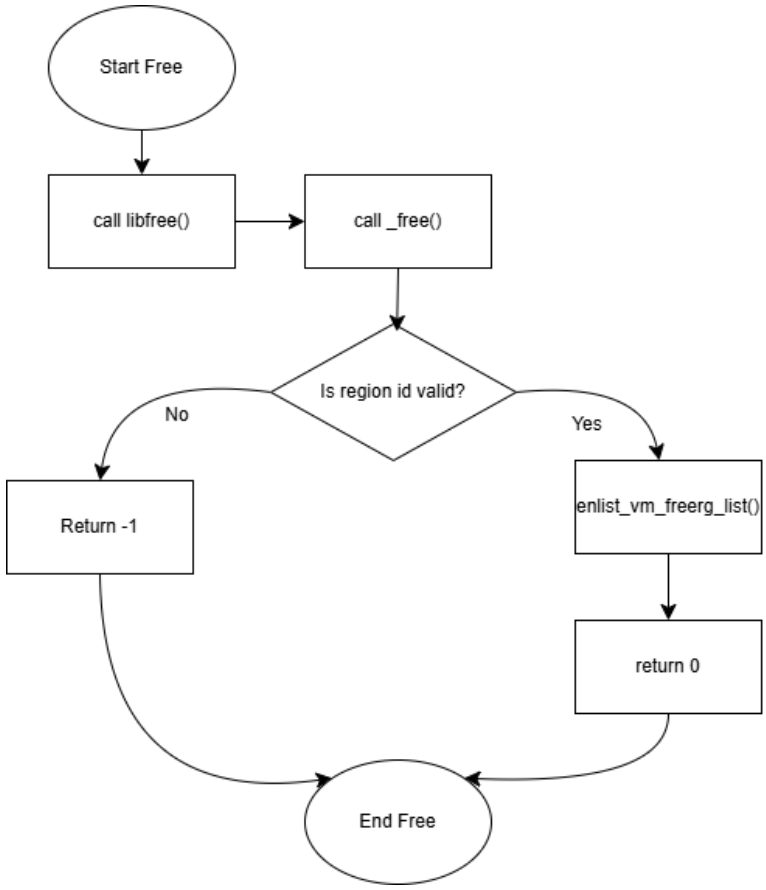
Explanation

1. call liballoc() to start the allocation with the parameter process, size to allocate, and region to allocated.
2. call alloc() with addition parameter return Address of the allocated.
3. call get_free_vmrg_area() to find the free region that is big enough to allocated.
4. If found free region, remove the free region out of the vm_free_vmrg_area then updated the new start of the free region if needed to enlist that free region again. Assign the new obtain free region to allocate and return the alloc_address.

5. If not found free region, call inc_vma_limit() to increase the space of the virtual memory area.
6. Call alloc_pages_range() to allocated the required number of page to frame in ram and put those newly allocated frame to freefp list.
7. Call vmap_page_range() to map the pages in the new space in virtual memory to physical frames in the process's page table.
8. Also return the return allocation address and saved to the desinated register.

3.3 Free.

Function: deallocate the memory of the desinated region.



[h]

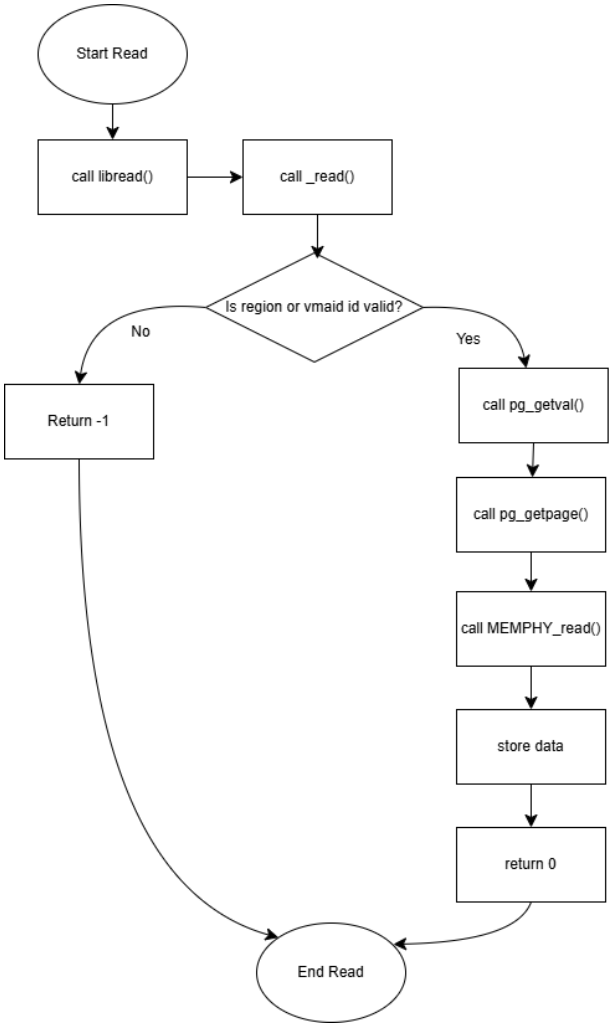
Figure 3.10: *Flow chart for free operation.*

Explanation:

1. call libfree() with the parameter is the region id of which the OS need to deallocated.
2. call free() to check if the region id is valid or not (does the region id exist?).
3. If the region is valid, add the region to the freerg_list for future allocation. Return 0.
4. If region id is not valid. Return -1.

3.4 Read.

Function: Read a byte from memory at the designated region and store in destination register.



