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

REPORT ASSIGNMENT

OPERATING SYSTEM

Class: CC01 - HK251
INSTRUCTOR: NGUYEN PHUONG DUY

TOPIC: DESIGN SIMPLE OPERATING SYSTEM

student's name	Student's ID
Le Phuc Khang	235
Tran Quoc Thang	2353125
Tran Xuan Hao	235
Nguyen Xuan Ngoc	235
Phan Tuan Kiet	2352654

Ho Chi Minh City, 2025

GROUP WORK RESULTS REPORT

Number	Full name	Student ID	Contribution
1	Le Phuc Khang	235	100%
2	Tran Quoc Thang	2353125	100%
3	Nguyen Xuan Ngoc	235	100%
4	Phan Tuan Kiet	2352654	100%
5	Tran Xuan Hao	235	100%

Contents

1 Theorical Background

1.1 Scheduler

The scheduler is the central component of the kernel responsible for deciding which process runs at any given time. Our implementation is grounded in the theoretical framework of CPU Scheduling described in Chapter 5 of *Operating System Concepts*. Specifically, we construct a **Symmetric Multiprocessing (SMP)** system utilizing a **Preemptive Multilevel Queue (MLQ)** algorithm with a **Time-Slicing allocation strategy** to address starvation.

1. **Multilevel Queue Scheduling (Theory of Partitioning):**
   - **Theoretical Basis:** According to *Section 5.3.5*, processes in a system often have different response-time requirements (e.g., interactive/foreground vs. batch/background). A Multilevel Queue (MLQ) algorithm partitions the Ready Queue into several separate queues, each assigned to a different class of processes.
   - **System Design:** In our assignment, we define `MAX_PRIO` distinct queues. Processes are permanently assigned to a specific queue based on their `priority` attribute. Unlike Multilevel Feedback Queues (MLFQ), which allow processes to move between queues to define their behavior dynamically, our standard MLQ implementation reduces runtime overhead ($\mathcal{O}(1)$ queue selection) by maintaining fixed process classification.
2. **Intra-Queue Scheduling: Round-Robin (RR) and Preemption:**
   - **Theoretical Basis:** To support time-sharing, *Section 5.3.3* introduces Round-Robin (RR) scheduling. RR is theoretically defined as FCFS scheduling with **Preemption**. A small unit of time, called a *Time Quantum*, is defined. If a process's CPU burst exceeds this quantum, the system timer generates an interrupt, causing the OS to preempt the process and move it to the tail of the ready queue.
   - **System Design:** We implement this by assigning a `time_slot` to the system. The `queue.c` module strictly enforces FIFO behavior (`dequeue` from head, `enqueue` to tail), which, combined with the timer interrupt in `os.c`, realizes the Round-Robin logic. This ensures **fairness** among processes of the same priority and minimizes the average response time.
3. **Inter-Queue Scheduling: Solving Starvation via Time-Slicing:**
   - **The Starvation Problem:** A fundamental flaw of fixed-priority scheduling is **Indefinite Blocking** (or Starvation), where low-priority tasks may never execute if a continuous stream of high-priority tasks arrives.
   - **Theoretical Solution:** The textbook proposes two solutions: *Aging* or *Time Slicing among queues*. In the Time Slicing approach, each queue gets a certain portion of the CPU time (e.g., 80% for foreground, 20% for background).
   - **System Design (Slot Mechanism):** We implement the **Time Slicing** solution using a deterministic slot mechanism. Each queue i is allocated a specific budget: $slot[i] = MAX\_PRIO - i$. The

scheduler iterates through queues; if a high-priority queue consumes its allocated slots, the scheduler is theoretically forced to "yield" to lower-priority queues, even if the high-priority queue is non-empty. This converts the algorithm from "Absolute Priority" to "Proportional Share," mathematically guaranteeing that even the lowest priority queue eventually receives CPU cycles.

4. **Symmetric Multiprocessing (SMP) and Load Balancing:**

   - **Theoretical Basis:** *Section 5.5* distinguishes between Asymmetric and Symmetric Multiprocessing. In SMP, each processor is self-scheduling. The textbook highlights two architectural choices for the ready queue: (1) Each processor has its own private queue, or (2) All processors share a **Common Ready Queue**.
   - **System Design:** We adopt the **Common Ready Queue** architecture. This design inherently solves the *Load Balancing* problem because no processor sits idle while there are tasks in the global queue (Work Stealing is not needed).
   - **Synchronization Theory:** A critical implication of a shared queue is the potential for **Race Conditions** (two CPUs picking the same process). To adhere to the mutual exclusion principles (Chapter 6), we implement a coarse-grained locking mechanism using `pthread_mutex_t`. Access to the scheduler (enqueue/dequeue operations) constitutes a **Critical Section** and is strictly serialized.

