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

Operating System (CO2018)

CLASS CC06

ASSIGNMENT REPORT

Advisor(s):	Diệp Thanh Đăng	
Student(s):	Lê Minh Hào (Group Leader)	ID: 2310846
	Nguyễn Hiếu	ID: 2352327
	Nguyễn Thanh Thảo	ID: 2353114
	Huỳnh Xuân Khương	ID: 2352636
	Nguyễn Hoàng Phương Anh	ID: 2310100

HO CHI MINH CITY, APRIL 2025

MEMBER LIST

CLASS CC06 - GROUP 03

No.	Fullname	Student ID	Problems	% done
1	Lê Minh Hào	2310846	- Put It All Together (Section 2.4) - Report LaTeX	100%
2	Nguyễn Hiếu	2352327	- System Call (Section 1.3 & 2.3) - Report LaTeX	100%
3	Nguyễn Hoàng Phương Anh	2310100	- Scheduler (Section 1.1 & 2.1)	100%
4	Nguyễn Thanh Thảo	2353114	- Memory Management (Section 1.2 & 2.2)	100%
4	Huỳnh Xuân Khương	2352636	- Memory Management (Section 1.2 & 2.2)	100%

Table 0.1: Member list & workload

Contents

1 Background **5**
1.1 Scheduling . . . 5
1.1.1 Scheduling Concepts . . . 5
1.1.2 Multi-level Queue (MLQ) . . . 6
1.2 Memory Management . . . 6
1.2.1 Virtual Memory Mapping . . . 6
1.2.2 Physical Memory . . . 8
1.2.3 Paging . . . 8
1.2.4 Translation Lookaside Buffer (TLB) . . . 11
1.2.4.1 Definition . . . 11
1.2.4.2 Fundamental works of TLB . . . 11
1.2.4.3 Advanced development . . . 12
1.3 System Call . . . 12

2 Implementation **13**
2.1 Scheduler . . . 13
2.1.1 Scheduler design . . . 13
2.1.2 Implementation . . . 14
2.1.3 Question Answering . . . 18
2.2 Memory Management . . . 19
2.2.1 Virtual Memory Regions . . . 19
2.2.2 Virtual Memory Areas . . . 21
2.2.3 Memory Allocation/Deallocation . . . 24
2.2.4 Memory Device Operations . . . 31
2.2.5 Page Table Operations . . . 32
2.2.6 Status of Memory Allocation in Heap and Data Segments . . . 33
2.2.7 Question Answering . . . 36
2.3 System Call . . . 40
2.3.1 Overview . . . 40
2.3.2 System calls in assignment . . . 40
2.3.3 Testing . . . 45
2.3.4 Question Answering . . . 49
2.4 Put It All Together . . . 51

List of Figures

1.1 The example of scheduler 5
1.2 CPU address 7
1.3 Page Table Entry Format 9
1.4 System call 'open' flow from user mode to kernel mode 13
2.2 Gantt diagram of how processes are executed by the CPU in test `sched_0` 17
2.4 Gantt diagram of how processes are executed by the CPU in test `sched` . . 18
2.5 Example ouput from os_mlq_1_paging 33
2.6 Example ouput from os_mlq_1_paging 34
2.7 Example ouput from os_mlq_1_paging 34
2.8 Example ouput from os_mlq_1_paging 35
2.9 Example ouput from os_mlq_1_paging 35
2.10 Test result 1 of scheduler without lock mechanism implementation 51
2.11 Test result 2 of scheduler without lock mechanism implementation 52
2.12 Race condition caused by not using lock during memory allocation 55

List of Tables

0.1 Member list & workload 2
2.1 Frame allocation for processor 55

1 Background

1.1 Scheduling

Scheduling is a foundational principle in operating systems, encompassing the distribution of CPU (Central Processing Unit) time among processes in a multitasking setting. The scheduler determines the sequence in which processes should execute, considering factors like priority, time allocation, or a blend of parameters, with the aim of enhancing system efficiency and making the most of available resources.

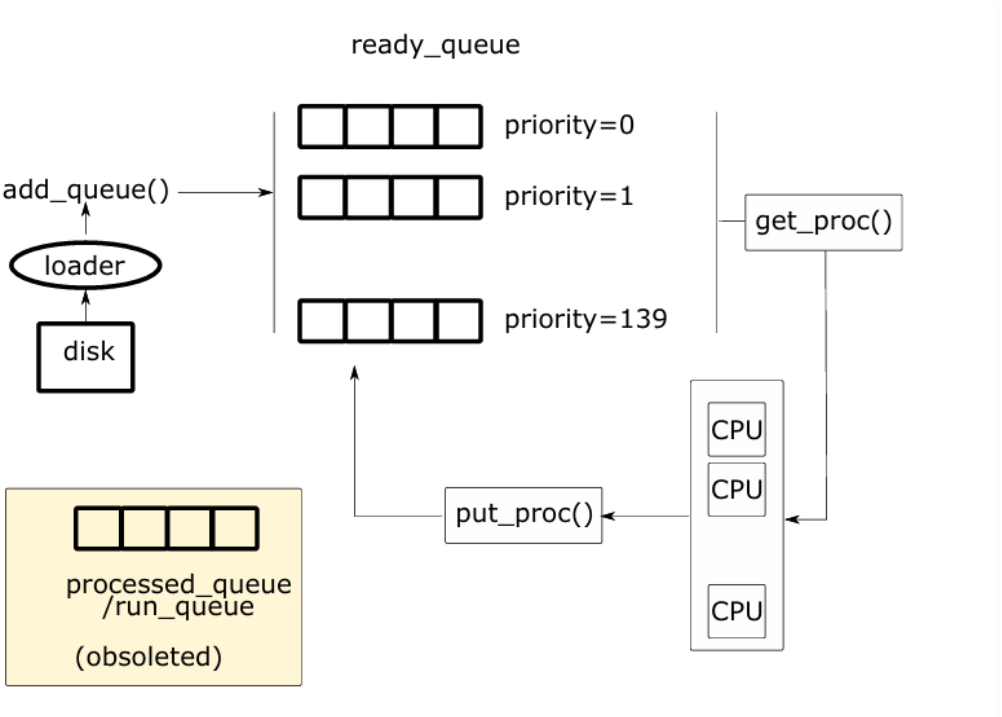


Figure 1.1: The example of scheduler

1.1.1 Scheduling Concepts

- CPU Utilization: Ensuring the CPU is fully utilized to maximize the execution of processes.
- Throughput: Maximizing the rate at which processes are completed within a specific timeframe.
- Turnaround Time: The overall duration it takes for a process to execute from the moment of submission to its completion. - Response Time: The interval between submitting a request and receiving the initial response.

- Fairness: Guaranteeing equitable distribution of CPU time among all processes to prevent any instances of starvation.
- Preemption: The capability to temporarily halt or interrupt the execution of a process in order to allocate CPU time to another process with higher priority.
- Context Switching: The additional workload associated with transitioning between processes, which involves preserving and reinstating their respective states.
- Scheduling Algorithms: Various methodologies like First-Come, First-Served (FCFS), Shortest Job Next (SJN), Round Robin (RR), Priority Scheduling, etc., are utilized to regulate the sequence in which processes are executed.

1.1.2 Multi-level Queue (MLQ)

- Multi-Level Queue (MLQ) represents a scheduling strategy that categorizes processes into numerous priority queues, each employing its own scheduling algorithm. Processes are sorted into queues based on attributes like priority, type, or other relevant criteria.
- Priority Levels: Processes are segmented into distinct priority levels, with each queue assigned a specific priority. For instance, real-time tasks might hold higher priority status compared to batch processing assignments.
- Queue Characteristics: Each queue has its unique scheduling algorithm, such as FCFS, SJN, or RR, tailored to suit the demands and characteristics of the processes housed within that queue.
- Dispatching: The selection of processes for execution is contingent upon the scheduling algorithm applied to their respective queue. Typically, higher priority queues are served first, ensuring critical processes receive timely CPU allocation.

1.2 Memory Management

1.2.1 Virtual Memory Mapping

Each process has a private virtual memory space structured into multiple areas, managed by the `mm_struct` structure. These areas, such as code, heap, and stack, exist within the address range `[vm_start, vm_end]` and are tracked using linked lists that manage both allocated and free regions.

Memory Area

A memory area is allocated continuously from `vm_start` to `vm_end`, but actual usage only reaches up to the program break (`sbrk`). Allocated regions are managed using

`vm_rg_struct`, while free regions are maintained in a linked list.

Memory Region

Each region represents a variable or logical memory block. These are stored in a fixed-size array called `symrgtbl`, with each entry containing the start and end addresses. This array functions as a simplified symbol table for tracking memory segments.

Memory Mapping

All memory regions are referenced in the `mm_struct` through a list (`mmap_list`). This structure also contains the `pgd` (page directory), which maps virtual pages to physical frames in the paging system. Other fields are included to support additional user-level memory operations.

CPU Addresses

In paging systems, CPU-generated addresses are divided into:

- **Page number (p)**: an index into the page table that identifies the base physical frame.
- **Page offset (d)**: added to the frame's base address to determine the exact physical location.

Physical memory is divided into fixed-size blocks (frames), such as 256B or 512B. For example, with a 22-bit CPU and 256B page size, the address is split accordingly for translation.

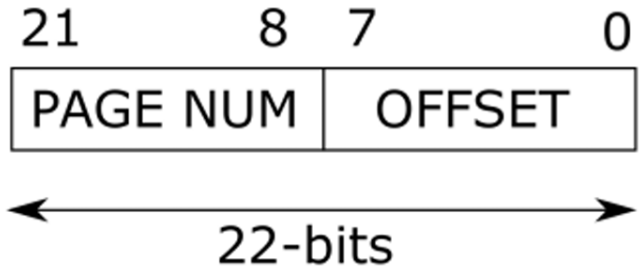


Figure 1.2: CPU address

1.2.2 Physical Memory

Virtual memory is ultimately mapped to physical hardware like RAM or SWAP. Although each process maintains separate memory mappings, they all reference the same underlying physical device.

Memory Devices

- **RAM:** Main memory directly accessible by the CPU.
- **SWAP:** Secondary memory used to offload inactive data from RAM. It cannot be accessed directly by the CPU and must be paged in and out through RAM.

Configuration

The system supports one RAM and up to four SWAP devices. These devices may vary in size, access type (random or sequential), and performance characteristics. This flexibility allows simulation of different hardware setups.

Data Structures

- `framephy_struct`: stores frame-specific information, including the frame number.
- `memphy_struct`: describes the physical memory, including total size, access mode (`rdmflg`), and lists of free and used frames (`free_fp_list`, `used_fp_list`).

These structures support memory allocation and basic operations like reading, writing, and swapping between RAM and SWAP.

1.2.3 Paging

The translation mechanism accommodates both segmentation and segmentation with paging. In this iteration, we've implemented a single-level paging system that utilizes nearly one RAM device and one SWAP instance of hardware. Although our codebase includes the functionality for multiple memory segments, we primarily focus on the initial segment, identified as the sole segment of **vm_area** with **vmaid = 0**. Future iterations will incorporate more robust paging schemes to handle multiple segments or potential overlap/non-overlap between segments.