[h]

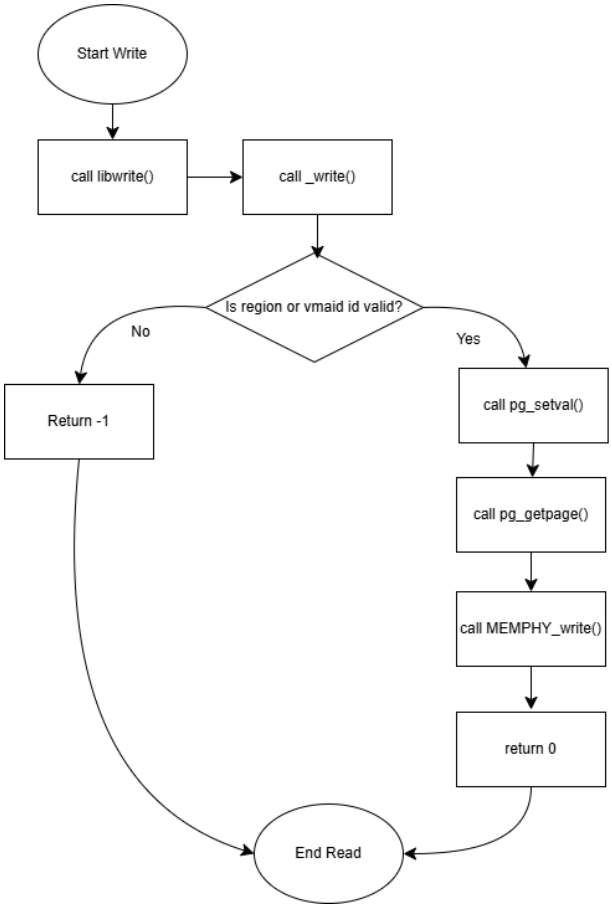
Figure 3.11: *Flow chart for Read operation.*

Explanation:

- call libread() with parameter region_id, offset, and register to store the data.
- call read() to check the validity of the region_id and the area_id.
- if cannot validify, return -1, read fail.
- if both region_id and area_id is valid, call pg_getval() to read the value at the logical address (source + offset).
- call pg_getpage() to bring the place online on RAM, if the page not already online, swap operation are needed to take place.
- call MEMPHY_read to read the value on the physical memory.
- store the read data to the register and return 0.

3.5 Write.

Function: Write 1 BYTE value at the offset of a region. **Explanation:**



[h]

Figure 3.12: *Flow chart for Write operation.*

- call libwrite() with parameter data to write, region_id and offset we want to write into.
- call _write() to check the validity of the region_id and the area_id.
- if invalid, return -1, mark the write operation had failed.
- if valid, call pg_setval() to read the value at the logical address (source + offset).
- call pg_getpage() to get the page online on RAM, if it had not already online, swap operation must occur.
- call MEMPHY_write to write the data on the physical memory.
- return 1, mark that write operation had completed.

3.6 Answer Question of the Assignment.

Question 2

What is the advantage of the proposed design of multiple segments?

Dividing memory into multiple segments creates a clearer and more organized memory structure. Each segment is dedicated to a specific function—such as code, data, stack, or heap—which improves protection, simplifies access, and makes debugging more modular. Additionally, this segmentation helps minimize external fragmentation by allowing each segment to be managed separately, reducing the need for large contiguous blocks of physical memory and enhancing overall memory efficiency.

Question 3

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

Multi-level paging, such as two-level or hierarchical paging, enhances the system's ability to efficiently manage very large address spaces by reducing the memory overhead associated with page tables. Only the segments of the page tables that are actively accessed are maintained in memory.

In a two-level paging system, the virtual address is divided into three components: an index for the outer page table, an index for the inner page table, and an offset. This design supports efficient handling of sparse memory usage without requiring large contiguous page tables.

[h]

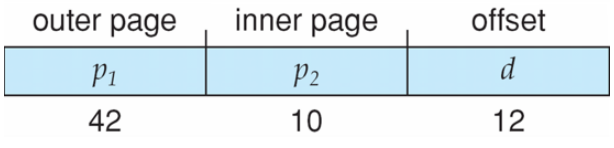


Figure 3.13: *Address of 2 level paging in 64-bit system*

[h]

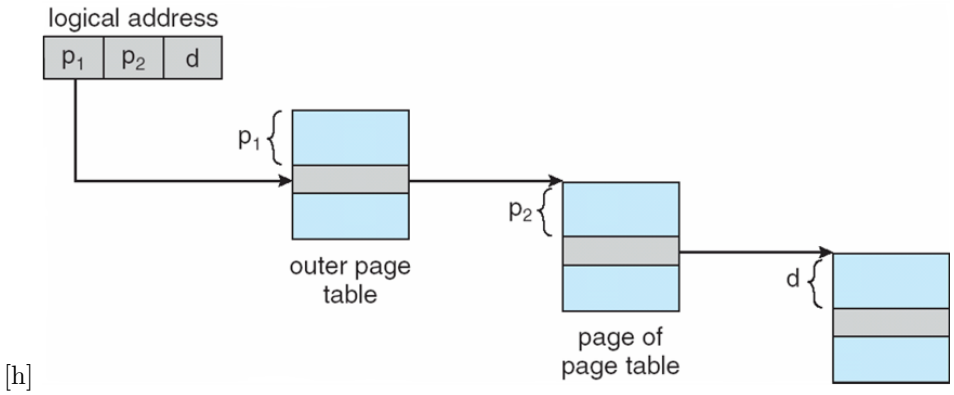


Figure 3.14: *Translation scheme of 2 level paging structure*

Nevertheless, the increased complexity of this scheme demands a CPU capable of addressing larger bit-widths and often leads to slower address translation, as multiple memory accesses are required during the process.

Question 4

What is the advantage and disadvantage of paging segmentation?

Paging and segmentation each off distinct advantages, but they also come with their own limitations. When we combine both techniques, we aim to enhance both of their strenghts while minimizing each other weaknesses.

Advantage of paging segmentation.

- Page table size is reduced because pages are created only for the data sections of segments, not for the entire process. Unlike traditional paging, where each part of a process requires its own page table, segmented paging stores pages only for data sections, reducing the number of pages and making memory management more efficient.
- Enhanced security by allowing protection at the segment level and isolating segments to prevent accidental overwrites. Paging ensure process can't access unauthorized memory through address translation. Together Paging segmentation provide better isolation and access control for more secure memory management.
- Minimize External fragmentation by using paging to allocate memory in fixed-size chunk. Each segment divided to smaller pages, which can be placed everywhere in the physical memory, allowing more efficient use of memory.
- Efficient Swapping by allows it to swapping out individual pages of a segment rather than the entire segment, reduce the data moving in and out of virtual memory, enhace performance and responsiveness.
- Flexibility in Allocate memory, allow memory to shrink and grow as needed.

Disadvantage of paging segmentation

- Maintaining both segment and page tables increases system complexity and memory usage.
- While paging eliminates external fragmentation, internal fragmentation can still occur within segments, leading to wasted memory.