1.2 Memory Management

1.2.1 Memory management

The fundamental task of an operating system is to manage the memory hierarchy and provide an abstraction mechanism to separate the user process from the physical details of the hardware.

a. Logical vs. Physical Address Space According to **Chapter 9.1.3**, the central concept of memory management is the separation of logical address space from physical address space:

- **Logical Address:** An address generated by the CPU during program execution. To a process, memory appears as a flat, continuous space ranging from address 0 to $Max_{Virtual}$. The set of all valid logical addresses for a process is defined as the *Logical Address Space*.
- **Physical Address:** The actual address seen and accessed by the memory unit on the physical RAM.
- **Memory Management Unit (MMU):** A hardware device responsible for the run-time mapping from virtual to physical addresses. Under this scheme, the user program never interacts with the real physical addresses.

b. Process Memory Layout Although the virtual address space is linear, logically a process is typically organized into distinct sections to separate access rights and usage purposes (Chapter 9.3):

- **Text section:** contains the executable code of the program (usually read-only).
- **Data section:** stores global and static variables.
- **Heap:** region for dynamic allocations that conceptually grows upward.

- **Stack:** holds stack frames and local variables, growing downward.

In full-featured UNIX-like systems, each of the above logical sections is usually represented by one or more Virtual Memory Areas (VMAs) maintained in a linked list inside the process's `mm_struct`.

In our simple OS implementation, we adopt a slightly different but simpler model. Each process owns a single user `vm_area_struct` that covers its whole logical user space (`vmaid = 0`). Inside this VMA, we represent individual logical "segments" by region descriptors `vm_rg_struct`, whose start and end addresses (`rg_start`, `rg_end`) are recorded in the symbol region table `symrgtbl[]`. Free holes between regions are maintained in the VMA's `vm_freerg_list`.

Therefore, in this project:

- segment $\approx$ `vm_rg_struct` region inside one VMA,
- while the VMA itself simply defines the outer bounds of the user address space for the process.

1.2.2 Multi-level paging

To implement memory mapping in modern architectures, **Paging** is widely used to eliminate external fragmentation.

a. The Page Table Problem in 64-bit Systems In traditional 32-bit architectures, a two-level paging scheme is often sufficient. However, with a 64-bit architecture (2^{64} bytes of address space), the size of the page table becomes prohibitively large if stored contiguously.

According to **Chapter 9.4.2 (Hierarchical Paging)**, the optimal solution is to divide the page table into smaller pieces. This allows the operating system to avoid allocating the entire page table structure in contiguous memory. Only the necessary sub-tables are allocated (on-demand allocation).

b. 5-Level Paging Architecture The system is designed based on an extension of the Intel x86-64 architecture, utilizing a **5-level paging scheme** to support a significantly larger virtual address space (up to 57-bit linear address).

The virtual address structure is decomposed into index fields and an offset. Assuming a standard page size of 4KB (2^{12} bytes), the virtual address is resolved as follows:

$$\texttt{Virtual\_Address} = PGD \oplus P4D \oplus PUD \oplus PMD \oplus PT \oplus Offset \tag{1}$$

The address translation process (also known as the **Page Walk**) occurs sequentially through 5 tables:

1. **PGD (Page Global Directory):** The top-level table (Level 5).
2. **P4D (Page Level 4 Directory):** The Level 4 table.
3. **PUD (Page Upper Directory):** The Level 3 table.
4. **PMD (Page Middle Directory):** The Level 2 table.
5. **PT (Page Table):** The final level (Level 1), containing the mapping to the physical frame.

Each entry in the upper-level tables points to the base address of the next lower-level table. The entry in the `PT` (Page Table Entry - PTE) contains the actual **Physical Frame Number (PFN)**.

1.2.3 Comparative Analysis of Page Table Architectures

a. Hierarchical Paging

When applying Single-level paging on a 64-bit system with a 4KB (2^{12}) page size, the virtual address space contains approximately 2^{52} pages. If a single page table were used, the Operating System would be required to allocate a massive, contiguous block of physical memory to store all Page Table Entries for each process. This is practically impossible due to the sheer size exceeding RAM limits and the difficulty of finding contiguous memory caused by fragmentation.

Hierarchical Paging resolves this by dividing the logical address into multiple parts to manage the page table as a tree structure (typically 4-level or 5-level paging in modern systems). This method offers two key advantages:

- **Memory Efficiency:** The system does not need to store the entire page table structure in RAM. Due to the sparse nature of address spaces, Inner Tables are only initialized for the virtual memory regions that the process actually uses.
- **Non-contiguous Allocation:** Only the highest-level table (Outer Page Table)—which is usually very small—needs to be contiguous. The Inner Tables can be scattered anywhere in physical memory, reducing the burden on the OS's memory management.

However, this method comes with significant disadvantages regarding performance and complexity:

- **Access Latency:** The number of memory accesses increases with the number of paging levels. For instance, with 5-level paging, the CPU requires 6 memory accesses to retrieve a single byte of data (5 for the page table walk and 1 for the actual data), significantly slowing down processing speed.
- **Design Complexity:** Both the hardware (specifically the MMU) and the Operating System must be designed with greater complexity to handle the page walk mechanism accurately.