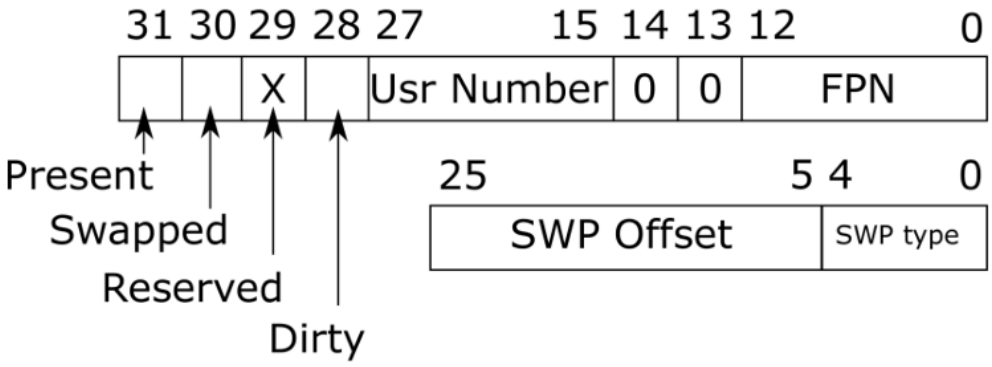


Figure 1.3: Page Table Entry Format

Page table The structure in Figure 5 facilitates userspace processes in determining the mapping of each virtual page to a physical frame. It comprises a 32-bit value for each virtual page, encapsulating the following information:

* Bits	0–12	page frame number (FPN) if present
* Bits	13–14	zero if present
* Bits	15–27	user-defined numbering if present
* Bits	0–4	swap type if swapped 5
* Bits	5–25	swap offset if swapped
* Bit	28	dirty
* Bit	29	reserved
* Bit	30	swapped
* Bit	31	presented

Each entity's virtual space is isolated, with each struct **pcb_t** containing its own table. To operate within a paging-based memory system, we need to update this structure, which will be discussed in the subsequent section. In all scenarios, every process possesses a completely isolated and unique space. In our setup with N processes, this results in N page tables, each of which must encompass entries for the entire CPU address space. For each entry, the paging number may be associated with a frame in either MEMRAM or MEMSWP, or it may have a null value. The functionality of each data bit within the page table entry is outlined in Figure 5. In our chosen configuration highlighted in Table 1, we have a 16,000-entry table, with each table requiring 64 KB of storage space.

In section 1.2.1, the process accesses the virtual memory space in a contiguous manner using the **vm_area** structure. The remaining task involves mapping between pages and frames to ensure contiguous memory space over a discrete frame storage mechanism. This process falls into two primary approaches: memory swapping and basic memory operations such as allocation, deallocation, read, and write. These operations are primarily associated with the pgd page table structure.

Memory swapping: we've learned that a memory segment may not utilize its entire storage space, leaving some storage spaces unmapped to MEMRAM. Swapping assists in transferring the contents of physical frames between MEMRAM and MEMSWAP. Swapping involves copying the contents of a frame from an external source into main memory RAM. Conversely, swapping out involves moving the contents of a frame from MEMRAM to MEMSWAP. In a typical scenario, swapping helps free up RAM frames since the size of the SWAP device is usually ample.

Basic memory operations in paging-based system

- **ALLOC**: memory, in most cases, it typically fits within an available region. However, if there's no suitable space, we must raise the barrier sbrk. Since it has not been previously accessed, it may require providing some physical frames and then mapping them using Page Table Entries.

FREE: the storage space associated with the region ID. We're unable to reclaim the physical frames that were previously allocated, potentially leading to memory fragmentation. Instead, we retain the freed storage space in a free list for use in subsequent allocation requests.

READ/WRITE: bring a page into main memory, it's necessary to perform page swapping, which is often the most resource-intensive step. If the page resides in the MEMSWAP device, it must be transferred back to the MEMRAM device (swapping in). If there's insufficient space in MEMRAM, we may need to return some pages to the MEMSWAP device (swapping out) to create additional room.

To execute these operations, collaboration among the mm's modules is essential, as depicted in Figure 6.

1.2.4 Translation Lookaside Buffer (TLB)

1.2.4.1 Definition

In the context of operating systems and memory management subsystems, each process typically possesses its own page table, containing entries that map virtual pages to physical frame numbers. The challenge arises from the need to optimize access times for these entries, which incur a double access penalty compared to accessing data directly from memory (MEMPHY).

To mitigate this overhead, a Translation Lookaside Buffer (TLB) is introduced, leveraging its cache-like properties to provide faster access times. Essentially, the TLB acts as a high-speed cache for frequently accessed page table entries. However, it's important to note that memory cache is a costly component and therefore has limited capacity.

1.2.4.2 Fundamental works of TLB

TLB, or Translation Lookaside Buffer, functions as a specialized memory unit with built-in support for mapping mechanisms, allowing it to associate content with identifier information. Drawing from knowledge acquired in previous computer hardware courses, various cache mapping techniques such as direct-mapped, set associative, and fully associative are leveraged.

In terms of setup, TLB stores recently accessed page table entries. When the CPU generates a virtual address, it first checks the TLB. If a page table entry is found (TLB hit), the corresponding frame number is retrieved. Conversely, if a page table entry is not found (TLB miss), the page number is used to access the page table in main memory. In case the page is not present in main memory, a page fault occurs, and the TLB is updated with the new page entry.

During a TLB hit, the CPU generates a virtual address, which is then checked in the TLB. If the entry is present, the corresponding frame number is retrieved. In contrast, during a TLB miss, the CPU generates a virtual address, checks the TLB (finding no entry), matches the page number to the page table in main memory, and retrieves the corresponding frame number.

Given that TLB serves as a memory storage device, there's flexibility in designing an efficient cache mapping method. While no fixed design is imposed, a suggestion is provided based on pid (process identifier) and page number. It's important to note that since TLB cache is shared by all system processes and is intended for usage at the CPU level, the inclusion of pid is necessary.

1.2.4.3 Advanced development

In the realm of advanced development, real developer communities prioritize native memory cache methods over TLB, as manufacturers vie for superior performance and hit ratios. To enhance cache performance, meticulous examination of each cached content preservation is conducted.

1. Cache Generation

Before modifying or updating cache content, developers preserve a version or designated cache generation. This ensures that significant memory modifications are accounted for in the cache content. While advantageous, this feature can be excluded for basic requirements.

2. Locality Threshold

Developers add an entry to the TLB after memory retrieval reaches a specific threshold. This maintains cache coherence and ensures that allocated TLB entries align with the current locality.

3. Multilevel Page Tables

Multilevel page tables offer increased efficiency, particularly in scenarios with small or medium-sized virtual address spaces. TLB utilization is optimized by reducing the number of entries that need to be searched. Implementing multilevel page tables can be more challenging compared to other methods.

1.3 System Call

System calls are the fundamental interface between an application and the kernel of an operating system. In this project, we will implement several basic system calls and understand how they work.

System calls are typically not invoked directly; instead, they are accessed through wrapper functions provided by standard libraries. These wrappers are usually quite thin, merely copying the arguments into the appropriate registers before invoking the system call.

In this simplified operating system, we have system calls such as **listsyscall** (to list all available system calls), **memmap** (to perform memory mapping), and **killall** (to terminate processes by name).

Particularly, in this assignment, we will complete the implementation of the **killall** system call, which is used to terminate all processes whose program name matches the string stored in the **REGIONID**.

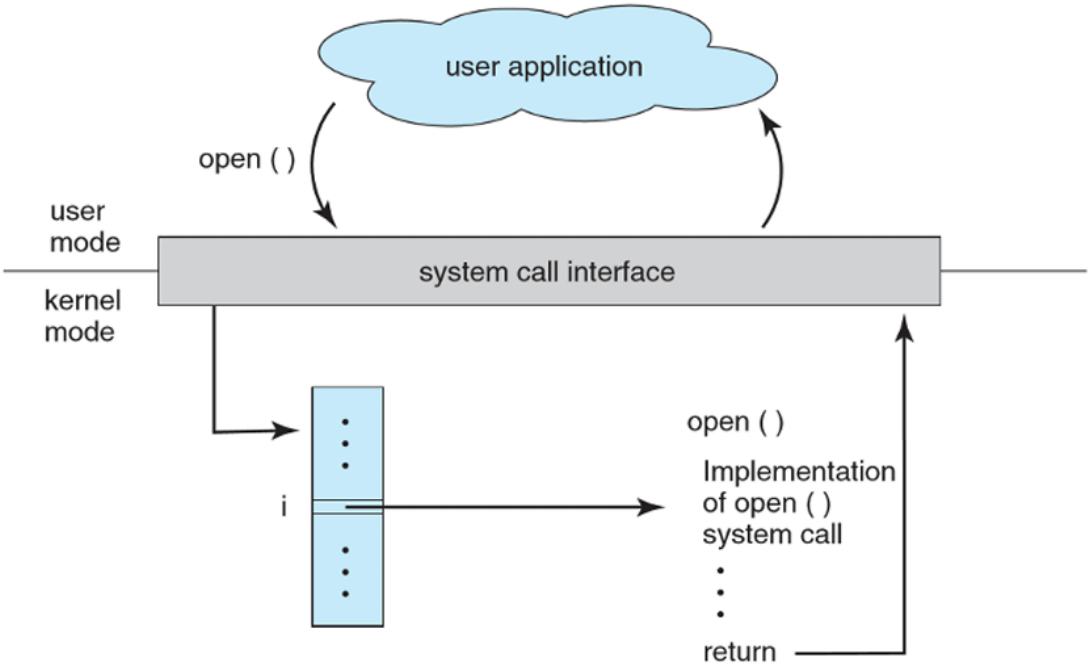


Figure 1.4: System call 'open' flow from user mode to kernel mode

Understanding and implementing system calls helps learners gain deeper insight into how the oprating system interacts with user applications and privides essential services to them.

2 Implementation

2.1 Scheduler

2.1.1 Scheduler design

The scheduler in this OS project is designed to work on multiple processors. We use a Multilevel Queue (MLQ) scheduling algorithm, where processes are organized into multiple queues based on their priority. The OS will have multiple queues called `mlq_ready_queue` to determine which process to run when a CPU becomes available. Each queue is associated with a fixed priority value. The scheduler is designed based on the "multilevel queue" algorithm. The CPU also runs processes in round-robin style. Each process is allowed to run in a time slice.

- **MLQ Scheduling:** A CPU scheduling algorithm that divides the ready queue into multiple queues, each with its own scheduling algorithm. Processes are assigned to these queues based on their characteristics, in this project – priority, and each queue has its own scheduling needs.

- **Slot-based Time slice:** Within each priority queue, processes are scheduled in First-Come-First-Serve (FCFS) order using round-robin. To manage fairness and avoid starvation, each queue is assigned a number of CPU time slots based on its priority. When the slots for all queues are exhausted, the system resets the slots, starting a new round of scheduling.

- **Fairness:** This mechanism ensures that even the lowest-priority processes are guaranteed at least one slot per round. No process is permanently starved while many high-priority jobs are active.

2.1.2 Implementation

a) Queue operation

- **enqueue:** Since we're using the MLQ with Round Robin inside each priority queue, we do not need to sort the priority of the processes inside the queue. This function simply checks if the current queue `q` is empty or full, then proceeds to add the new process `proc` to the end of the queue and increase the queue's size.

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

- **dequeue:** This function checks if the current queue is empty or full, gets the first process in the queue, then shifts all remaining elements left by one and returns the process that was at the front. Same as in enqueue, we do not need to find the highest priority process in the queue first.