3.7 Output explanation

3.7.1 Input os_0_mlq_paging

File Process p0s

```
1 14
calc
alloc 300 0
alloc 300 4
free 0
alloc 100 1
write 100 1 20
read 1 20 20
write 102 2 20
read 2 20 20
write 103 3 20
read 3 20 20
calc
free 4
calc
```

File Process p1s

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

Input File os_0_mlq_paging

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

In this input we can see that the configuration for this run is

- Time slice = 6.
- Cpu count = 2.
- Item of product = 4.

Step by step Explanation of output.

Because this part only cover basic operation like alloc, free, write and read so we only cover time slots that these operation occurs in.

- **Time Slot 2.**

```
Time slot   2
        CPU 0: Dispatched process  2
================ 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
================================================================
================ PHYSICAL MEMORY AFTER ALLOCATION ==============
PID=1 - Region=4 - Address=00000200 - 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
```

Action:

- proc pid 1 alloc 300 bytes for region 0
- proc pid 1 alloc another 300 bytes for region 4.

Details:

- First the programs allocate for process pid 1 the 300 bytes at the address 00000000. Because to allocated 300 bytes need 2 page aligned so it mapped page number 0 and 1 to frame 1 and 0.
- Then it allocate another 300 bytes to the same process at addresss 0000200, which is 512 bytes offset due to 2 page had already been allocated before it.

- **Time slot 3.**

```
Time slot   3
        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
```

Action:

- proc pid 1 deallocated all the memory in region 0.

Details: The programs deallocated all the memory in region 1 which is the space from 0 - 300 bytes and put it in free_rg_list.

- **Time slot 5.**

```
Time slot   5
================ 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
================================================================
        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
```

Action:

- proc pid 1 alloc 100 bytes for region 1.

- proc pid 1 write into region 1 at offset 20

Details:

- The program traverse the free_rg_list and found that there is free space between the address 0-512 bytes so it just allocated region 1 from 0-100 and from 100-512 into free_rg_list.
- The program write into region 1 at offset 20 the value 100. Which is written at the address 0000014.

- **Time slot 12.**

```
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 - 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
================================================================
read 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 =====
```

Action:

- proc pid 1 read the data in region 1 offset 20

Details:

- The program read the value in the virtual address at region 1 offset 20 and in the physical memory is 00000014, which is the address we have written 100 above.

- **Time slot 13.**

```
Time slot  13
================= PHYSICAL MEMORY AFTER WRITING ================
write region=2 offset=20 value=102
Time slot  14
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  3
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
===== PHYSICAL MEMORY END-DUMP =====
```

Action:

- proc pid 1 write the data in region 2 at offset 20.

Details:

- Actually at this step, the program hadn't allocated region 2 yet, but due to the limited of this simple OS, the unallocated region still had the start and end address at 0, so when the write operation, the program still recognized the region had the starting address 0 so it will calculated the with the virtual address 0 + offset = 20 in this case. And map it to the physical frame is 0000114 is where we write before therefore the data we write before is overwritten.

- **Time slot 15.**

```
Time slot  15
================= 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
================================================================
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
===== PHYSICAL MEMORY END-DUMP =====
================= 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: 103
```

```
Time slot  16
===== PHYSICAL MEMORY END-DUMP =====
```

Action:

- proc pid 1 read the data in region 2 at offset 20.
- proc pid 1 write the data in region 3 at offset 20.

Details:

- Read the memory in Physical memory at address 00000114, with the value 102.
- Write in the physical memory at address 00000114, with the value 103.

- **Time slot 16.**

```
Time slot  16
================= 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
================================================================
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  17
===== PHYSICAL MEMORY END-DUMP =====
```

Action:

- proc pid 1 read the data in region 3 at offset 20.

Details:

- Read the memory in Physical memory at address 00000114, with the value 103.

- **Time slot 19.**

```
============== PHYSICAL MEMORY AFTER DEALLOCATION ==============
Time slot  19
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
================================================================
```

Action:

- proc pid 1 deallocated all memory in region 4.

Details:

- Deallocated the memory region from 512->812 and put it into free_rg_list.

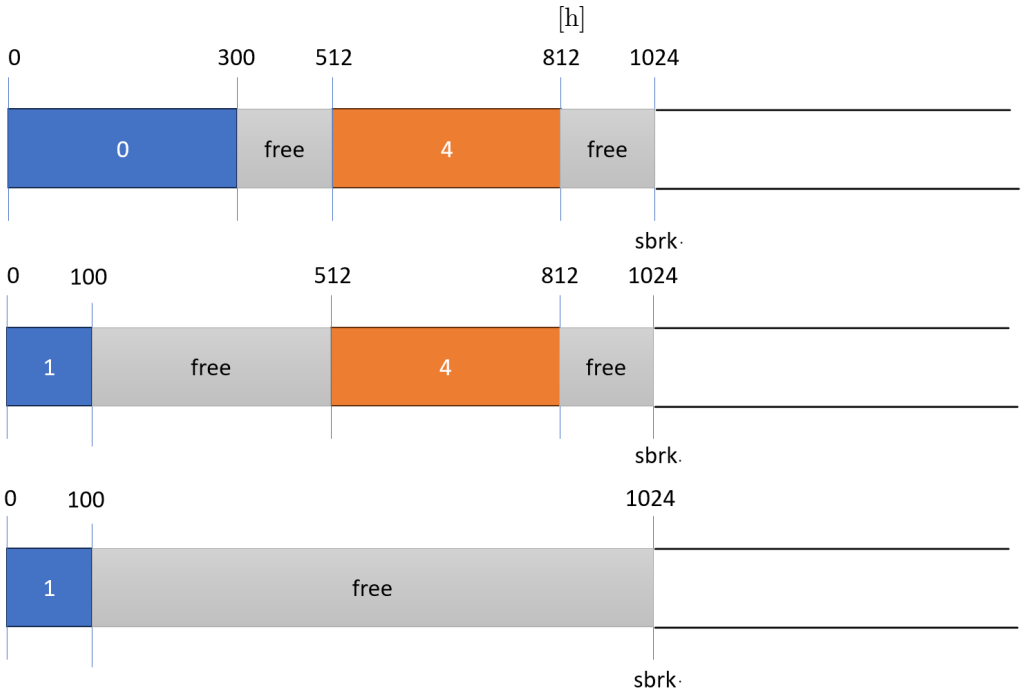


Figure 3.15: *Memory Manage for process p0s.*

Input File os_1_mlq_paging

Time slot 19.

```
2 4 4
1048576 16777216 0 0 0
1 p0s  130
2 s3  39
4 m1s  15
6 s2  120
```

We only keep 4 process for explanation simplicity After the Promgram run, we can see the paging structure of the OS.