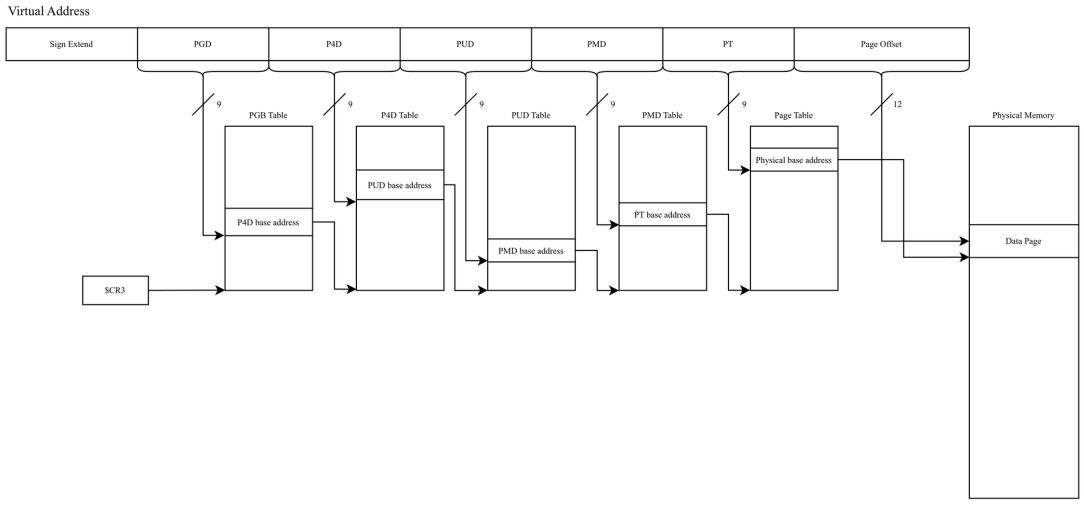

Figure 1: 5-Level Paging Scheme (Address Translation for 57-bit Virtual Address)

b. Hashed Page Tables

This method utilizes a **Hash Table** to manage page table entries. Instead of using linear storage or a hierarchical tree structure, the Virtual Page Number (VPN) is passed through a hash function to determine its storage location in the table. This approach is particularly effective for systems with a "sparse address space," where the majority of virtual addresses are not in use. Each entry in the hash table typically serves as the head of a linked list (to handle hash collisions). Each element in the list consists of three main fields:

- **Virtual Page Number (VPN):** Used to identify the page.
- **Physical Frame Number (PFN):** The corresponding mapped physical frame value.
- **Pointer:** A reference to the next element in the linked list.

Advantages

- **Optimal Memory Efficiency:** The size of the hashed page table is proportional to the number of *physical pages actually in use*, rather than the size of the entire virtual address space. This results in significant memory savings in massive 64-bit systems.
- **Flexible Structure:** Since it does not require storing information for unused memory regions, this method eliminates memory waste associated with page tables for unallocated address ranges.

Disadvantages

- **Variable Memory Access Cost:** Performance depends heavily on the quality of the hash function. High collision rates result in longer linked lists, forcing the CPU to perform multiple memory accesses to traverse the list, thereby reducing processing speed.
- **Hardware Design Complexity:** Traversing a linked list is more difficult to implement in hardware (MMU) compared to walking a tree structure in hierarchical paging.
- **Poor Locality of Reference:** Due to the random nature of hashing, pages that are contiguous in virtual memory may be scattered across the hash table, reducing the efficiency of the CPU Cache.

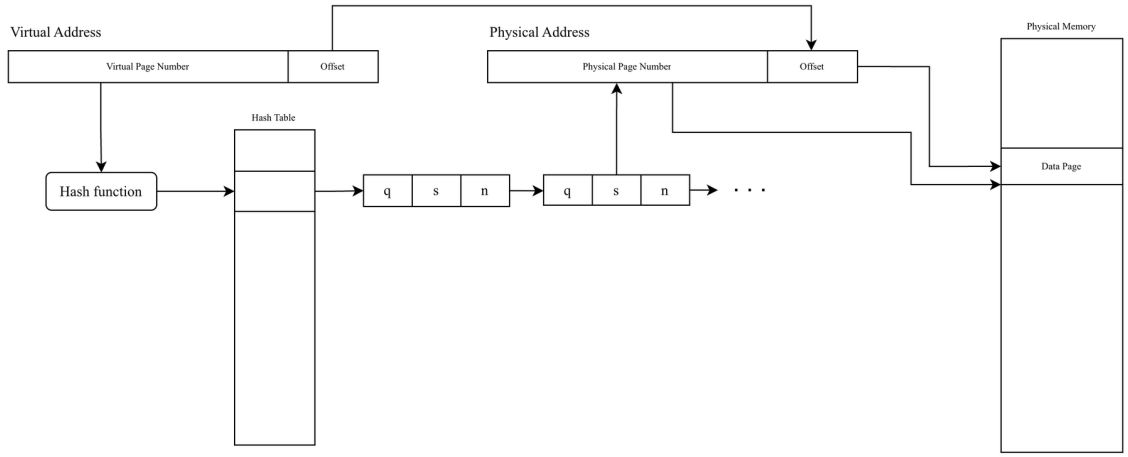

Figure 2: Hashed Page Tables

Variation: Clustered Page Tables

A common variation of this method is **Clustered Page Tables**. Instead of storing a single page per entry in the list, it stores mappings for multiple contiguous pages. This technique helps reduce the length of the linked list and improves search performance for large memory blocks.