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

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

b) Scheduler algorithm implementation

- **get_mlq_proc:** This is the core function of the MLQ scheduler, deciding which process to run next, based on the remaining time slots for each queue. Firstly, we dequeue the first process from the queue `mlq_ready_queue[i]` that holds processes of priority `i`, then decrease the time slot that queue currently has. If `proc == NULL`, it means that all the time slots for the queue this round have been exhausted, and we have to reset it to continue picking a process from a queue with refreshed time slots. The implementation is also supported with mutex lock to ensure safety and no race conditions.

```
struct pcb_t * get_mlq_proc(void) {
  struct pcb_t * proc = NULL;
  /*TODO: get a process from PRIORITY [ready_queue].
   * Remember to use lock to protect the queue.
   * */
  pthread_mutex_lock(&queue_lock);
  // 1st scan
  for (int i=0; i<MAX_PRIO; i++)
  {
    if (slot[i]>0 && !empty(&mlq_ready_queue[i]))
    {
      proc=dequeue(&mlq_ready_queue[i]);
      slot[i]--;
      break;
    }
  }
  if (proc==NULL)
  {
    for (int i=0; i<MAX_PRIO; i++)
    {
      slot[i]=MAX_PRIO-i;
    }
    // 2nd scan
    for (int i=0; i<MAX_PRIO; i++)
    {
      if (slot[i]>0 && !empty(&mlq_ready_queue[i]))
```

```
27          {
28            proc=dequeue(&mlq_ready_queue[i]);
29            slot[i]--;
30            break;
31          }
32        }
33    }
34    pthread_mutex_unlock(&queue_lock);
35    return proc;
36  }
```

c) Testing

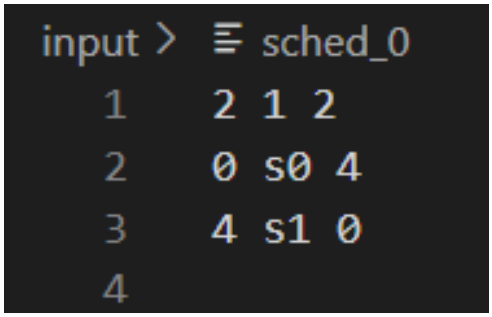
- Test sched_0:

Running the first test, `sched_0`:

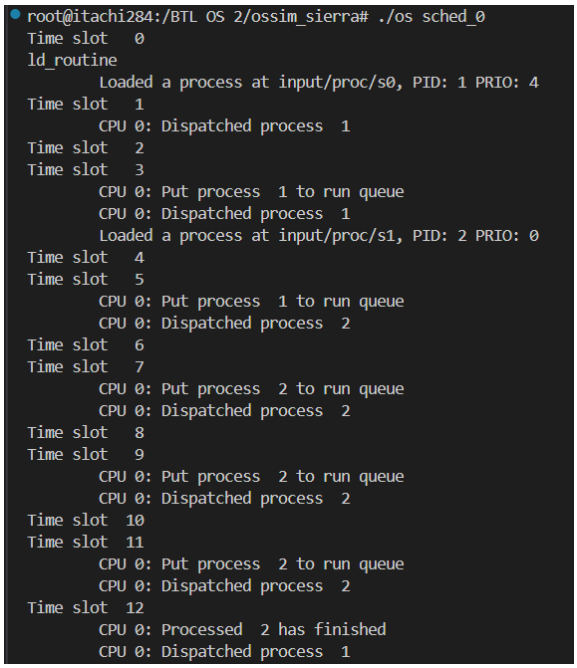
Time slice = 2, Number of CPUs = 1, Number of processes = 2.

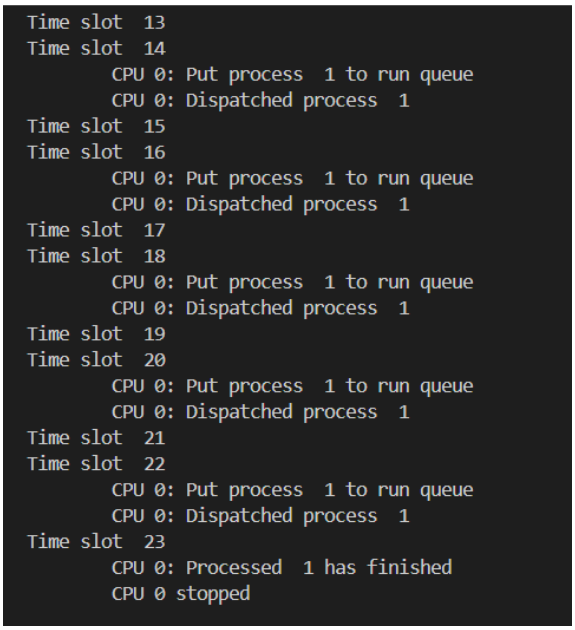
First process: arrives at time 0, path `s0`, priority 4.

Second process: arrives at time 4, path `s1`, priority 0.



We get the output of the test:





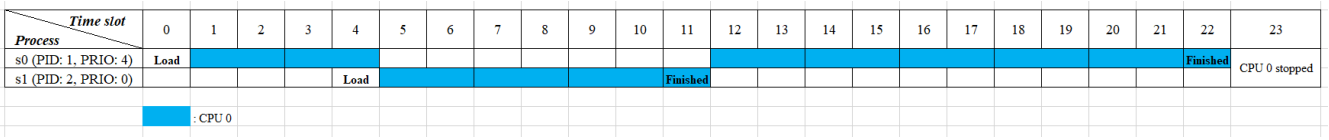


Figure 2.2: Gantt diagram of how processes are executed by the CPU in test `sched_0`

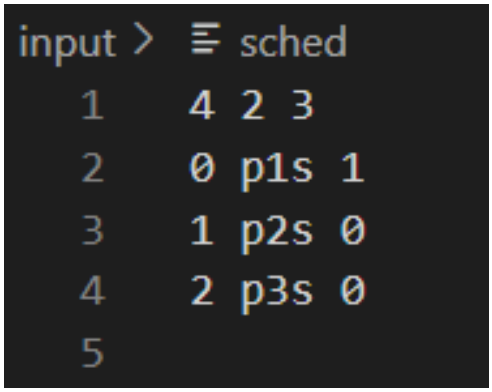
- Test sched:

Running the second test, `sched`: Time slice = 4, Number of CPUs = 2, Number of processes = 3.

First process: arrives at time 0, path `p1s`, priority 1.

Second process: arrives at time 1, path `p2s`, priority 0.

Third process: arrives at time 2, path `p3s`, priority 0.



We get the output of the test:

```
root@itachi284:/BTL OS 2/ossim_sierra# ./os sched
Time slot   0
ld_routine
        Loaded a process at input/proc/p1s, PID: 1 PRIO: 1
Time slot   1
        CPU 0: Dispatched process  1
        Loaded a process at input/proc/p2s, PID: 2 PRIO: 0
Time slot   2
        CPU 1: Dispatched process  2
        Loaded a process at input/proc/p3s, PID: 3 PRIO: 0
Time slot   3
Time slot   4
Time slot   5
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  3
        CPU 1: Put process  2 to run queue
        CPU 1: Dispatched process  2
Time slot   6
Time slot   7
Time slot   8
Time slot   9
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  3
        CPU 1: Put process  2 to run queue
        CPU 1: Dispatched process  2
```

```
Time slot  10
Time slot  11
Time slot  12
Time slot  13
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  3
        CPU 1: Processed  2 has finished
        CPU 1: Dispatched process  1
Time slot  14
Time slot  15
Time slot  16
        CPU 0: Processed  3 has finished
        CPU 0 stopped
Time slot  17
Time slot  18
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  1
Time slot  19
Time slot  20
        CPU 1: Processed  1 has finished
        CPU 1 stopped
```

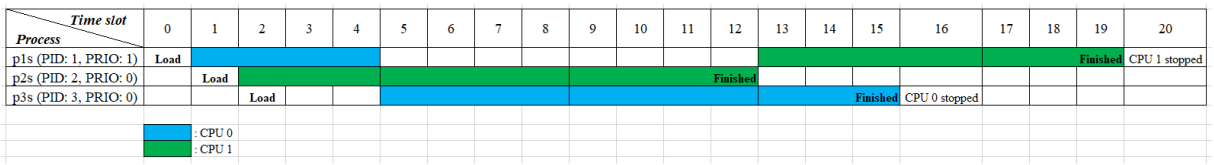


Figure 2.4: Gantt diagram of how processes are executed by the CPU in test `sched`

2.1.3 Question Answering

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

Answer: The scheduler implemented in this project is based on a Multi-Level Queue (MLQ) strategy with a time slot mechanism for each priority level. This approach introduces several benefits compared to the other traditional scheduling algorithms in operating systems.

- Priority-based and Fairness: Unlike normal Round Robin (RR) or First-Come-First-Served (FCFS) which treat all processes equally, MLQ clearly separates processes based on their assigned priority. Each priority has its own queue, and the system always considers higher-priority queues first when selecting which process to run. But different from simple priority-based schedulers that may starve the lower priority tasks, this implementation includes fixed time slots for each queue. This ensures that

even processes with low priority are not forever delayed, as they are still guaranteed CPU time once higher priority queues exhaust their slots.

- Responsive and Progress: Because higher priority queues are assigned more time slots per cycle (`slot = MAX_PRIO - prio`), the scheduler is highly responsive to important tasks. At the same time, the periodic slot reset mechanism guarantees that processes in lower queues will eventually be scheduled, ensuring steady progress across all workloads. This balance gives the MLQ approach an edge over algorithms like Shortest-Job-First (SJF) or Shortest-Remaining-Time-First (SRTF), which can favor short jobs so strongly that longer ones may be postponed forever.

- Predictable and Simple to manage: Each process is stored in the same queue based on its priority, and each queue is managed with a round-robin policy. This makes the scheduling decisions easier to trace and analyze compared to algorithms like Multilevel Feedback Queues (MLFQ), which dynamically adjust priorities during execution.

- Effective Resource Utilization: High priority processes can access the CPU quickly when needed and lower priority tasks can fill in idle CPU time when higher queues are empty or waiting.

2.2 Memory Management

2.2.1 Virtual Memory Regions

`get_free_vmrg_area` function plays a vital role in Sierra OS's memory allocation system, implementing a first-fit memory allocation strategy to find suitable free memory regions within a virtual memory area (VMA). This function is responsible for locating an available memory region of the requested size from a list of free memory regions, making it a critical component in the memory allocation subsystem of the operating system.

When a process requests memory via the `__alloc` function, this function first attempts to find an existing free region before resorting to expanding the process's memory footprint. The function begins by retrieving the virtual memory area struct and its associated list of free regions, then initializes the output parameter with invalid values. It then traverses the linked list of free regions, checking each one to determine if it meets three critical criteria:

- The region must be large enough to accommodate the requested size.

- It must not extend beyond the current break pointer (`sbrk`) of the VMA.
- It must start at or before the break pointer.

When a suitable region is found, the function configures the output parameter with the details of the allocated region (setting start and end addresses), updates the free list by advancing the starting pointer of the partially used free region, and returns a success code. If no appropriate region can be found after examining all available free regions, the function returns a failure code, triggering the memory allocation system to expand the process's memory area using `inc_vma_limit`. This efficient reuse of freed memory spaces is essential for minimizing memory fragmentation and avoiding unnecessary expansion of the process's memory footprint.

```
1  int get_free_vmrg_area(struct pcb_t *caller, int vmaid, int size,
       struct vm_rg_struct *newrg)
2  {
3    struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm,
       vmaid);
4    struct vm_rg_struct *rgit = cur_vma->vm_freerg_list;
5    if (rgit == NULL)
6      return -1;
7
8    /* Probe uninitialized newrg */
9    newrg->rg_start = newrg->rg_end = -1;
10
11   while (rgit != NULL) {
12     if ((rgit->rg_end - rgit->rg_start) >= size &&
13         rgit->rg_end <= cur_vma->sbrk &&
14         rgit->rg_start <= cur_vma->sbrk) {
15       // Found a suitable region
16       newrg->rg_start = rgit->rg_start;
17       newrg->rg_end = rgit->rg_start + size;
18       newrg->rg_next = NULL;
19       // Update the free region list
20       rgit->rg_start += size;
21       return 0; // Successfully found and allocated a region
22     }
23     rgit = rgit->rg_next;
24   }
```

```
25
26   return -1; // No suitable region found
27 }
```

Listing 2.1: Function `get_free_vmrg_area`

2.2.2 Virtual Memory Areas

`get_vm_area_node_at_brk` function is a fundamental part of the Sierra OS memory management system. It is responsible for dynamically expanding a process's virtual memory by appending a new memory region at the current break position, akin to the `sbrk()` mechanism in traditional operating systems.

This function operates by first retrieving the specified Virtual Memory Area (VMA) using the `get_vma_by_num` function. Upon successfully locating the VMA, it allocates a new `vm_rg_struct` to represent the extended memory region. The starting address of the new region is set to the current break pointer (`cur_vma->sbrk`), and the ending address is calculated by adding the aligned size (`alignedsz`) to the start. The VMA's break pointer is then updated to reflect the extension.