Final page table of p0s process

```
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
```

Final page table of m1s process

```
============== PHYSICAL MEMORY AFTER DEALLOCATION ==============
PID=3 - Region=1
print_pgtbl: 0 - 512
00000000: 80000005
00000004: 80000004
Page number: 0 -> Frame Number : 5
Page number: 1 -> Frame Number : 4
================================================================
```

[h]

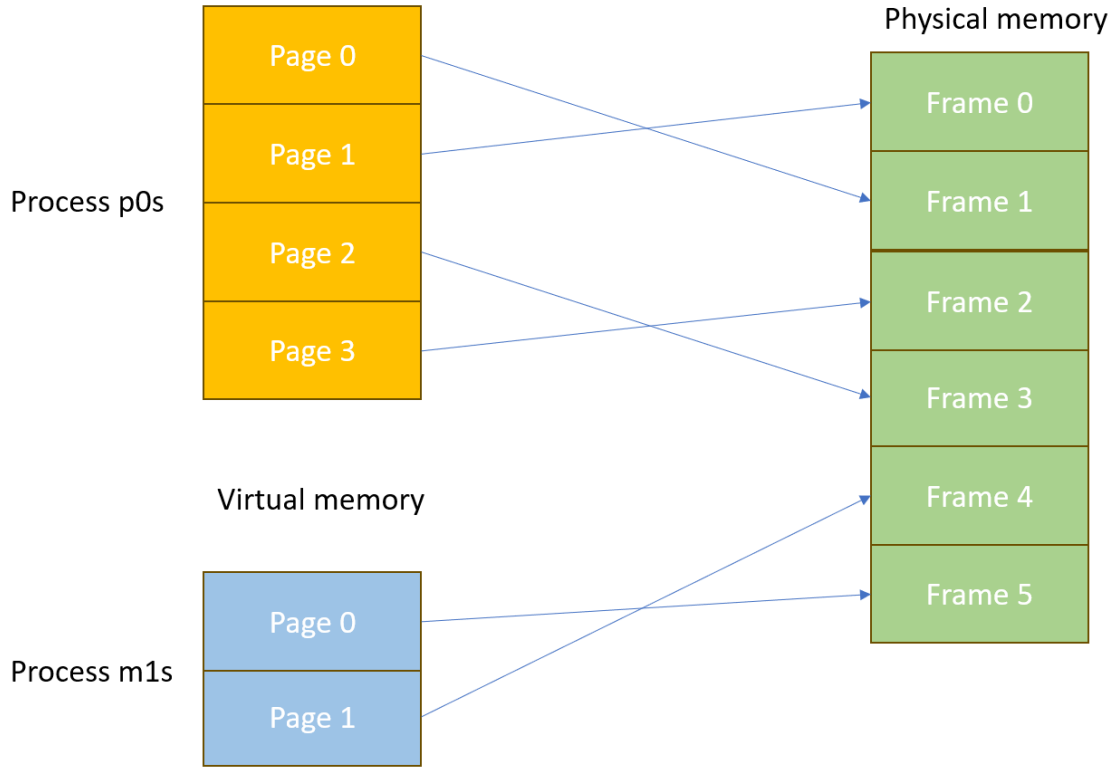


Figure 3.16: *Paging for os_1_mlq_paging*

This memory management algorithm offers a straightforward and powerful approach to handling heap memory. It reduces fragmentation, prevents unnecessary memory growth, and enables dynamic reuse through free list tracking and memory coalescing. By visualizing each step, we gain a clear understanding of how allocations, deallocations, and system optimizations work together to promote efficient memory usage — a fundamental principle in operating system design and systems programming.

Chapter 4

System Call

4.1 Theoretical Basis

System call provide an interface to the services made available by an operating system. It has a mechanism that allows users to switch from user mode to kernel mode, or allows user programs to access services from the kernel through **Software interrupt**(trap or exception). So it acts as the only one intermediary between user applications and hardware, ensuring that the operating system controls all resources and maintains stability and security.

Even simple programs may use a lot of system call. Therefor, a systems execute a thousands of system calls per second. And not all programmers fully understand and know how to use them. So that, normally, they design a program through an **Application Programming Interface (API)**. API is a set of function that provide for user programs to interact with the operating system. A program (user) send a request to API, then API will transform it to the system call can understand and let kernel do it. API is easier to use than using system calls directly; and it has portability, any program using the same API can compile and run.

Another important factor is **Run-Time Environment (RTE)**, it is the full set of software that a program needs to execute applications. It serves as the link to system calls like such as memory management, multithreading, communication with the operating system, and code execution. But the programmer do not need to know what it does, instead of that, we just need to follow the API and understand what the operating system will do. Thus, the programmer hardly interacts with system calls because everything is managed by the RTE.

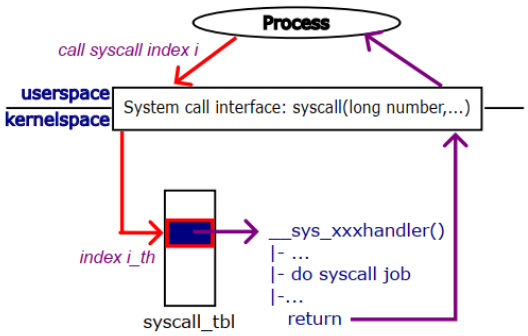


Figure 4.1: *The handling of user application invoking a system call*

4.2 Types of System Call

System calls can be grouped into six major categories: Process control, File management, device management, information maintenance, communications, and protection.

	Windows	Unix
Process control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File management	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device management	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communications	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

Figure 4.2: *Windows and Unix System Calls*

4.2.1 Process control

Process Control is a core component of the operating system (OS), allowing the creation, execution, and termination of processes. A process represents a running program, managed through system calls that interact with the OS kernel.