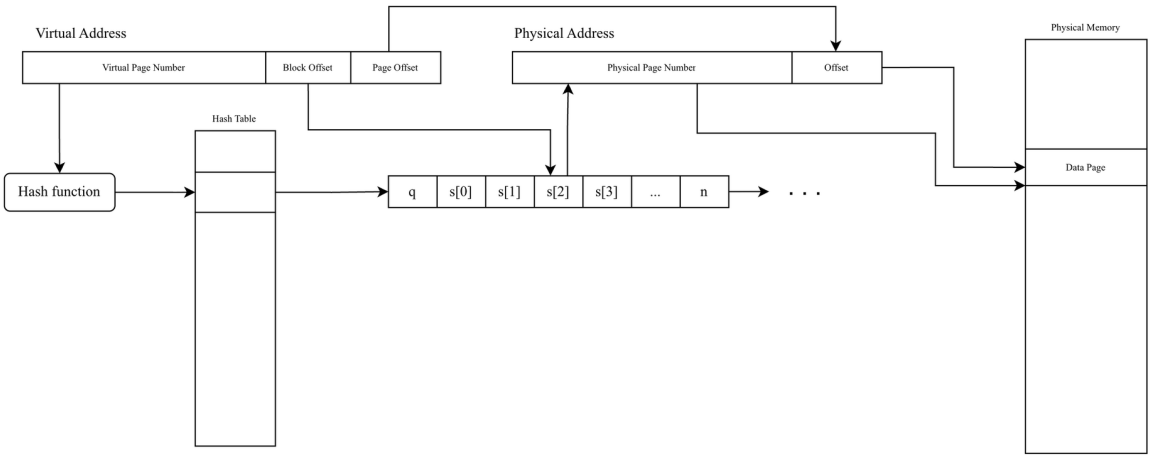

Figure 3: Clustered Page Tables

c. Inverted Page Tables

In both Hierarchical Paging and Hashed Page Tables methods, each process maintains its own separate page table. Each table contains an entry for every virtual page the process is using. This results in the system managing millions of entries, consuming a significant amount of physical memory just to store page table structures to track memory usage.

To solve this problem, the **Inverted Page Table** method was developed. Instead of indexing based on the virtual address space of each process, this method indexes based on the frames of the actual **physical memory**.

Structure and Mechanism

- **Global Nature:** There is only one Global Page Table for the entire system, regardless of the number of running processes.
- **Fixed Size:** The number of entries in this table corresponds exactly to the number of **Physical Frames** in RAM. For example, if RAM has 1 million frames, the table will have 1 million entries.
- **Entry Structure:** The i-th entry in the table contains information about the virtual page stored in the i-th physical frame. Each entry typically consists of:
  1. **Virtual Page Address:** To identify the logical page.
  2. **Process ID (PID):** Information about the process that owns the page (essential because different processes may generate identical virtual addresses).

Advantages

- **Maximum Memory Efficiency:** This is the most significant advantage. The size of the page table is fixed and depends only on the size of physical RAM, making it independent of the number of processes or the size of the virtual address space.

Disadvantages

- **High Lookup Latency:** Since the table is sorted by physical address (while the CPU queries based on virtual address), the system cannot use direct indexing ($O(1)$) as in traditional methods. Instead, it must perform a search operation. Although often assisted by a Hash Table, handling collisions and traversing the table is inherently slower than direct array access.
- **Difficulty in Shared Memory:** This is the major drawback. In traditional paging, memory sharing is achieved by having entries in different process page tables point to the same physical frame. However, in an Inverted Page Table, each physical frame corresponds to exactly **one single entry**. Therefore, it can only store one ⟨PID, VPN⟩ pair, making the implementation of shared memory complex and often requiring auxiliary data structures.

d. Reasons for Choosing Hierarchical Page Tables In this simple operating system simulation project, we selected Hierarchical Page Tables (5-level Paging Scheme) for the following three reasons:

- **Dynamic Allocation:**

  The hierarchical structure enables "on-demand" memory allocation. There is no need to initialize the entire massive data structure upfront. When a process requires access to a memory region, we simply invoke `alloc_aligned_table()` for that specific branch. This approach effectively prevents the wastage of management memory within a 64-bit environment.

- **Simplified Implementation of Sharing and Protection:**

  The Address-Translation model allows for the assignment of access permissions (Read/Write/Execute) at each level of the tree (from top-level directories down to individual pages). This facilitates the implementation of Memory Protection, allowing for easy setup and granular control over specific memory regions.

- **Ease of Implementation:**

  Compared to Inverted Page Tables or Hashed Page Tables, the hierarchical structure is easier to implement within a C-based simulation environment. We primarily rely on bitwise shifts and array indexing to decompose virtual addresses into their respective indices .

2 Implementation

2.1 Scheduler

2.1.1 Queue Operations (Implementation of Round-Robin)

To support the Round-Robin scheduling within each priority level, the queue operations in `queue.c` must strictly enforce First-In-First-Out (FIFO) behavior. I have implemented three core functions as follows:

a. Enqueue Operation (Arrival/Preemption) The `enqueue` function handles the arrival of a new process or the return of a preempted process.

- **Mechanism:** It places the process at index `q->size`. This is critical for Round-Robin: a process finishing its time slice is moved to the back of the line. We check for buffer overflow (`MAX_QUEUE_SIZE`) before assignment to ensure memory safety.

```
void enqueue(struct queue_t *q, struct pcb_t *proc) {
    /* TODO: put a new process to queue [q] */
    if (q->size < MAX_QUEUE_SIZE) {
        q->proc[q->size] = proc;
        q->size++;
    }
}
```

Listing 1: Enqueue Implementation

b. Dequeue Operation (Dispatching) The `dequeue` function selects the next process for execution.