This new region is returned to the caller and subsequently used in conjunction with other functions such as `inc_vma_limit` (for expanding the virtual space) and `vm_map_ram` (for mapping to physical memory). If the function encounters an error—such as the VMA not existing or memory allocation failing—it returns `NULL`, signaling the failure.

```
1  struct vm_rg_struct *get_vm_area_node_at_brk(struct pcb_t *caller,
       int vmaid, int size, int alignedsz)
2  {
3    struct vm_rg_struct *newrg;
4    struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm,
      vmaid); // Retrieve the current VMA
5
6    if (cur_vma == NULL)
7      return NULL; // Invalid VMA
8
9    newrg = malloc(sizeof(struct vm_rg_struct));
10   if (newrg == NULL)
11     return NULL;
12
```

```
13   newrg->rg_start = cur_vma->sbrk;            // Start at the
     current break pointer
14   newrg->rg_end = cur_vma->sbrk + alignedsz; // Extend by the
     aligned size
15
16   // Update the VMA's break pointer
17   cur_vma->sbrk += alignedsz;
18
19   return newrg;
20 }
```

Listing 2.2: Function `get_vm_area_node_at_brk`

`validate_overlap_vm_area` function ensures memory integrity in Sierra OS by checking whether a proposed memory region overlaps with any existing regions in the process's address space. This is critical to prevent memory corruption or undefined behavior. The function iterates through each Virtual Memory Area (VMA) in the process's memory map and compares the new region's boundaries against those of existing areas. It uses a standard overlap-checking logic, where overlap exists if the end of the new region exceeds the start of an existing one and its start is before the end of that existing region. If such a conflict is found, the function returns `-1`; otherwise, it returns `0`, allowing memory allocation to proceed. This function is often called by higher-level allocation routines like `inc_vma_limit` to validate newly reserved regions.

```
1  int validate_overlap_vm_area(struct pcb_t *caller, int vmaid, int
       vmastart, int vmaend)
2  {
3    struct vm_area_struct *vma = caller->mm->mmap;
4
5    while (vma != NULL) {
6      // Check for overlap
7      if (!(vmaend <= vma->vm_start || vmastart >= vma->vm_end)) {
8        return -1; // Overlap detected
9      }
10     vma = vma->vm_next;
11   }
12
13   return 0;
14 }
```

Listing 2.3: Function `validate_overlap_vm_area`

`inc_vma_limit` function is responsible for safely extending a process's virtual memory in Sierra OS, mimicking the behavior of the `brk()` system call in Unix-like operating systems. It begins by aligning the requested increment size to the system's page size and calculating the number of pages required. It then calls `get_vm_area_node_at_brk` to reserve a new region at the current break point. Before committing this region, it ensures that it does not overlap with any existing regions by invoking `validate_overlap_vm_area`. If the region passes validation, the function updates the VMA's boundary and proceeds to map the new virtual memory region to physical memory using `vm_map_ram`. If any of these steps fail, the function returns `-1` to indicate failure; otherwise, it returns `0`, signifying successful memory extension.

```
1  int inc_vma_limit(struct pcb_t *caller, int vmaid, int inc_sz)
2  {
3    struct vm_rg_struct * newrg = malloc(sizeof(struct vm_rg_struct)
     );
4    int inc_amt = PAGING_PAGE_ALIGNSZ(inc_sz);
5    int incnumpage = inc_amt / PAGING_PAGESZ;
6    struct vm_rg_struct *area = get_vm_area_node_at_brk(caller,
     vmaid, inc_sz, inc_amt);
7    struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm,
     vmaid);
8
9    if (cur_vma == NULL || area == NULL)
10     return -1;
11
12   int old_end = cur_vma->vm_end;
13
14   // Validate overlap of obtained region
15   if (validate_overlap_vm_area(caller, vmaid, area->rg_start, area
     ->rg_end) < 0)
16     return -1; // Overlap detected
17
18   cur_vma->vm_end = area->rg_end; // Extend the VMA's end boundary
19
20   if (vm_map_ram(caller, area->rg_start, area->rg_end,
```

```
21                  old_end, incnumpage, newrg) < 0)
22     return -1; // Mapping failed
23
24   return 0;
25 }
```

Listing 2.4: Function `inc_vma_limit`

2.2.3 Memory Allocation/Deallocation

`liballoc` and `__alloc` functions together form the memory allocation subsystem in Sierra OS, implementing a two-layered design for managing virtual memory requests. The public-facing `liballoc` function handles thread synchronization and invokes the internal `__alloc` function, which contains the detailed allocation logic. This separation of concerns enables safe, concurrent memory management while providing flexibility in memory reuse and expansion.

When a process requires dynamic memory, it calls `liballoc` with the desired size and a register index. `liballoc` first locks a global mutex to prevent race conditions in multi-threaded environments. It then delegates the allocation request to `__alloc`, specifying the default virtual memory area ID as 0. Once the allocation completes, `liballoc` may print memory state information for debugging and then unlocks the mutex before returning.

The `__alloc` function attempts to reuse previously freed memory regions by calling `get_free_vmrg_area`. If a suitable free region is found, it updates the process's symbol region table and returns the start address. If not, the function calculates a page-aligned size, saves the current break pointer, and invokes system call 17 with the `SYSMEM_INC_OP` operation to request additional memory. It then updates the break pointer, enlists any excess memory as a new free region, updates the symbol table, and returns the starting address of the allocated memory. This design allows the system to efficiently manage memory fragmentation and dynamic growth.

```
1 int __alloc(struct pcb_t *caller, int vmaid, int rgid, int size,
     int *alloc_addr)
2 {
3   /* Allocate from available free list */
4   struct vm_rg_struct rgnode;
5
6   if (get_free_vmrg_area(caller, vmaid, size, &rgnode) == 0)
7   {
```

```
8       caller->mm->symrgtbl[rgid].rg_start = rgnode.rg_start;
9       caller->mm->symrgtbl[rgid].rg_end = rgnode.rg_end;
10
11      *alloc_addr = rgnode.rg_start;
12      return 0;
13    }
14
15    struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm,
       vmaid);
16    int inc_sz = PAGING_PAGE_ALIGNSZ(size);
17    int old_sbrk = cur_vma->sbrk;
18
19    /* SYSCALL 17: sys_memmap with SYSMEM_INC_OP */
20    struct sc_regs regs;
21    regs.a1 = (uint32_t)SYSMEM_INC_OP;
22    regs.a2 = (uint32_t)vmaid;
23    regs.a3 = (uint32_t)inc_sz;
24    pthread_mutex_lock(&mmvm_lock);
25    syscall(caller, 17, &regs);
26    pthread_mutex_unlock(&mmvm_lock);
27
28    cur_vma->sbrk = old_sbrk + inc_sz;
29
30    struct vm_rg_struct *rgit = malloc(sizeof(struct vm_rg_struct));
31    rgit->rg_start = old_sbrk + size;
32    rgit->rg_end = cur_vma->sbrk;
33    rgit->rg_next = NULL;
34    enlist_vm_freerg_list(caller->mm, rgit);
35
36    caller->mm->symrgtbl[rgid].rg_start = old_sbrk;
37    caller->mm->symrgtbl[rgid].rg_end = old_sbrk + size;
38    *alloc_addr = old_sbrk;
39
40    return 0;
41  }
```

Listing 2.5: Function `__alloc`

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

  /* Use default virtual memory area ID = 0 */
  int result = __alloc(proc, 0, reg_index, size, &addr);

#ifdef CHECKTEST
  printf("===== PHYSICAL MEMORY AFTER ALLOCATION =====\n");
  printf("PID=%d - Region=%d - Address=%08x - Size=%d byte\n",
   proc->pid, reg_index, addr, size);
#ifdef PAGETBL_DUMP
  print_pgtbl(proc, 0, -1); // Dump full page table
#endif
  printf("
   ================================================================\
   n");
#endif

  return result;
}
```

Listing 2.6: Function `liballoc`

`libfree` and `__free` functions form the deallocation subsystem in the Sierra OS simulator, working together to return previously allocated memory back to the system for future reuse. The `libfree` function acts as the public API exposed to user-level processes, while the internal `__free` function handles the actual memory bookkeeping. This structure follows the same layered design pattern used in memory allocation, where the higher-level interface ensures thread safety and invokes the lower-level logic to manipulate internal memory structures directly. This approach allows processes to safely release memory while the system ensures integrity and synchronization across concurrent threads.

When a process calls `libfree`, it passes in the register index corresponding to the region to be released. The function locks a global mutex to ensure exclusive access to memory management structures, then calls `__free` with virtual memory area ID 0 and the given region index. The `__free` function first validates the region ID, then copies the

region's start and end addresses from the symbol table, clears the entry by marking both fields as -1, and appends the region to the free list using `enlist_vm_freerg_list`. Once the memory is marked free, control returns to `libfree`, which optionally prints memory status for debugging purposes before releasing the mutex lock. This design efficiently supports memory recycling and prevents leaks by allowing memory to be reused in future allocations.

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

  struct vm_rg_struct* region = malloc(sizeof(struct vm_rg_struct)
   );
  *region = *(get_symrg_byid(caller->mm, rgid));
  region->rg_next = NULL;
  if (region == NULL) {
    free(region);
    return -1; // Invalid region index
  }

  // Clear the symbol table entry for the freed region
  caller->mm->symrgtbl[rgid].rg_start = -1;
  caller->mm->symrgtbl[rgid].rg_end = -1;
  return enlist_vm_freerg_list(caller->mm, region);
}

int libfree(struct pcb_t *proc, uint32_t reg_index)
{
  // By default using vmaid = 0
  int result = __free(proc, 0, reg_index);
#ifdef CHECKTEST
  printf("===== PHYSICAL MEMORY AFTER DEALLOCATION =====\n");
  printf("PID=%d - Region=%d\n", proc->pid, reg_index);
#ifdef PAGETBL_DUMP
  print_pgtbl(proc, 0, -1); // print max TBL
#endif
  printf("
```

```
    ================================================================\
    n");
#endif
  return result;
}
```

Listing 2.7: __free and libfree

`alloc_pages_range` function in the Sierra OS simulator fulfills a crucial role in the physical memory allocation process by acquiring a sequence of physical memory frames for a process. This function serves as the intermediary between high-level memory allocation requests and the actual reservation of physical memory frames from the system's RAM. When a process needs to expand its memory footprint, this function is called to obtain a contiguous sequence of physical frames that can later be mapped to the process's virtual address space. The function accepts three parameters: the calling process, the number of pages required, and an output parameter to return the linked list of allocated frames.

The function implements its core functionality as a loop that iterates through the requested number of pages, attempting to allocate each one individually. For each iteration, it first allocates a new frame structure, then attempts to obtain a free physical frame using the `MEMPHY_get_freefp` function which pulls an available frame from the physical memory's free list. If successful, the function records the frame page number (FPN) in the newly created structure and carefully builds a linked list of these structures, handling both the initial case (where the list is empty) and subsequent cases by maintaining a pointer to the last element. This approach allows the function to construct a complete list of allocated frames that can be passed to mapping functions like `vmap_page_range`. If at any point the system runs out of free frames, the function returns a specific error code (-3000) to indicate memory exhaustion, which is later interpreted by higher-level functions to implement appropriate error handling. This staged allocation strategy enables the memory management subsystem to efficiently handle varying allocation sizes while maintaining a clear chain of responsibility between different components of the memory management system.

```
int alloc_pages_range(struct pcb_t *caller, int req_pgnum, struct
    framephy_struct **frm_lst)
{
  int pgit, fpn;
  struct framephy_struct *newfp = NULL;
  struct framephy_struct *last = NULL;
```

```
6
7   for (pgit = 0; pgit < req_pgnum; pgit++)
8   {
9     newfp = malloc(sizeof(struct framephy_struct));
10    if (!newfp) return -1;
11
12    if (MEMPHY_get_freefp(caller->mram, &fpn) == 0)
13    {
14      newfp->fpn = fpn;
15      newfp->fp_next = NULL;
16
17      if (*frm_lst == NULL)
18        *frm_lst = newfp;
19      else
20        last->fp_next = newfp;
21
22      last = newfp;
23    }
24    else
25    {
26      free(newfp);
27      return -3000;
28    }
29  }
30
31  return 0;
32 }
```

Listing 2.8: alloc_pages_range Function

`vmap_page_range` function is a critical component of Sierra OS's memory management subsystem that establishes the mapping between virtual memory pages and physical memory frames. This function is responsible for updating the page table entries to reflect the association between a process's virtual address space and actual physical memory locations. When a process allocates memory, this function creates the necessary page table entries that the CPU will later use to translate memory accesses from virtual to physical addresses. The function takes five parameters: the calling process, the starting virtual address (which must be page-aligned), the number of pages to map, a linked list of physical

frames that will be mapped to these pages, and an output parameter to return information about the mapped region.