A process has two ways to terminate:

- **Normally:** Using `exit()`, and return status code of 0 (success).
- **Abnormally:** Using `abort()` or the program have problem and causes an error trap. Then the operating system make a dump of memory and write error message to examine by debugger tool. The severity is determined by the parameter (e.g., level 0 is normal, higher levels indicate an error).

After that, the operating system must transfer control to the invoking command interpreter. It will wait for the next command. In an interactive system, the command interpreter simply continues with the next command. In GUI system, the command interpreter will pop up with an error message window.
A process can load another program via command `load()` and `execute()`. It can return the control right to the root process after save the memory state of current process, or create new process by `create_process()` to run concurrently (multiprogramming). With each of process, we can adjust the maximum execution time through `get_process_attributes()`, `set_process_attributes()` commands. Or `terminate_process()` to terminate a not needed or error process. Using `Wait_time()`, `wait_event()` and `signal_event()` to synchronize the process.
To share date among the process, we use Lock to ensure integrity of shared data, `accquire_lock()`; `release_lock()`.

4.2.2 File Management

File management system calls handle the creation, manipulation, and organization of files and directories. Key operations include:

- **Creation and Deletion**: `create()` and `delete()` generate or remove files, requiring a file name and optional attributes.
- **Access and Modification**: `open()`, `read()`, `write()`, and `close()` manage file access, while `reposition()` adjusts the file pointer (e.g., skipping to the end).
- **Directory Operations**: Directories use similar calls for organizing files, such as creating hierarchical structures.
- **Attributes**: `get_file_attributes()` and `set_file_attributes()` retrieve or modify metadata (e.g., permissions, timestamps). Some OSes extend functionality with `move()` or `copy()` via APIs or system utilities.

This abstraction simplifies storage interactions, treating files as logical units regardless of physical storage details.

4.2.3 Device Management

Devices, whether physical (disks) or virtual (network sockets), are managed similarly to files:

- **Exclusive Access**: `request()` and `release()` ensure controlled access to devices, preventing conflicts.
- **Unified I/O**: Systems like UNIX merge files and devices into a single structure. For example, a printer might be accessed via a special file (e.g., `/dev/lp0`), using standard `read()`/`write()` calls.
- **Risks**: Unmanaged access can lead to contention or deadlock, necessitating careful resource allocation.

4.2.4 Information Maintenance

These calls facilitate data exchange between user programs and the OS:

- **System Data**: `get_time()`, `get_version()`, and `free_memory()` provide real-time system status.
- **Debugging Tools**: `dump()` captures memory snapshots for analysis, while profilers track execution time via interrupts or tracing.
- **Process Metadata**: `get_process_attributes()` and `set_process_attributes()` access or modify process details (e.g., priority, execution limits).

Such calls are vital for debugging, optimization, and system monitoring.

4.2.5 Communication

Interprocess communication (IPC) occurs through two models:

4.2.5.1 Message-Passing

Processes exchange data via messages, either directly or through mailboxes.

- **Steps**: Resolve identifiers(`get_hostid()`, `get_processid()`), establish connections (`open_connection()`), and transfer data (`read_message()`, `write_message()`).

Message-passing is common in networked systems for simplicity and low conflict risk.

4.2.5.2 Shared Memory

Processes collaboratively create shared memory regions (`shared_memory_create()`, `shared_memory_attach()`), enabling high-speed data access.

Requires explicit synchronization to prevent concurrent write conflicts.

Both models coexist in modern OSes, balancing speed (shared memory) and safety (message-passing).

4.2.6 Protection

Protection mechanisms regulate resource access, critical in multi-user and networked environments:

- **Permissions**: `set_permission()` and `get_permission()` manage access rights for files, devices, or memory.
- **User Control**: `allow_user()` and `deny_user()` grant or revoke privileges dynamically.
- **Security vs. Protection**: While protection focuses on internal access control, security extends to external threats (e.g., malware, breaches).

These calls are foundational for maintaining system integrity, especially in distributed systems.

Question 5

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

Answer:

In the simple OS, there are limited numbers of registers to be used according to section 1.3, such that arguments are passed to system calls. For example, `memmap` with syntax SYSCALL 17 SYSMEM_OP REG_ARG2 REG_ARG3, and `killall` with syntax SYSCALL 101 REGIONID. While these registers suffice for simple data types such as integers and pointers, they present a limitation when system calls require a larger number of arguments or involve complex data arguments such as data structures, strings, or large arrays that cannot fit within the register space.

This necessitates the implementation of an alternative mechanism to facilitate the transfer of more extensive or intricate information between user space and the kernel, called **Pointer-based passing** in which the **complex argument is stored in the user's memory space** and **passed by reference (pointer) to that memory** via available registers. The system call interface in the kernel receives the pointer and accesses the memory.

In the code snippet of sys_killall.c:

```
uint32_t memrg = regs->a1;
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++;
}
```

Region IDs are stored in registers regs->a1. The system call handler then uses libread() to retrieve the actual data byte by byte from the specified memory region, as `killall` uses a memory region to store the program name associated with `REGIONID`. In the assignment, the `memmap` structure also suggests the practice of Pointer-based passing. Instead of passing the entire large, complex arguments, pointers (point to memory space of the arguments and stored in REG_ARG2 REG_ARG3) are passed by reference to the kernel via registers. Upon arriving, the system dereferences the pointers to retrieve and process actual data.

Passing complex arguments to system calls through memory references ensures a robust communication bridge between user application and the OS kernel. Take a close look at the struct sc_regs in syscall.h:

```
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 on entry for:
         * - the syscall number (syscall, sysenter, int80)
         * - error_code stored by the CPU on traps and exceptions
         * - the interrupt number for device interrupts
         */
    uint32_t orig_ax;
    int32_t flags;
};
```

All syscall arguments pass through the 32-bit registers (a1-a6) in sc_regs, and the different syscalls use these registers differently. For `killall`, a1 contains a region ID, which is used with libread. For `memmap`, registers can contain pointers to structures that the kernel then dereferences, or an OPCODE (like SYSMEM_SWP_OP). In all of these handlers, no large struct or array is packed directly in registers, ensuring the complex, heavy data structures stay in user memory while actual data can still be accessed in the kernel by reference.

Question 6

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

Answer:

When a syscall implementation takes too long to execute in the OS, several important consequences arise:

- **Scheduler starvation**
  Syscalls execute in kernel context at elevated priority, often with preemption or interrupts disabled. The MLQ scheduler (sched.c) cannot interrupt a long-running syscall, which means other processes in the ready queue remain waiting even if they have higher priority, causing potential starvation. A long-running handler prevents the scheduler from running other processes, therefore, user tasks get starved and system responsiveness is reduced significantly.