- **Mechanism:** It selects the process at index 0 (the Head), which corresponds to the process that has been waiting the longest. Since we use a static array, removing index 0 leaves a "hole". The function includes a loop to shift all subsequent elements $(i + 1)$ to position i. This ensures the array remains contiguous and the next candidate is always at index 0.

```
struct pcb_t *dequeue(struct queue_t *q) {
    /* TODO: return a pcb whose prioprity is the highest
     * in the queue [q] and remember to remove it from q */
        if (q == NULL || q->size == 0) {
        return NULL;
    }
    struct pcb_t *first_proc = q->proc[0];

    int i;
    for (i = 0; i < q->size - 1; i++) {
        q->proc[i] = q->proc[i + 1];
    }
    q->size--;
```

```
14      return first_proc;
15  }
```

Listing 2: Dequeue Implementation

2.1.2 MLQ Scheduler Implementation (sched.c)

The `sched.c` module implements the core logic of the Multi-Level Queue scheduler. This implementation is designed to handle process retrieval and placement within a Symmetric Multiprocessing (SMP) environment, strictly following the Slot-based policy defined in the assignment specifications.

a. Scheduler Initialization (`init_scheduler`) This function sets up the system state before any process execution begins.

- **Slot Pre-calculation:** Instead of calculating the slot quota at runtime, we initialize the `slot` array immediately using the formula $slot[i] = MAX\_PRIO - i$.
- **Reasoning:** This optimization reduces CPU overhead during the critical context-switching phase. By pre-filling the quotas, the scheduler simply needs to decrement or reset values during execution.
- **Mutex Initialization:** The `pthread_mutex_init` call prepares the lock required for thread safety in the SMP architecture.

```
1  void init_scheduler(void) {
2      int i;
3      for (i = 0; i < MAX_PRIO; i++) {
4          mlq_ready_queue[i].size = 0;
5          slot[i] = MAX_PRIO - i; // Pre-calculate quotas
6      }
7      // ...
8      pthread_mutex_init(&queue_lock, NULL);
9  }
```

Listing 3: Initialization Logic

b. Process Retrieval Logic (`get_mlq_proc`): The `get_mlq_proc` function serves as the core of the scheduler, responsible for selecting the next process for CPU allocation from the set of priority queues. The algorithm is designed based on the Multi-Level Queue (MLQ) model combined with a *Slot-based Allocation* mechanism to address the issue of resource starvation.

1. **General Operating Principle**

   The function iterates sequentially through the queues from the highest priority (`prio = 0`) to the lowest (`prio = MAX_PRIO - 1`). The decision to select a process is based on two prerequisites:

   - **Readiness:** The queue at that priority level must have waiting processes (`!empty`).
   - **Resource Quota (Slot):** The queue must have remaining CPU usage quota (`slot[i] > 0`).

2. **Algorithm Flow**

The process selection procedure occurs through the following steps:

- **Step 1: Critical Section Protection.** Since the system simulates a multi-processor environment, the function begins by locking the Mutex (`pthread_mutex_lock(&queue_lock)`). This ensures the data integrity of the `mlq_ready_queue` when multiple CPUs access it simultaneously.
- **Step 2: Priority Traversal.** A loop iterates from `i = 0` (highest priority) to `MAX_PRIO`. At each priority level `i`:
  - *Check Empty:* If `mlq_ready_queue[i]` is empty, the system skips it and checks the next priority level.
  - *Check Slot (Core Logic):*
    - **Case A (Slot Remaining - `slot[i] > 0`):** The queue is allowed to run. The system retrieves the first process (dequeue) and decrements the remaining slots by 1 (`slot[i]--`). **Decision:** Break the loop immediately to return this process to the CPU.
    - **Case B (Slot Exhausted - `slot[i] == 0`):** The queue has used up its quota for the current cycle. The system performs a Slot Refill using the formula: `slot[i] = MAX_PRIO - i`. **Decision:** Crucially, the algorithm DOES NOT select a process from this queue but continues the loop (`continue`) to check the next lower priority queue.
- **Step 3: System State Update.** If a process is found (`proc != NULL`), it is added to the running list (`running_list`) for tracking.
- **Step 4: Finish.** Unlock the Mutex (`pthread_mutex_unlock`) and return the `pcb_t` pointer of the selected process (or `NULL` if no viable process exists).

3. **Slot-based Mechanism Analysis**

This implements a mechanism of **"Conditional Yielding"**:

- **Purpose:** To prevent high-priority processes (e.g., Priority 0) from monopolizing the CPU indefinitely.
- **Operation:** When a high-priority queue consumes its allocated slots (e.g., 140 slots for Prio 0), it is forced to reset its slots and "yield" the checking turn to lower-priority queues within the current function call. This ensures relative fairness, allowing lower-priority processes (such as Prio 139) a chance to execute (CPU time) even under heavy load conditions.

4. **Complexity Analysis**

In the worst-case scenario, the algorithm must traverse through all `MAX_PRIO` queues.
Complexity: $O(K + N)$

- K: The number of priority levels (`MAX_PRIO`).
- N: The cost of the `dequeue` operation.