The function begins by calculating the page number from the starting address using the `PAGING_PGN` macro and initializes the return region structure with the appropriate start and end addresses. The core functionality is implemented in a loop that iterates through each page in reverse order, establishing mappings between virtual pages and physical frames. For each page, the function calls `pte_set_fpn` to update the corresponding page table entry in the process's page directory (`pgd`), effectively creating the virtual-to-physical mapping. Additionally, it calls `enlist_pgn_node` to add each newly mapped page to the process's FIFO page list, which is used for page replacement when memory becomes scarce—a clear indication that Sierra OS uses a simple FIFO algorithm for page replacement. This comprehensive approach to memory mapping ensures that virtual memory addresses used by processes are properly translated to physical memory locations, providing the foundation for address space isolation between processes while maintaining shared access to physical memory resources.

```
1  int vmap_page_range(struct pcb_t *caller,           // process
       call
2                      int addr,                       // start
       address which is aligned to pagesz
3                      int pgnum,                      // num of
       mapping page
4                      struct framephy_struct *frames, // list of the
        mapped frames
5                      struct vm_rg_struct *ret_rg)    // return
       mapped region, the real mapped fp
6  {                                                   // no
       guarantee all given pages are mapped
7    int pgit = 0;
8    int pgn = PAGING_PGN(addr);
9    struct framephy_struct *fpit = frames;
10   ret_rg->rg_start = addr;
11   ret_rg->rg_end = addr + pgnum * PAGING_PAGESZ;
12
13   for (pgit = pgnum - 1; pgit >= 0; pgit--) {
14     if (!fpit) break;
15     pte_set_fpn(&caller->mm->pgd[pgn + pgit], fpit->fpn);
16     enlist_pgn_node(&caller->mm->fifo_pgn, pgn + pgit);
```

```
17      fpit = fpit->fp_next;
18    }
19
20    return 0;
21  }
```

Listing 2.9: vmap_page_range Function

2.2.4 Memory Device Operations

`MEMPHY_dump` serves as a diagnostic tool in the Sierra OS simulator's physical memory management subsystem, providing visibility into the actual content of physical memory for debugging and educational purposes. This function traverses the entire storage array of a physical memory structure and prints out the values of non-zero bytes, offering a selective view of memory that focuses on meaningful data while filtering out empty spaces. It accepts a single parameter: a pointer to a `memphy_struct` structure that represents either the RAM or swap space being examined.

The function begins with basic validation, checking that the provided memory structure and its storage array exist, and returns an error code if either is null. It then iterates through each byte in the memory's storage array, from index 0 to the maximum size of the memory (`mp->maxsz`), employing a space optimization strategy by only displaying bytes that have non-zero values—a practical approach since most memory locations in real systems are typically unused.

For each non-zero byte found, it prints the byte's address in hexadecimal format and its value in decimal, providing a comprehensive view of the memory's actual contents. This function is particularly valuable during debugging sessions and for educational demonstrations of memory management operations, as it allows developers to directly observe how data is stored in physical memory and verify the correct operation of memory allocation, deallocation, and paging operations in the simulator.

```
1  int MEMPHY_dump(struct memphy_struct *mp)
2  {
3      printf("===== PHYSICAL MEMORY DUMP =====\n");
4      if (!mp || !mp->storage) return -1;
5
6      for (int i = 0; i < mp->maxsz; i++) {
7          if (mp->storage[i] != 0) {
8              // Print byte in expected format
```

```
            printf("BYTE %08x: %d\n", i, mp->storage[i]);
        }
    }

    printf("===== PHYSICAL MEMORY END-DUMP =====\n");
    return 0;
}
```

Listing 2.10: MEMPHY_dump Function

2.2.5 Page Table Operations

The `init_mm` function initializes the memory management structure (`mm`) for a new process. It allocates a page table directory (`pgd`) and creates the first virtual memory area (`vma0`), which represents the initial empty memory space for the process. The function sets the start and end of the virtual memory area, initializes a free region list with a single region, and connects the VMA to the memory management structure. This function is fundamental for setting up the basic virtual memory environment that allows a process to manage dynamic memory allocations and future expansions. Without this initialization, processes would not have a valid memory map to operate on.

```
int init_mm(struct mm_struct *mm, struct pcb_t *caller)
{
    struct vm_area_struct *vma0 = malloc(sizeof(struct
    vm_area_struct));

    mm->pgd = malloc(PAGING_MAX_PGN * sizeof(uint32_t));

    /* By default the owner comes with at least one VMA */
    vma0->vm_id = 0;
    vma0->vm_start = 0;
    vma0->vm_end = vma0->vm_start;
    vma0->sbrk = vma0->vm_start;
    struct vm_rg_struct *first_rg = init_vm_rg(vma0->vm_start,
    vma0->vm_end);
    enlist_vm_rg_node(&vma0->vm_freerg_list, first_rg);

    vma0->vm_next = NULL;       // No next VMA yet
    vma0->vm_mm = mm;           // Set owner of VMA
```

```
17
18      mm->mmap = vma0;            // Set this VMA as the mmap start
19      return 0;
20  }
```

Listing 2.11: Function `init_mm`

2.2.6 Status of Memory Allocation in Heap and Data Segments

The system utilizes a **paging-based memory management scheme** to translate virtual addresses into physical addresses. Each process is assigned a *page table*, which maps *virtual page numbers (VPNs)* to *physical frame numbers (PFNs)*. When a process accesses memory, its virtual address is split into two parts: the page number and the offset. The system uses the page number to look up the corresponding entry in the page table.

The address translation mechanism is primarily implemented through the function `pg_getpage()` in `libmem.c`, which checks if the page is present in memory. If the page is valid and mapped, `pte_set_fpn()` from `mm.c` retrieves the physical frame number, and the memory access proceeds. If the page is not present, a page fault occurs, triggering a swap-in operation via `SYSCALL 17` with the `SYSMEM_SWP_OP` flag. In such cases, the system uses `pte_set_swap()` to update the page table entry with the swap frame information.

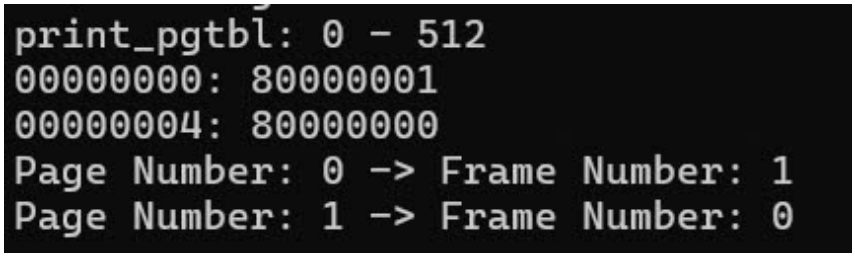


Figure 2.5: Example ouput from os_mlq_1_paging

These lines indicate that virtual pages 0 and 1 are mapped to physical frames 1 and 0, respectively. The hexadecimal values (e.g., `80000001`) represent the raw page table entries, where the most significant bit indicates that the page is present in memory.

Memory operations such as reading and writing to virtual addresses are handled by `pg_getval()` and `pg_setval()` respectively, which internally invoke `pg_getpage()` to perform address translation. For example, when writing to a memory region:

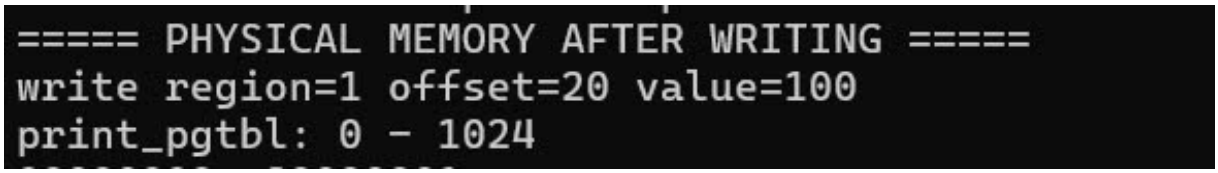


Figure 2.6: Example ouput from os_mlq_1_paging

The system translates the virtual address (*base + offset*) into a physical address (e.g., `0x114`) and writes the value `100` at that location in RAM.

If a page is evicted due to limited memory, a value like `c0000000` may appear in the page table, indicating that the page has been swapped out. In this case, when the page is accessed again, it will be swapped back in using the `pg_getpage()` function and updated in the page table accordingly.

The memory allocation for each process in the system is divided into *data segments* and *heap segments*, both of which are managed through virtual memory areas (VMAs) and page mappings. This section describes how each segment type is allocated and tracked during simulation.

1. Data Segment Allocation (Region 0)

The *data segment* (Region 0) is allocated when a process is first loaded into memory. This region is created by the `init_mm()` function, which initializes a virtual memory area (VMA) for static data. It is typically fixed in size and located at the beginning of the virtual address space (starting at address `0x00000000`).

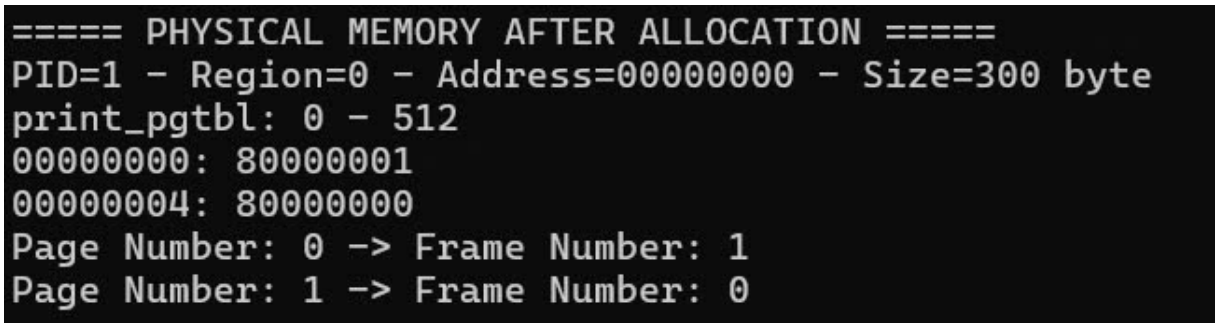


Figure 2.7: Example ouput from os_mlq_1_paging

This shows that process `PID=1` was assigned a 300-byte data segment, mapped across two pages (Page 0 and Page 1), corresponding to frames 1 and 0 in physical memory.

2. Heap Segment Allocation (Region $\geq$ 1)

The *heap segment* is dynamically allocated during process execution when a process invokes `ALLOC`. This triggers the `liballoc()` $\rightarrow$ `__alloc()` function sequence, which either:

- Finds free space within the existing heap using `get_free_vmrg_area()`, or
- Extends the heap using `inc_vma_limit()` and maps new pages to RAM using `vm_map_ram()`.

```
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=1 - Region=4 - Address=00000200 - Size=300 byte
print_pgtbl: 0 - 1024
Time slot   4
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
```

Figure 2.8: Example ouput from os_mlq_1_paging

This allocation extends `PID 1`'s heap by 300 bytes starting at `0x00000200`, backed by frames 3 and 2. Subsequent calls to `ALLOC` (e.g., Region 1, 2, or 3) follow similar logic.

Heap segments may also be deallocated with `FREE`, resulting in:

```
===== PHYSICAL MEMORY AFTER DEALLOCATION =====
PID=3 - Region=2
print_pgtbl: 0 - 512
00000000: 80000005
00000004: 80000004
Page Number: 0 -> Frame Number: 5
Page Number: 1 -> Frame Number: 4
```

Figure 2.9: Example ouput from os_mlq_1_paging

The `libfree()` and `__free()` functions remove the allocated region and return it to the free region list, making it available for reuse.

3. Summary of Memory Allocation

PID	Region	Segment Type	Address Start	Size	Mapped Frames
1	0	Data	0x00000000	300 byte	Frame 1,0
1	4	Heap	0x00000200	300 byte	Frame 1, 0, 2, 3
3	0	Data	0x00000000	300 byte	Frame 5, 4
3	1	Heap	0x0000012c	100 byte	Frame 3, 4
1	1	Heap	0x00000000	100 byte	Frame 1, 0, 2, 3
5	0	Data	0x00000000	300 byte	Frame 7, 6
3	2	Heap	0x00000000	100 byte	Frame 5, 4
5	1	Heap	0x0000012c	100 byte	Frame 7, 6
5	2	Heap	0x00000000	100 byte	Frame 7, 6

This table captures the allocation state of memory across multiple processes during the simulation of os_mlq_1_paging, clearly showing both static and dynamic usage of memory, page-to-frame mapping, and regions that were later deallocated.