- **Resource contention and deadlocks**
  Long syscalls often hold locks too long that can block other kernel subsystems, resulting in deadlocks and resource starvation. For example, the mutex locks (like mmvm_lock in libmem.c) for critical sections held during syscall execution block other processes trying to access the same resources. A prolonged syscall increases the risk of deadlock if processes are waiting on each other's resources.

- **Time slice scheduling discruption**
  The scheduler allocates time based on priority (slot[i] = MAX_PRIO - i in sched.c). A lengthy syscall not only disrupts the time slice scheduling but creates disproportionate starvation for lower-priority processes. Since high-priority queues already receive more CPU time by design, any additional delay caused by long syscalls further reduces the already limited execution opportunities for low-priority processes. This amplifies the unfairness in the MLQ scheduling algorithm, potentially causing complete starvation of processes with low priorities (high numerical values) even after the long syscall completes, as high-priority processes would continue to monopolize the CPU.

- **Memory Congestion**
  Long-running memory syscalls (e.g., in libmem.c) monopolize the memory subsystem. Since memory access is serialized through these syscalls, other processes—even on other CPUs—stall waiting for memory operations, creating a system-wide bottleneck.

4.3 Algorithm

4.3.1 Sys_lstsyscall

This file will list all system call that the wrapper (API) can support. **Sysnopsis: SYSCALL 0**

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

4.3.2 Sys_memmap

It can supports various operations on mapping memory, through the operations, which are described in `libmem.h`. It sysnopsis is: **SYSCALL 17 SYSMEM_OP REG_ARG2 REG_ARG3**

```
    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;
   }
```

4.3.3 Sys_killall

Sysnopsis: **SYSCALL 101 REGIONID** Killall will take the REGIONID then send the signals to all processes that are running. If the process have the name match with the string stored in REGIONID. We can divide it into two parts:

- First: we use while and libread in libmem.h to take the proc_name.

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

    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++;
    }
```

- Second: We still use loop to search until the end of the running list. To find process that match with proc_name. Except itself the caller.

```
    struct queue_t *running_list = caller->running_list;
    struct queue_t temp_queue = {0};
    int idx = 0;

    while (!empty(running_list))
    {
        struct pcb_t *proc = dequeue(running_list);
        char *name = strrchr(proc->path, '/');
        if (!name) name = strrchr(proc->path, '\\');
        if (name) name++;

        if (strcmp(proc->path, proc_name) == 0)
        {
            if (proc->pid == caller->pid) continue;
            libfree(proc, idx);
        }
        else enqueue(&temp_queue, proc);
        idx++;
    }

    while (!empty(&temp_queue))
    {
        struct pcb_t *proc = dequeue(&temp_queue);
        enqueue(running_list, proc);
    }
```

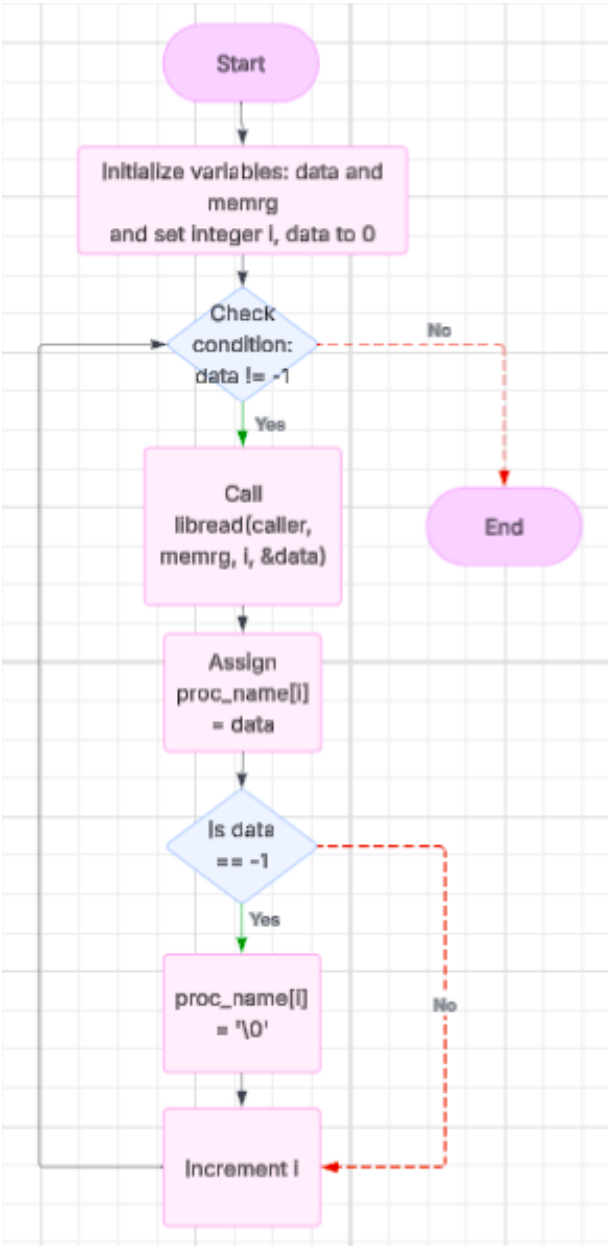


Figure 4.3: *Flowchart of find proc_ name*

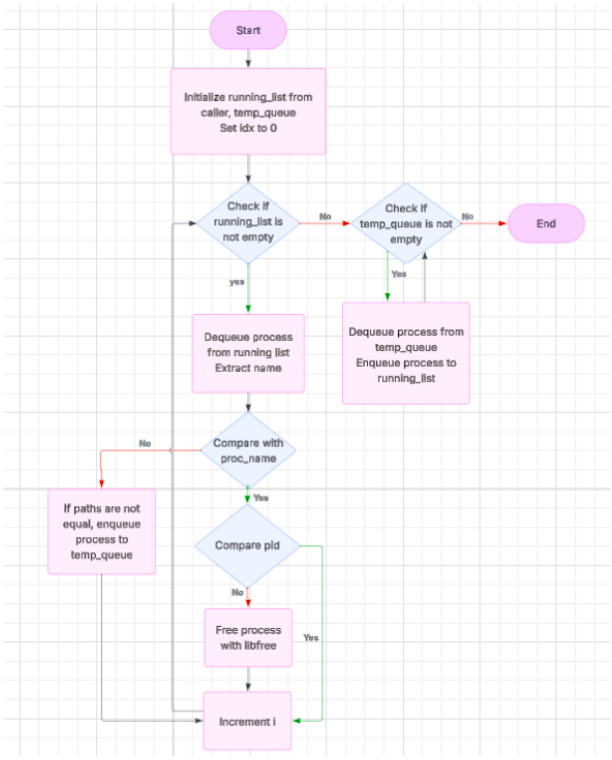


Figure 4.4: *Flowchart of send signal*

4.3.4 Execution Output

This section presents the execution output of the system call implementation across different scenarios: `os_scnucle`, `os_syscall`, and `os_syscall_list`.

Output for os_sc:

```
dmin@Admin-PC:/mnt/c/Users/Admin/Desktop/Code  n p$  ./os os_sc
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
Time slot   9
Time slot  10
        CPU 0: Dispatched process  1
Time slot  11
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Output for os_syscall:

```
dmin@Admin-PC:/mnt/c/Users/Admin/Desktop/Code  n p$  ./os os_syscall
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/sc2, PID: 1 PRIO: 15
Time slot   9
Time slot  10
        CPU 0: Dispatched process  1
================ PHYSICAL MEMORY AFTER ALLOCATION ==============
PID=1 - Region=1 - Address=00000000 - Size=100 byte
print_pgtbl: 0 - 256
00000000: 80000000
Page number: 0 -> Frame Number : 0
================================================================
Time slot  11
================= PHYSICAL MEMORY AFTER WRITING ================
write region=1 offset=0 value=80
print_pgtbl: 0 - 256
00000000: 80000000
Page number: 0 -> Frame Number : 0
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 80
===== PHYSICAL MEMORY END-DUMP =====
Time slot  12
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
================= PHYSICAL MEMORY AFTER WRITING ================
write region=1 offset=1 value=48
print_pgtbl: 0 - 256
00000000: 80000000
Page number: 0 -> Frame Number : 0
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 80
BYTE 00000001: 48
===== PHYSICAL MEMORY END-DUMP =====
Time slot  13
================= PHYSICAL MEMORY AFTER WRITING ================
write region=1 offset=2 value=-1
print_pgtbl: 0 - 256
00000000: 80000000
Page number: 0 -> Frame Number : 0
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 80
BYTE 00000001: 48
BYTE 00000002: -1
===== PHYSICAL MEMORY END-DUMP =====
Time slot  14
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
================ PHYSICAL MEMORY AFTER READING ================
read region=1 offset=0 value=80
print_pgtbl: 0 - 256
00000000: 80000000
Page number: 0 -> Frame Number : 0
================================================================
read region=1 offset=0 value=80
print_pgtbl: 0 - 256
```

```
00000000: 80000000
Page number: 0 -> Frame Number : 0
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 80
BYTE 00000001: 48
BYTE 00000002: -1
===== PHYSICAL MEMORY END-DUMP =====
Data 0: 80
================= PHYSICAL MEMORY AFTER READING ================
read region=1 offset=1 value=48
print_pgtbl: 0 - 256
00000000: 80000000
Page number: 0 -> Frame Number : 0
================================================================
read region=1 offset=1 value=48
print_pgtbl: 0 - 256
00000000: 80000000
Page number: 0 -> Frame Number : 0
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 80
BYTE 00000001: 48
BYTE 00000002: -1
===== PHYSICAL MEMORY END-DUMP =====
Data 1: 48
================= PHYSICAL MEMORY AFTER READING ================
read region=1 offset=2 value=-1
print_pgtbl: 0 - 256
00000000: 80000000
Page number: 0 -> Frame Number : 0
================================================================
read region=1 offset=2 value=-1
print_pgtbl: 0 - 256
00000000: 80000000
Page number: 0 -> Frame Number : 0
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 80
BYTE 00000001: 48
BYTE 00000002: -1
===== PHYSICAL MEMORY END-DUMP =====
Data 2: -1
The procname retrieved from memregionid 1 is "P0"
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 80
BYTE 00000001: 48
BYTE 00000002: -1
===== PHYSICAL MEMORY END-DUMP =====
Data 2: -1
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 80
BYTE 00000001: 48
BYTE 00000002: -1
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 80
BYTE 00000001: 48
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 80
BYTE 00000001: 48
BYTE 00000002: -1
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 80
BYTE 00000001: 48
BYTE 00000002: -1
```

```
===== PHYSICAL MEMORY END-DUMP =====
Data 2: -1
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 80
BYTE 00000001: 48
BYTE 00000002: -1
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 80
BYTE 00000001: 48
BYTE 00000000: 80
BYTE 00000001: 48
BYTE 00000002: -1
BYTE 00000002: -1
===== PHYSICAL MEMORY END-DUMP =====
===== PHYSICAL MEMORY END-DUMP =====
Data 2: -1
The procname retrieved from memregionid 1 is "P0"
Time slot  15
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Output for os_syscall_list:

```
dmin@Admin-PC:/mnt/c/Users/Admin/Desktop/Code  n p$  ./os os_syscall_list
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
        Loaded a process at input/proc/sc1, PID: 1 PRIO: 15
Time slot  10
        CPU 0: Dispatched process  1
0-sys_listsyscall
17-sys_memmap
101-sys_killall
Time slot  11
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Chapter 5

Put It All Together

5.1 Theoretical Basis

Synchronization is the coordination of multiple processes or threads to ensure safe access to shared resources in an operating system, such as CPU queues, memory frames, or process control blocks (PCBs). In a multi-processor environment, concurrent access to these resources without proper synchronization can lead to data inconsistencies, system crashes, or unpredictable behavior. In our simple operating system, synchronization is crucial for integrating the multi-level queue (MLQ) scheduler (Section 2.1), paging-based memory management (Section 2.2), and system calls (Section 2.3). By using synchronization mechanisms like mutex locks, we ensure that shared resources are accessed safely, maintaining system reliability and correctness.