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

    pthread_mutex_lock(&queue_lock);

    for (int i = 0; i < MAX_PRIO; i++) {
        if (!empty(&mlq_ready_queue[i])) {
            if (slot[i] > 0) {
                proc = dequeue(&mlq_ready_queue[i]);
                slot[i]--;
                break;
            } else {
                slot[i] = MAX_PRIO - i;
            }
        }
    }

    if (proc != NULL)
        enqueue(&running_list, proc);

    pthread_mutex_unlock(&queue_lock);

    return proc;
}
}
```

Listing 4: Slot-based Decision Logic

c. Process Placement (`put_mlq_proc` / `add_mlq_proc`) These functions handle putting a process back into the ready queue (e.g., after a time slice expires or upon creation).

- **Mechanism:** They map the process's priority (`proc->prio`) to the correct index in the `mlq_ready_queue` array and invoke the FIFO `enqueue` operation.
- **Synchronization:** These operations are wrapped in mutex locks to ensure the queue structure is not corrupted by concurrent access from the loader or other CPUs.

d. Synchronization Mechanism (Mutex Protection) A critical aspect of this implementation is the handling of Shared Resources in a multi-core environment.

In this simulation, the `mlq_ready_queue` is a **Global Shared Resource**.

- *Scenario without Mutex:* CPU 0 checks queue 0 and sees 1 process. Simultaneously, CPU 1 checks queue 0 and sees the same process. Both CPUs try to `dequeue` the same pointer. This creates a **Race Condition**, leading to memory corruption or one process being executed twice.

We use `pthread_mutex_lock(&queue_lock)` to define a **Critical Section**.

- **Scope:** The lock covers the entire duration of reading the state (checking `empty` and `slot`) and modifying the state (`dequeue`, `enqueue`, $slot--$).
- **Atomicity:** This ensures that the complex operation of "Check Slot $\rightarrow$ Dequeue $\rightarrow$ Decrement Slot" appears as a single atomic instruction to other CPUs. No other thread can interrupt or modify the queue during this sequence.

```
pthread_mutex_lock(&queue_lock); // BEGIN Critical Section
// ...
// Safe access to shared mlq_ready_queue and slot arrays
// ...
pthread_mutex_unlock(&queue_lock); // END Critical Section
```

Listing 5: Critical Section Protection

2.1.3 Scheduler Algorithm Conclusion

This section presented the implementation of a **Slot-based Multi-Level Queue (MLQ)** scheduler for an SMP environment. By integrating a quota mechanism ($slot = MAX\_PRIO - prio$) into the priority traversal logic, the design successfully balances two conflicting objectives: prioritizing critical tasks while preventing the starvation of lower-priority processes. Additionally, the rigorous application of **Mutex locks** ensures thread safety, preventing race conditions as multiple CPUs concurrently access the shared ready queue structure.

2.2 Memory Management

2.2.1 Memory management

a. Overview This module implements logical memory management per process in a simplified way. Each process owns an `mm_struct` that contains:

- One user `vm_area_struct` (`vmaid = 0`) acting as the container for the whole user address space,
- A fixed-size symbol region table `symrgtbl[]` that stores allocated regions by `rgid`,
- And the root pointers to the multi-level page tables handled by `mm64.c`.

Instead of multiple VMAs for text/data/heap/stack, our design keeps a single user VMA and represents logical "segments" as virtual memory regions (`vm_rg_struct`, VMR) inside that VMA. Each VMR is a contiguous range `[rg_start, rg_end)`, recorded in `symrgtbl[rgid]`, while free gaps are chained in `vm_freerg_list` for reuse.

The core design principles are:

- **VMA / region management:** All user regions live inside one user `vm_area_struct`; logical segments are `vm_rg_struct` descriptors, and free holes are kept in `vm_freerg_list`.
- **User / Kernel separation:** User code never touches kernel structures (`pcb_t`, `mm_struct`); it uses system calls and PID, and the kernel side looks up the correct PCB/MM before operating.

- **Allocation strategy:** Always try to reuse holes from `vm_freerg_list` first; if none fits, expand the VMA (`sbrk` / `vm_end`) via the `SYSMEM_INC_OP` syscall, then let the paging module map the new pages.

b. Implementation Details The following code snippets illustrate the core logic handling VMA organization and the secure User-Kernel interface.

1. VMA Data Structure (`os_mm.h`)

The `vm_area_struct` represents a contiguous range of logical addresses $[vm\_start, vm\_end]$.

```
struct vm_area_struct {
    unsigned long vm_id;     // Region ID (e.g., DATA, HEAP)
    unsigned long vm_start; // Logical Start Address
    unsigned long vm_end;   // Logical End Address
    struct vm_rg_struct *vm_freerg_list; // List of free holes for reuse
    struct vm_area_struct *vm_next;      // Pointer to next VMA
};
```

Listing 6: Virtual Memory Area Structure

2. Secure User/Kernel Interface (`libstd.c`)

This wrapper demonstrates the protection mechanism. The user space only passes value arguments and the `pid`, never the raw PCB pointer, ensuring isolation.

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

    /* CRITICAL: Only caller->pid is passed to the kernel.
       The Kernel must resolve the PCB internally. */
    return syscall(caller->krnl, caller->pid, syscall_idx, &regs);
}
```

Listing 7: User Space System Call Wrapper

3. Memory Allocation Algorithm - `__alloc` (`libmem.c`)

This function executes in **Kernel Mode** after the PID has been validated. It implements the logic to search for available memory space within the VMA.