2.2.7 Question Answering

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

The Sierra OS simulator adopts a memory management design based on multiple memory segments or areas. Each segment, represented by the `vm_area_struct` in the source code, serves distinct purposes such as managing the code, data, stack, or heap areas. This segmentation approach introduces several important advantages:

- **Logical Organization and Modularity:** Each segment corresponds to a logical unit of memory (e.g., code, data, stack), leading to better organization. This modular approach allows independent management of each segment, improving maintainability and clarity in memory operations.

- **Improved Resource Allocation:** Segments can be allocated memory according to their specific needs. For instance, the heap can grow dynamically, while code regions remain fixed. This targeted allocation helps improve memory utilization and system efficiency. The use of `vm_freerg_list` within each segment facilitates localized memory reuse and reduces fragmentation.

- **Enhanced Security and Isolation:** By dividing memory into segments, the system can isolate different components and enforce access restrictions. For example, the code segment can be marked read-only, preventing modification at runtime, thereby increasing protection against malicious behavior or unintended changes.

- **Support for Memory Protection:** Each segment can have customized access permissions (read-only, read-write, or execute). This fine-grained access control enhances system reliability by preventing illegal operations on memory regions, such as writing to executable code or accessing restricted data.

- **Dynamic Memory Growth:** The use of segment-specific break pointers (`sbrk`) and functions like `inc_vma_limit` allow memory segments to grow dynamically. This is particularly useful for the heap and stack, where memory requirements can vary at runtime.

- **Simplified Memory Management:** Segmentation simplifies memory allocation and deallocation by localizing these operations within individual segments. This reduces the overall complexity of memory management algorithms and improves the robustness and reliability of the system.

In summary, the design of multiple memory segments in the Sierra OS simulator promotes efficient, secure, and maintainable memory management, which is essential even in educational or simplified operating system models. In a paging-based memory management system, the address translation process typically involves one or more levels of page tables. Increasing the number of levels beyond two—known as multi-level paging—offers both significant advantages and notable drawbacks. This technique is widely adopted in modern operating systems to efficiently manage large and sparsely populated virtual address spaces. Below is a detailed analysis of the implications of implementing multi-level paging, with relevance to the Sierra OS simulator.

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

Advantages of Multi-Level Paging

- **Memory Efficiency for Page Tables:** Multi-level paging reduces the memory overhead associated with maintaining large page tables. Only the page tables required for active address spaces are allocated, which is particularly beneficial for processes with sparse memory usage. In the context of the Sierra OS, this approach would eliminate the need for wasteful static allocations observed in `mm.c`.
- **Support for Larger Address Spaces:** Additional paging levels allow the system to manage significantly larger virtual address spaces. For instance, while 32-bit systems may only require two levels, 64-bit systems often use three or four levels to support terabytes of addressable memory. Implementing multi-level paging in Sierra OS would enable it to simulate more realistic and scalable memory architectures.
- **Enhanced Memory Protection:** By distributing protection attributes across multiple page table levels, multi-level paging enables more refined control over memory access. This supports stronger isolation between memory segments, thereby enhancing Sierra's capabilities for simulating advanced protection mechanisms.
- **Improved Representation of Process Memory Layouts:** Multi-level paging naturally aligns with the hierarchical nature of process memory layouts. It supports more efficient memory sharing and isolation between processes. For Sierra OS, this would result in a more realistic and flexible implementation of `vm_area_struct`.
- **Support for Flexible Page Sizes:** Multi-level paging facilitates the use of multiple page sizes, such as huge pages, which are crucial for optimizing performance in real-world systems. This could be implemented in Sierra OS by extending its address translation logic in `mm.c`.

Disadvantages of Multi-Level Paging

- **Increased Memory Access Overhead:** Each additional paging level introduces extra memory accesses during address translation. This increases latency for every memory access, potentially impacting performance.
- **More Complex Page Fault Handling:** Handling page faults becomes more involved, as it requires traversing multiple levels to allocate the necessary page tables.

In Sierra OS, this would necessitate significant restructuring of the page fault handler in `mm.c`.

- **TLB Performance Penalties:** Multi-level paging can increase Translation Lookaside Buffer (TLB) misses due to the deeper address translation hierarchy. Each TLB miss incurs the full cost of a multi-level page table walk, which can degrade system performance if not mitigated by efficient hardware support.
- **Higher Implementation Complexity:** Introducing multi-level paging increases the complexity of address translation logic. For Sierra OS, both `mem.c` and `mm.c` would require major modifications, making debugging and maintenance more challenging.
- **Overhead in Memory Allocation and Deallocation:** The process of creating and destroying page tables becomes more complex with additional levels, increasing the overhead associated with memory management. This would complicate process creation and termination routines in Sierra OS.

Question: What are the advantages and disadvantages of segmentation with paging?
Answer:
If a page is evicted, its PTE may display a value such as `c0000000`, indicating it has been swapped out.

Advantages:

- *Efficient Memory Utilization*: Paging eliminates external fragmentation, and segmentation logically organizes memory.
- *Logical Structure*: Segments represent meaningful parts of a program (code, data, heap).
- *Protection*: Different permissions can be enforced for each segment.
- *Flexible Management*: Segments like the heap can grow independently via mechanisms like `sbrk()`.

Disadvantages:

- *Increased Complexity*: Translation requires segment lookup followed by page lookup.
- *Higher Overhead*: Both segment tables and page tables must be maintained.

- *Hardware Dependency*: Full support requires complex MMU hardware or software emulation.
- *Internal Fragmentation*: Fixed-size pages may result in wasted space.

The memory allocation for each process in the system is divided into *data segments* and *heap segments*, both of which are managed through virtual memory areas (VMAs) and page mappings. This section describes how each segment type is allocated and tracked during simulation.

2.3 System Call

2.3.1 Overview

System Call Mechanism in Operating Systems:

- Userapplications invoke system calls through wrapper functions provided in libraries (commonly **libc**).
- The wrapper function places arguments into appropriate CPU registers.
- Acontext switch (via interrupt or trap) transfers control to kernel mode.
- The kernel invokes the appropriate system call handler via the system call table.
- The handler executes the requested operation with kernel privileges.
- Control is returned to user mode to continue execution of user-level tasks.

This architecture provides a clear separation between user space and kernel space, ensuring system security while also offering essential services to applications.

2.3.2 System calls in assignment

System Call Table

Number	Name	Entry Point
0	listsyscall	sys_listsyscall
17	menmmap	sys_menmmap
101	killall	sys_killall

Listsyscall

```c
#include "syscall.h"

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;
}
```

Listing 2.12: Implementation of listsyscall system call

Syntax

SYSCALL 0

Description

Lists all the system calls in the system call table

Memmap

```c
#include "syscall.h"
#include "libmem.h"
#include "mm.h"

//typedef char BYTE;

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;
```

```
18      case SYSMEM_SWP_OP:
19              __mm_swap_page(caller, regs->a2, regs->a3);
20              break;
21      case SYSMEM_IO_READ:
22              MEMPHY_read(caller->mram, regs->a2, &value);
23              regs->a3 = value;
24              break;
25      case SYSMEM_IO_WRITE:
26              MEMPHY_write(caller->mram, regs->a2, regs->a3);
27              break;
28      default:
29              printf("Memop code: %d\n", memop);
30              break;
31      }
32
33      return 0;
34  }
```

Listing 2.13: Implementation of memmap system call

Syntax

```
SYSCALL 17 SYSMEM_OP REG_ARG2 REG_ARG3
```

Description

`SYSCALL 17`: This invokes syscall number 17, which deals with memory operations. `SYSMEM_OP`: The specific memory operation to perform. It could be one of the following:

- **SYSMEM_MAP_OP:** Related to memory mapping with a dummy handler.
- **SYSMEM_INC_OP:** Expands the memory region with the handler **inc_vma_limit()**. This operation increases the limit of the Virtual Memory Area (VMA).
- **SYSMEM_SWP_OP:** Swaps memory pages with the handler **__mm_swap_page()**. This operation moves a memory page to a different physical frame.
- **SYSMEM_IO_READ:** Reads from physical memory with the handler **MEMPHY_read()**. This operation reads data from the physical memory (MEMPHY).
- **SYSMEM_IO_WRITE:** Writes to physical memory with the handler **MEMPHY_write()**. This operation writes data to the physical memory (MEMPHY).

`REG_ARG2, REG_ARG3`: These are two arguments for the syscall, depending on the type of memory operation.

Killall

Syntax

`SYSCALL 101 REGION ID`

Description

The **__sys_killall** function is responsible for locating and terminating all processes that are either currently running or in the ready queue, whose names match the name provided from user-space memory.

Input: A pointer to the calling process (caller), and the system register set (regs).

Output: The number of processes that were terminated.

Execution Steps

1. Read process name from user memory:

```
uint32_t memrg = regs->a1;

   /* TODO: Get name of the target proc */
   //proc_name = libread..
   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++;
   }
```

- **memrg** stores the starting memory address (provided by the a1 register) where the process name is stored.
- Aloop uses **libread** to read each byte from user memory.
- The process name is built character by character and stored in **proc_name**.

- If `data == -1`, it indicates the end of the name (null terminator), so the string is completed

2. Concatenating Strings to Form Full Path:

```
strcat(proc_path, "input/proc/");
strcat(proc_path, proc_name);
```

- The base path `"input/proc/"` is appended to the beginning of **my_proc_name**.
- Then, the actual process name read from memory is appended, resulting in the full path to the process.

3. Traversing the Running Process List and Terminating Matching Names:

```
if (running_list != NULL) {
    pthread_mutex_lock(&queue_lock);
    for (int i = 0; i < running_list->size; i++) {
        proc = running_list->proc[i];
        if (strcmp(proc->path, proc_path) == 0) {
            printf("Terminating process %d with name %s from
   running list\n",
            proc->pid, proc_name);
            free(proc);
        for (int j = i; j < running_list->size - 1; j++) {
            running_list->proc[j] = running_list->proc[j + 1];
            }
        running_list->size--;
        i--;
    }
    }
pthread_mutex_unlock(&queue_lock);
}
```

- If `running_list` exists, lock `queue_lock` and scan each process.
- If a process's path matches `proc_path`, print a termination message, free the process, shift the array to remove the gap, decrease size, and decrement `i` to recheck the new process at the current position.

- Finally, unlock `queue_lock`.

4. Traversing the Ready Queue by Priority Level (MLQ- Multi-Level Queue):

```
for (int i = 0; i < MAX_PRIO; i++) {
    struct queue_t *queue = &mlq_ready_queue[i];
    for (int j = 0; j < queue->size; j++) {
        proc = queue->proc[j];
        if (strcmp(proc->path, proc_path) == 0) {
            printf("Terminating process %d with name %s from ready
    queue\n",
            proc->pid, proc_name);
            free(proc);
        for (int k = j; k < queue->size - 1; k++) {
            queue->proc[k] = queue->proc[k + 1];
            }
            queue->size--;
            j--;
        }
    }
}
```

- Traverse through each priority queue (`mlq_ready_queue[i]`).
- For each queue, check each process.
- If the process's path matches `proc_path`, print the termination message, free the process, shift the subsequent processes to fill the gap, decrease the queue's size, and decrement `j-` to check the new process in position.