5.1.1 Why Synchronization Matters?

In our system, multiple CPUs may concurrently enqueue or dequeue processes from the ready queues, or multiple processes may request memory frames from the RAM device. Without synchronization, these operations could corrupt the queue state or allocate the same memory frame to multiple processes, leading to system failures.

Example: The readers-writers problem illustrates a scenario where multiple processes attempt to read from or write to a shared memory region. Without synchronization, a writer could modify the memory while a reader is accessing it, resulting in inconsistent data.

5.1.2 Common Synchronization Challenges

Race Condition

A race condition occurs when the outcome of a program depends on the unpredictable order of execution of processes accessing a shared resource. For instance, in our MLQ scheduler, if two CPUs dequeue a process from the same ready queue simultaneously without synchronization, one CPU may receive an invalid PCB, causing a segmentation fault.

Deadlock

Deadlock happens when multiple processes hold resources and wait for each other to release additional resources, forming a circular dependency. In our memory management system, a deadlock could occur if two processes each hold a memory frame and wait for the other to free another frame. For example, Process A holds frame 0 and waits for frame 1, while Process B holds frame 1 and waits for frame 0. Without proper resource allocation ordering, both processes will stall indefinitely.

Starvation

Starvation occurs when a process is perpetually denied access to a resource due to higher-priority processes monopolizing it. In our MLQ scheduler, low-priority processes in lower-priority queues may starve if high-priority processes continuously arrive.

5.1.3 Critical Sections and Mutex Locks

A critical section is a code segment that accesses shared resources and must be executed exclusively by one process at a time. In our system, critical sections include the enqueue and dequeue functions in the MLQ scheduler (Section 2.1) and the memory allocation functions in the paging subsystem (Section 2.2).

To protect critical sections, we use mutex locks, which ensure that only one process can access a shared resource at a time. A mutex is locked before entering a critical section and unlocked afterward, preventing concurrent access. Comparison of some Synchronization Mechanisms:

- Peterson's Solution: Effective for two processes but impractical for multiple CPUs.
- Semaphores: Suitable for counting resources but complex to implement correctly.
- Monitors: High-level abstraction but not supported in our C-based system.
- Mutex Locks: Simple, efficient, and ideal for our MLQ scheduler and memory management.

We chose mutex locks for their simplicity and compatibility with our system's C-based implementation using the POSIX threads library - pthread.

Question 8

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:

Without synchronization, processes or threads on different CPUs may simultaneously access or modify shared resources (such as queues, memory, or timers). This can lead to unpredictable issues, including race conditions, data corruption, segmentation faults, double frees, and even system crashes.

Example from Simple OS:

If `pthread_mutex_lock` and `pthread_mutex_unlock` are not used to protect access to the process queue (`queue_t`), critical problems can occur. Consider the following scenario without synchronization:

```
// Example without lock
enqueue(&ready_queue, proc); // CPU 0
dequeue(&ready_queue);       // CPU 1
```

In this situation:

- CPU 0 writes a new process into `ready_queue->proc[size]` and increments `size`.
- At the same time, CPU 1 reads from `ready_queue->proc[0]` and decrements `size`.

This simultaneous access can result in incorrect `size` values, overwriting of unread elements, data loss, or other critical failures.

Observed Effects in the Assignment:

- Processes may be dispatched multiple times if `get_proc()` and `put_proc()` are not properly synchronized.
- A process could be retrieved by multiple CPUs simultaneously or become stuck in the queue, leading to repeated entries in the system log.

CPU dispatching procces repeatedly: Segmentation fault / Double free: If a process is free()d from a syscall (e.g., killall) while being accessed by the CPU, it can cause a segmentation fault or double free:
– CPU holds the proc pointer and calls run(proc), leading to a segmentation fault.
– CPU calls free(proc) again, resulting in a double free

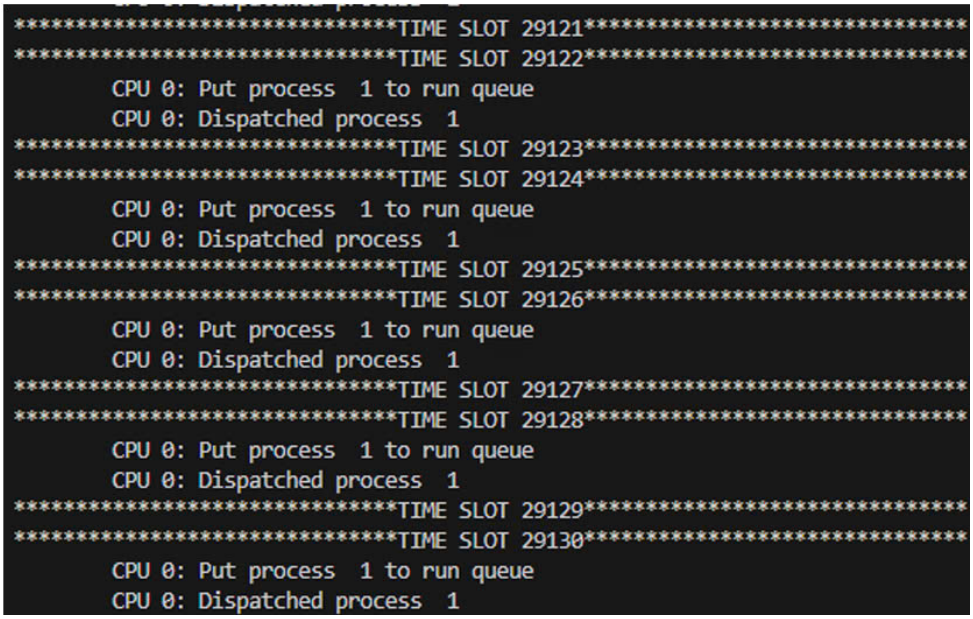
[H]


Figure 5.1: *Infinite loop*

[H]


Figure 5.2: *Double free*

Conclusion: Impact of Missing Synchronization

Without proper synchronization mechanisms, the following serious issues may arise:

- **CPU crash / Segmentation fault:** Caused by concurrent access to critical memory regions such as `proc` or `queue`.
- **Race condition:** Occurs when locks are missing for shared resources like queues, memory blocks, or timers, leading to unpredictable behavior.
- **Infinite loop:** Results from inconsistent updates to `queue->size`, potentially causing the system to hang.
- **Double free:** Happens when `proc` is freed simultaneously from a system call and from CPU processing, causing memory corruption.