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

    // Check if we can reuse a free region
    if (get_free_vmrg_area(caller, vmaid, size, &rgnode) == 0)
    {
```

```
        caller->mm->symrgtbl[rgid].rg_start = rgnode.rg_start;
        caller->mm->symrgtbl[rgid].rg_end = rgnode.rg_end;
        *alloc_addr = rgnode.rg_start;
        pthread_mutex_unlock(&mmvm_lock);
        return 0;
    }

    /* If get_free_vmrg_area FAILED, we must expand the heap */
    struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);
    if (!cur_vma)
    {
        pthread_mutex_unlock(&mmvm_lock);
        return -1;
    }

    addr_t inc_sz = PAGING_PAGE_ALIGNSZ(size);
    // Usually aligning to page size is good practice, or use exact size if your
        inc_vma handles it.
    // The provided skeleton had some ifdef logic, let's stick to standard alignment.

    addr_t old_sbrk = cur_vma->sbrk;

    /* SYSCALL to increase limit */
    struct sc_regs regs;
    regs.a1 = SYSMEM_INC_OP;
    regs.a2 = vmaid;
    regs.a3 = inc_sz; // Request expansion

    // Note: In real syscall, we don't pass PCB, but here we simulate it via wrapper
    // The wrapper 'syscall' takes (krnl, pid, nr, regs)
    if (syscall(caller->krnl, caller->pid, 17, &regs) < 0)
    {
        pthread_mutex_unlock(&mmvm_lock);
        return -1; // Failed to expand
    }

    /* Successful increase limit */
    caller->mm->symrgtbl[rgid].rg_start = old_sbrk;
    caller->mm->symrgtbl[rgid].rg_end = old_sbrk + size;
    *alloc_addr = old_sbrk;

    // The remaining space (inc_sz - size) should technically be added to free list
    // to avoid internal fragmentation, but for this simple assignment we might skip
        it
```

```
    // or add logic:
    if (inc_sz > size)
    {
        struct vm_rg_struct *fragment = malloc(sizeof(struct vm_rg_struct));
        fragment->rg_start = old_sbrk + size;
        fragment->rg_end = old_sbrk + inc_sz;
        fragment->rg_next = NULL;
        enlist_vm_freerg_list(caller->mm, fragment);
    }

    pthread_mutex_unlock(&mmvm_lock);
    return 0;
}
```

Listing 8: Kernel Space Allocation Routine

4. Memory Deallocation Algorithm - `enlist_vm_freerg_list` (`libmem.c`)

This function inserts a recently freed memory region back into the reuse list (free list), marking it available for future allocations.

```
int enlist_vm_freerg_list(struct mm_struct *mm, struct vm_rg_struct *rg_elmt)
{
    struct vm_rg_struct *rg_node = mm->mmap->vm_freerg_list;

    if (rg_elmt->rg_start >= rg_elmt->rg_end)
        return -1;

    if (rg_node != NULL)
        rg_elmt->rg_next = rg_node;

    /* Enlist the new region */
    mm->mmap->vm_freerg_list = rg_elmt;

    return 0;
}
```

Listing 9: Deallocation Logic

c. Algorithm Description Step 1: User-Kernel Transition (Protection Mechanism) To ensure memory safety (referencing *Silberschatz Chapter 2*), the allocation process begins with a System Call:

- **User Request:** The user program calls a library function (e.g., `malloc`), which triggers the wrapper `libsyscall`.
- **Trap to OS:** `libsyscall` passes the Process ID (`PID`) and an operation code (such as `SYSMEM_INC_OP` in our allocation path, or `SYSMEM_MAP_OP` in other mapping scenarios) to the Kernel via a software interrupt/trap.

- **Authentication:** Upon entering Kernel Mode, the kernel uses the provided `PID` to traverse the process table and retrieve the correct `pcb_t` pointer. This prevents any user process from manipulating the memory of another process.

Step 2: VMA Management (`get_vma_by_num`)

Once in Kernel Mode, the system must identify which memory region the process is requesting.

- **Activity:** The function `get_vma_by_num` traverses the linked list `mm->mmap` to find the VMA whose `vm_id` equals the requested `vmaid`.
- **Purpose:** In our implementation, there is exactly one user VMA per process (`vmaid = 0`), which acts as the container for all user regions. The logical separation between different user "segments" (variables/objects) is handled by `vm_rg_struct` entries stored in `mm->symrgtbl[]`, rather than by multiple VMAs.

Step 3: Memory Allocation (`__alloc`)

The allocation logic prioritizes filling "holes" in the memory space before expanding it, ensuring resource efficiency.

1. **Reuse Strategy:** The system inspects the `vm_freerg_list` of the current VMA. This list contains memory regions that were previously freed. The algorithm applies a **First-Fit** strategy: it traverses the list, and if a node satisfying $rg\_end - rg\_start \geq size$ is found, the system either splits the node or claims the entire node for the new allocation.
2. **Expansion Strategy:** If the free list is empty or contains no suitable regions, the system performs a linear expansion. The new allocation address starts at the current `vm_end`, and the boundary is updated: $vm\_end = vm\_end + size$. This behavior is analogous to the Unix `sbrk()` system call.
3. **Registration:** Finally, the system updates the `symrgtbl` (Symbol Region Table) with the `rgid`. This maps a User-managed Region ID to the actual Kernel-managed logical address.

Step 4: Memory Deallocation (`enlist_vm_freerg_list`)

When a process frees memory, the region is not immediately physically erased but transitioned to an "Available" state.

- **Logic:** The function receives a memory region structure (`rg_elmt`).
- **Operation:** It inserts this node into the head of the singly linked list `vm_freerg_list`.
- **Significance:** This turns used memory into a new "hole". The next allocation request (via `__alloc`) will scan and potentially reuse this region, thereby minimizing external fragmentation of the logical address space.

Conclusion:

The Memory Management module at the Logical Level has successfully established a secure memory management mechanism (via strict User/Kernel separation) and an efficient allocation strategy (via Free List reuse). This serves as the initial abstraction layer before these logical addresses are translated into physical addresses through the 5-Level Paging mechanism described in the next section.

2.2.2 Multi-level paging

a. Implementation Details The following core components constitute the paging mechanism.

1. Bit Masking and Address Decoding (mm64.h) The system uses macros to "peel off" each layer of the virtual address.