2.3.3 Testing

Test sys_killall

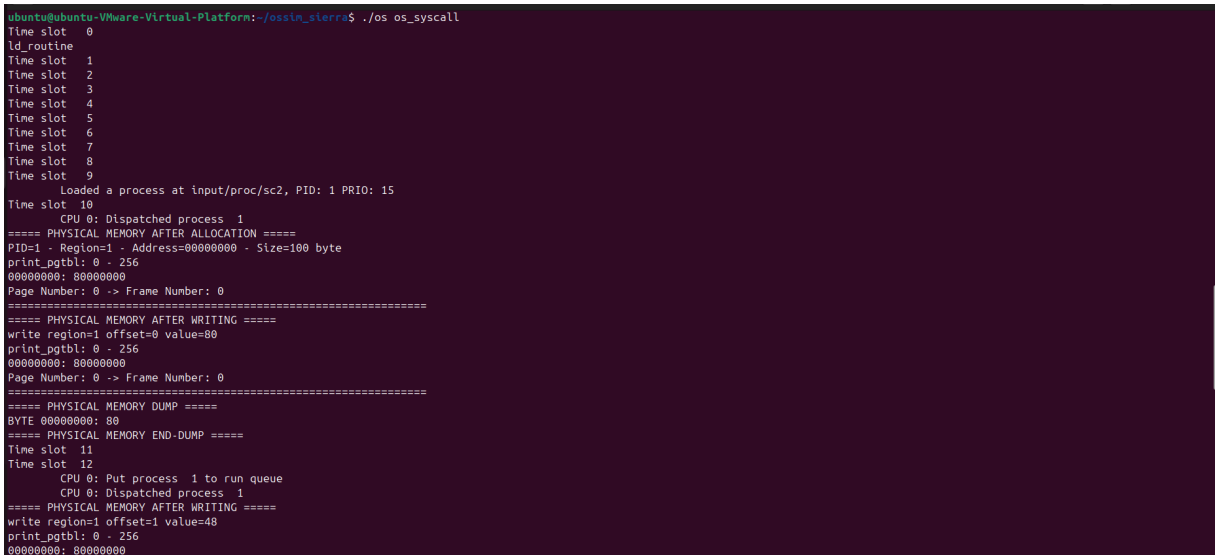
The test case is retrieved from the **"os_syscall"** text file, which has the following contents:

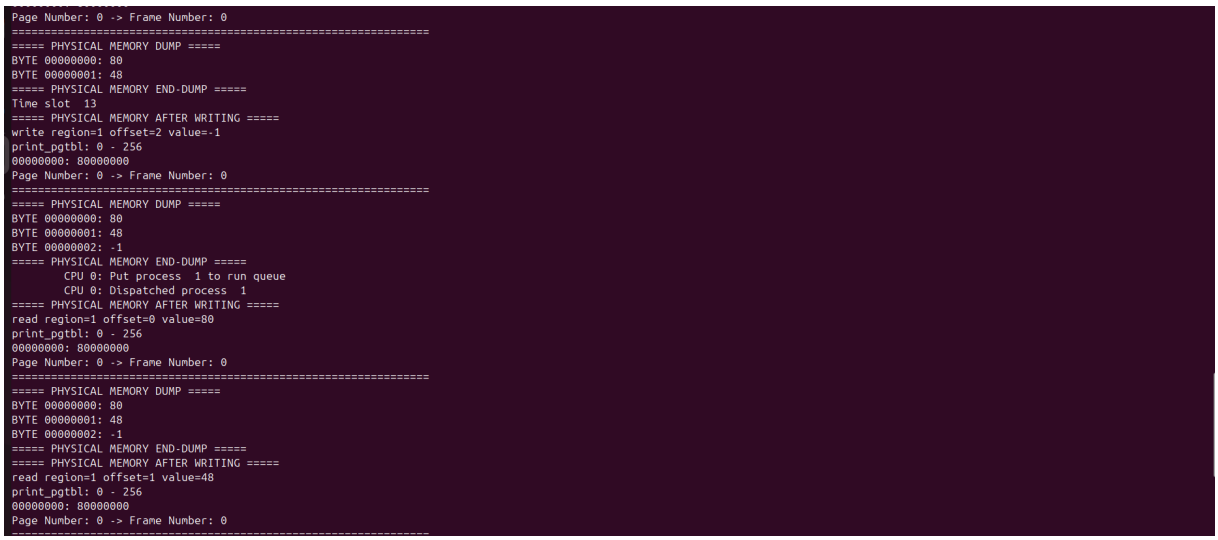
```
2 1 1
2048 16777216 0 0 0
9 sc2  15
```

process sc2 has contents:

```
20 5
alloc 100 1
write 80 1 0
write 48 1 1
write -1 1 2
syscall 101 1
```

Output: The output for this test case is shownbelow, representing the time slots and the sequence of events during process execution:





Explanation:

- Time slot 0-8: Various routines are executed, including loading and preparing processes.
- Time slot 9: The process is loaded with a process ID (PID) of 1 and priority 15.
- Time slot 10 - 13: The system allocates memory for the process and displays the physical memory state. The process performs several write operations, updating the memory and displaying the physical memory state after each write.
- Time slot 13: At the end of the time slot, syscall 101 is called. The system wants to kill the process with ID 1.
- Time slot 14-15: The process finishes execution, and CPU 0 stopped.

Test sys_listsyscall

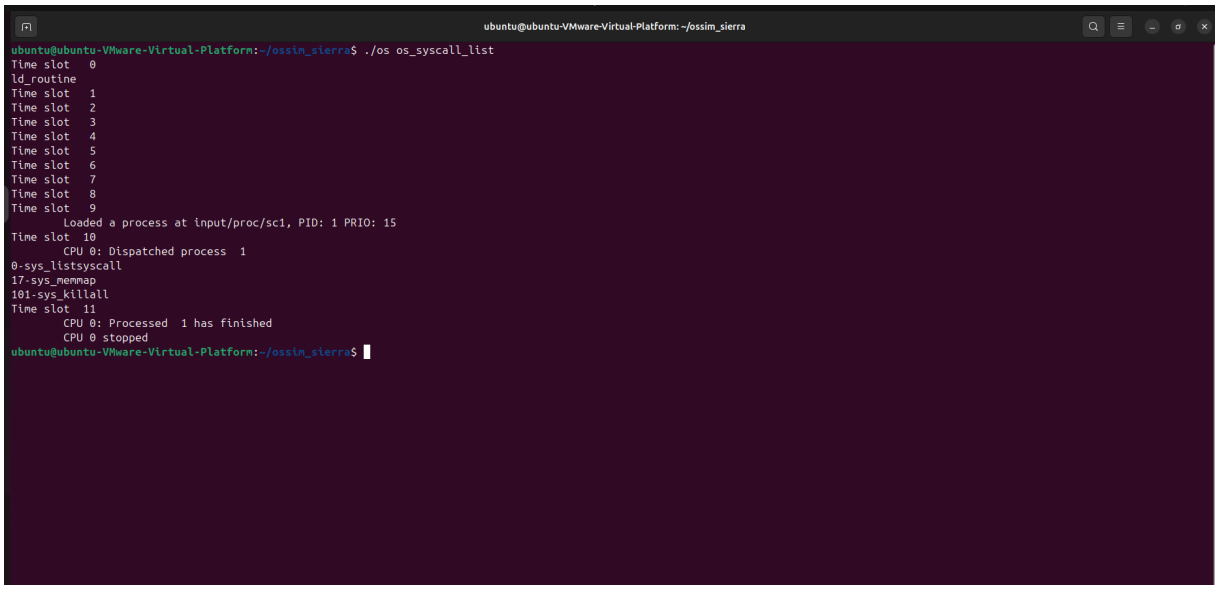
The test case is retrieved from the **"os_syscall_list"** text file,which has the following contents:

```
2 1 1
2048 16777216 0 0 0
9 sc1 15
```

process sc2 has contents:

```
20 1
syscall 0
```

Output: The output for this test case is shown below, representing the time slots and the sequence of events during process execution:



Explanation:

- Time slot 0-8: Various routines are executed, including loading and preparing processes.
- Time slot 9: A process is loaded with a process ID (PID) of 1 and priority 15.
- Time slot 10: The system invokes the following syscalls:
  - `0 - sys_list syscall` – A system call to list available syscalls.
  - `17 - sys_memmap` – A memory mapping syscall.
  - `101 - sys_killall` – The `sys_killall` syscall, which terminates matching processes.
- Time slot 11: The process has finished execution, and `CPU0` is stopped.

Test with System Call Not in System Call List

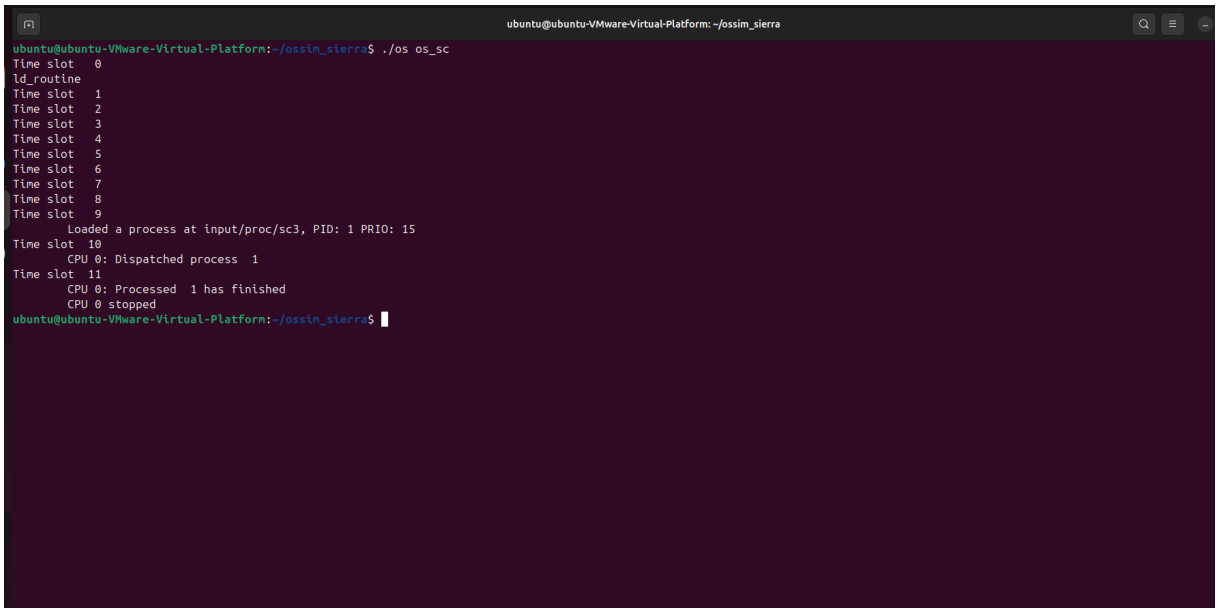
The test case is retrieved from the **"os_sc"** text file,which has the following contents:

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

process sc3 has contents:

```
20 1
syscall 440 1
```

Output: The output for this test case is shown below, representing the time slots and the sequence of events during process execution:



Explanation:

- Time slot 0-8: Various routines are executed, including loading and preparing processes.
- Time slot 9 - 10: The process with PID 1 and priority 15 is loaded into memory and dispatched for execution.
- Time slot 10 - 11: The system tries to execute the `syscall 440`. However, since this syscall is not defined in the system call list, no action is taken, and the process finishes execution. The process finishes execution, and `CPU 0` stops. The process finishes execution, and `CPU 0` stops.

Conclusion Since `syscall 440` is not part of the defined system calls, the system does nothing and simply proceeds with the termination of the process.

2.3.4 Question Answering

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

Answer:

Modern operating systems are built around the idea that while CPUs provide a small number of registers for passing arguments, these registers are not enough to convey large or complex data structures. To overcome this limitation, the common mechanism is to use **pointers**. Here is how it works:

• **Indirect Argument Passing:** Instead of passing all components of a complex argument in registers, the calling process allocates a memory block (or uses a struc ture) in its address space. All the details required by the system call are stored in this memory region. This is called *Indirect Argument Passing*.

For example, the system call interface uses registers like `REG_ARG2` and `REG_ARG3`(as seen in the memmap system call) to store pointers to user-space memory regions.

```
SYSCALL 17 SYSMEMOP REG_ARG2 REG_ARG3
```

This instruction passes the address of a memory region and an operation code via registers. The kernel dereferences these pointers to access the actual data (e.g., operation parameters or buffers) stored in the process's virtual memory.

• **Storing Data in User-Space Memory:** The user process stores large data (e.g., the string **"test"** for killall) in its heap (dynamic memory) or stack (local vari ables). The process retrieves the virtual address of the data (e.g., `0x1234`) and loads it into a register (e.g., `REG_ARG1`). The process then executes a system call (e.g., `SYSCALL 101` for `killall`), passing the register holding the address:

```
REG_ARG1 = 0x1234
```

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

Answer:

System calls are designed to be efficient interfaces between user applications and the OS kernel. However, if a particular system call's implementation takes an excessively long time to execute, several issues can arise: 40

- **CPU Starvation**: If the system call does not return promptly, it can monopolize CPU resources, especially in a uniprocessor system, by holding onto the processor for a long duration. In a multiprocessor or multitasking environment, this may delay other processes' execution, leading to system-wide performance degradation. This is called CPU Starvation.

  `- Example: The MLQ (Multi-Level Queue) scheduler cannot preempt a syscall executing in kernel mode. A memmap syscall scanning the entire RAM blocks all other processes until completion.`

- **Deadlocks and Resource Blocking**: In extreme cases, if the long-running system call also involves shared resources (e.g., locks, I/O resources, or memory buffers), other processes may also become blocked waiting for these resources. This can create a cascade effect, potentially leading to deadlocks or overall instability of the OS.

  `- Example: Shared resources (e.g., free_fp_list in memphy_struct) may remain locked, causing deadlocks if other processes wait indefinitely.`

- **Scheduler Disruption**: Time slices (defined in the input file) are wasted, violating the MLQ policy's fairness. Processes in lower-priority queues starve, leading to unbalanced CPU allocation.

2.4 Put It All Together

Finally, we combine all the previous work to form a complete operating system. Since the OS runs on multiple processors, shared resources may be accessed concurrently by multiple processes. Therefore, we need to implement a locking mechanism to ensure everything works correctly.

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: As mentioned above, the OS runs on multiple processors. Therefore, if shared memory is read or modified at the same time (a race condition), it can lead to incorrect output.

For instance, when we did not implement a locking mechanism for the scheduler functions (get_mlq_proc(), put_mlq_proc(), and add_mlq_proc()), we created a test that forced the OS to run 20 processes on 7 CPUs. With this number of CPUs, race conditions are likely to occur. When we ran the test multiple times, race conditions occasionally occurred, as shown in Figures 2.10 and 2.11.

```
Time slot  20
        CPU 3: Processed 13 has finished
        CPU 3: Dispatched process 19
        CPU 2: Processed 15 has finished
        CPU 2 stopped
        CPU 5: Put process 16 to run queue
        CPU 5: Dispatched process 16
        CPU 4: Dispatched process 18
        CPU 0: Processed 15 has finished
        CPU 0 stopped
        CPU 6: Dispatched process 20
double free or corruption (out)
        CPU 1: Put process 17 to run queue
        CPU 1: Dispatched process 17
Aborted (core dumped)
```

Figure 2.10: Test result 1 of scheduler without lock mechanism implementation

The incorrect result shown in Figure 2.10 occurs when add_mlq_proc() or put_mlq_proc() is not protected by a locking mechanism. As a result, process 15 is enqueued twice, allowing both CPU 0 and CPU 2 to take the same process and finish it. This causes the process to run twice and results in a double free() on that process.

```
xiaolin@LAPTOP-1HK4QPQB:/mnt/c/Uni/242/Operating System/BTL/ossim_sierra$ ./os test
Time slot   0
ld_routine
        Loaded a process at input/proc/s1, PID: 1 PRIO: 0
        CPU 6: Dispatched process  1
        CPU 3: Dispatched process  1
Time slot   1
        CPU 5: Dispatched process  1
        CPU 2: Dispatched process  1
        CPU 1: Dispatched process  1
        CPU 0: Dispatched process  1
        CPU 4: Dispatched process  1
        Loaded a process at input/proc/s1, PID: 2 PRIO: 0
        CPU 6: Processed  1 has finished
        CPU 3: Processed  1 has finished
        CPU 2: Processed  1 has finished
        CPU 0: Processed  1 has finished
        CPU 3: Dispatched process -217333760
        CPU 5: Processed  1 has finished
        CPU 0: Dispatched process -217333760
        CPU 2: Dispatched process -217333760
        CPU 5: Dispatched process -217333760
        CPU 6: Dispatched process -217333760
        CPU 1: Processed  1 has finished
        CPU 4: Processed  1 has finished
Time slot   2
        CPU 4: Dispatched process -217333760
        CPU 1: Dispatched process -217333760
        Loaded a process at input/proc/s1, PID: 3 PRIO: 0
        CPU 6: Processed -217333760 has finished
free(): double free detected in tcache 2
        CPU 4: Processed -217333760 has finished
        CPU 3: Processed -217333760 has finished
free(): double free detected in tcache 2
        CPU 2: Processed -217333760 has finished
free(): double free detected in tcache 2
free(): double free detected in tcache 2
Time slot   3
        CPU 5: Processed -217333760 has finished
free(): double free detected in tcache 2
Aborted (core dumped)
```

Figure 2.11: Test result 2 of scheduler without lock mechanism implementation

The incorrect result shown in Figure 2.11 occurs when get_mlq_proc() is not protected by a locking mechanism. As a result, two or more CPUs might attempt to access the same process in mlq_ready_queue[i] simultaneously. In this case, both may call dequeue(mlq_ready_queue[i]). However, if the queue contains only one process, one CPU will successfully receive the correct process for dispatch, while the others will receive a NULL value.

To avoid these problems, we implement a locking mechanism in the three scheduler functions: get_mlq_proc(), put_mlq_proc(), and add_mlq_proc(), as shown below:

```
1 struct pcb_t * get_mlq_proc(void) {
2   struct pcb_t * proc = NULL;
3   /*TODO: get a process from PRIORITY [ready_queue].
```

```
    * Remember to use lock to protect the queue.
    * */
  pthread_mutex_lock(&queue_lock);
        ...
  pthread_mutex_unlock(&queue_lock);
  return proc;
}

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

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

Listing 2.14: Lock mechanism for scheduler funtions

Additionally, when implementing syscall 101, the code in that section also modifies the processes stored in running_list and mlq_ready_queue. Therefore, to ensure that it does not interfere with the scheduler, we must use a lock in sched.c for this function.
To achieve this, I declared extern pthread_mutex_t queue_lock in queue.h—a header file included by both sched.c and sys_killall.c—to ensure that queue_lock is a unique and shared instance between the two files.
The placement of the locking mechanism in sys_killall.c is shown below:

```
int __sys_killall(struct pcb_t *caller, struct sc_regs* regs)
{
    ...
    /* Traverse running_list to terminate matching processes */
    if (running_list != NULL) {
        pthread_mutex_lock(&queue_lock);
        for (int i = 0; i < running_list->size; i++) {
            ...
        }
```

```
10          pthread_mutex_unlock(&queue_lock);
11      }
12
13  #ifdef MLQ_SCHED
14      /* Traverse mlq_ready_queue to terminate matching processes */
15      struct queue_t *mlq_ready_queue = caller->mlq_ready_queue;
16      if (mlq_ready_queue != NULL) {
17          pthread_mutex_lock(&queue_lock);
18          for (int i = 0; i < MAX_PRIO; i++) {
19              ...
20          }
21          pthread_mutex_unlock(&queue_lock);
22      }
23  #endif
24
25      return 0;
26  }
```

Listing 2.15: Lock mechanism for functions in sys_killall.c

Besides the scheduler, memory management is also an area where race conditions can occur. Since the system works with both physical and virtual memory, the likelihood of encountering such issues is high. The example below demonstrates a race condition when allocating memory for a process. In this test, the OS is configured with 7 CPUs and runs 14 processes. Each process performs 200 allocations. Since each frame is 256 bytes, every process requires 4 frames, totaling 56 frames (from 0 to 55). However, when running the test multiple times without a locking mechanism, race conditions occasionally occur (see Figure 2.12).

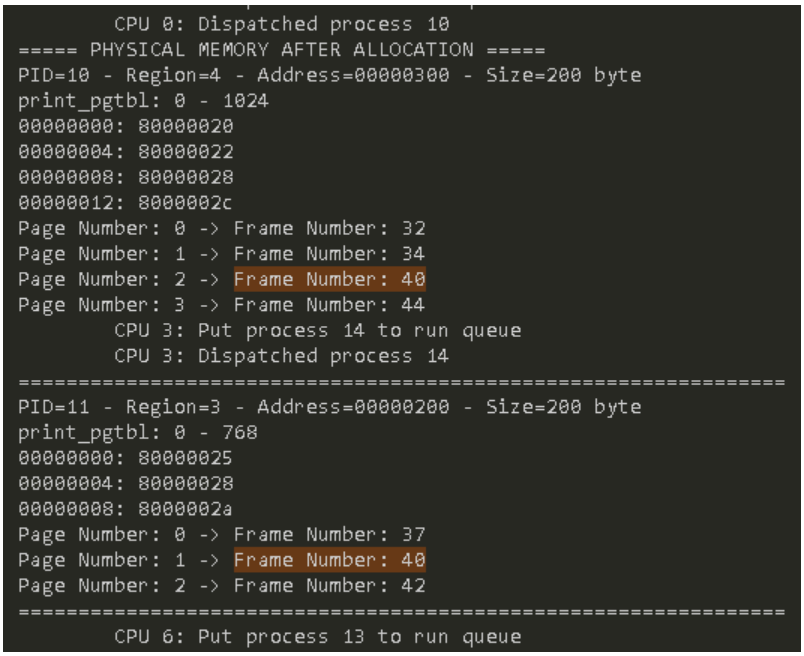


Figure 2.12: Race condition caused by not using lock during memory allocation

Moreover, Table 2.1 shows which frames are allocated to which processes. From the table and Figure 2.12, we can observe that both processes 10 and 11 were allocated frame 40. Therefore, the result is incorrect.

PID	**Frame**				**PID**	**Frame**			
1	0	2	3	8	**8**	25	27	31	36
2	1	5	6	12	**9**	29	33	35	38
3	4	9	11	16	**10**	32	34	40	44
4	7	10	14	18	**11**	37	40	42	47
5	13	17	19	23	**12**	39	43	46	49
6	15	20	22	26	**13**	41	45	50	52
7	21	24	28	30	**14**	48	51	53	54

Table 2.1: Frame allocation for processor

To ensure correct operation, all functions that allow processors to access or modify physical memory must be protected with a locking mechanism. These functions are shown below:

```
int __alloc(struct pcb_t *caller, int vmaid, int rgid, int size,
    int *alloc_addr)
{
    ...
    pthread_mutex_lock(&mmvm_lock);
    syscall(caller, 17, &regs);
    pthread_mutex_unlock(&mmvm_lock);
    ...
    return 0;
}

int pg_getpage(struct mm_struct *mm, int pgn, int *fpn, struct
    pcb_t *caller)
{
    pthread_mutex_lock(&mmvm_lock);
    uint32_t pte = mm->pgd[pgn];
    if (!PAGING_PAGE_PRESENT(pte))
    { /* Page is not online, make it actively living */
        ... // Calling syscall 2 times
    }
    *fpn = PAGING_FPN(mm->pgd[pgn]);
    pthread_mutex_unlock(&mmvm_lock);
    return 0;
}

int pg_getval(struct mm_struct *mm, int addr, BYTE *data, struct
    pcb_t *caller)
{
    ...
    pthread_mutex_lock(&mmvm_lock);
    syscall(caller, 17, &regs);
    *data = regs.a3;
    pthread_mutex_unlock(&mmvm_lock);
    return 0;
}

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

```
35 {
36     ...
37     pthread_mutex_lock(&mmvm_lock);
38     syscall(caller, 17, &regs);
39     pthread_mutex_unlock(&mmvm_lock);
40     return 0;
41 }
42
43 int libread(...)
44 {
45     ...
46 #ifdef IODUMP
47     pthread_mutex_lock(&mmvm_lock);
48     ...
49     MEMPHY_dump(proc->mram);
50     pthread_mutex_unlock(&mmvm_lock);
51 #endif
52     return val;
53 }
54
55 int libwrite(...)
56 {
57     ...
58 #ifdef IODUMP
59     pthread_mutex_lock(&mmvm_lock);
60     ...
61     MEMPHY_dump(proc->mram);
62     pthread_mutex_unlock(&mmvm_lock);
63 #endif
64     return result;
65 }
```

Listing 2.16: Lock mechanism for functions in libmem.c