```
/* PGD Index: Extract 9 bits from position 48 to 56 */
#define PAGING64_ADDR_PGD_MASK    GENMASK64(56, 48)
#define PAGING64_ADDR_PGD_LOBIT   48
#define PAGING64_ADDR_PGD(addr)   ((addr & PAGING64_ADDR_PGD_MASK) >>
    PAGING64_ADDR_PGD_LOBIT)

/* P4D -> PUD -> PMD: Shift the bit window downwards */
#define PAGING64_ADDR_P4D(addr)   ((addr & GENMASK64(47, 39)) >> 39)
#define PAGING64_ADDR_PUD(addr)   ((addr & GENMASK64(38, 30)) >> 30)
#define PAGING64_ADDR_PMD(addr)   ((addr & GENMASK64(29, 21)) >> 21)

/* PT Index (Leaf Level): Bits 20-12 */
#define PAGING64_ADDR_PT_MASK     GENMASK64(20, 12)
#define PAGING64_ADDR_PT(addr)    ((addr & PAGING64_ADDR_PT_MASK) >> 12)

/* Offset: Last 12 bits */
#define PAGING64_ADDR_OFFST(addr) (addr & GENMASK64(11, 0))
```

Listing 10: Bit Manipulation Macros in mm64.h

Page Table Entry - PTE (mm64.h & mm64.c) The PTE stores the page state. Manipulating the PTE requires bitwise operations (|, &, ∼).

```
/* mm64.h - PTE Bit Definitions */
#define PAGING_PTE_PRESENT_MASK BIT_ULL(0)  // Bit 0: Page is in RAM
#define PAGING_PTE_DIRTY_MASK   BIT_ULL(6)  // Bit 6: Page was written to
#define PAGING_PTE_SWAPPED_MASK BIT_ULL(9)  // Bit 9: Page is swapped out
#define PAGING_PTE_FPN_MASK     GENMASK64(51, 12) // Bits 12-51: Frame Number

/* mm64.c - PTE Initialization Function */
int init_pte(addr_t *pte, int pre, addr_t fpn, int drt, int swp, int swptyp, addr_t
    swpoff) {
    if (pre != 0) { // If page exists (Present)
        if (swp == 0) { // Case 1: Page is in RAM
            SETBIT(*pte, PAGING_PTE_PRESENT_MASK);
            CLRBIT(*pte, PAGING_PTE_SWAPPED_MASK);
            /* Write FPN to bits 12-51 */
            SETVAL(*pte, fpn, PAGING_PTE_FPN_MASK, PAGING_PTE_FPN_LOBIT);
        } else { // Case 2: Page is Swapped
            SETBIT(*pte, PAGING_PTE_SWAPPED_MASK);
            CLRBIT(*pte, PAGING_PTE_PRESENT_MASK);
```

```
            /* Write Swap Offset */
            SETVAL(*pte, swpoff, PAGING_PTE_SWPOFF_MASK, PAGING_PTE_SWPOFF_LOBIT);
        }
    }
    return 0;
}
```

Listing 11: PTE Definition and Initialization

3. Page Walk Mechanism (`mm64.c`) The `get_page_table_entry` function is the core of translation, traversing 5 table levels.

```
uint64_t *get_page_table_entry(struct mm_struct *mm, addr_t addr, int alloc)
{
    // Get Indices
    int pgd_idx = PAGING64_ADDR_PGD(addr);
    int p4d_idx = PAGING64_ADDR_P4D(addr);
    int pud_idx = PAGING64_ADDR_PUD(addr);
    int pmd_idx = PAGING64_ADDR_PMD(addr);
    int pt_idx = PAGING64_ADDR_PT(addr);

    // Check Root (PGD)
    if (mm->pgd == NULL)
    {
        if (!alloc)
            return NULL;
        mm->pgd = alloc_aligned_table();

        if (!mm->pgd)
            return NULL;
        memset(mm->pgd, 0, sizeof(struct pgd_t));
    }

    // Walk PGD -> P4D
    struct p4d_t *p4d_table = get_next_level((uint64_t *)&mm->pgd->entries[pgd_idx],
        alloc);
    if (!p4d_table)
        return NULL;

    // Walk P4D -> PUD
    struct pud_t *pud_table = get_next_level((uint64_t *)&p4d_table->entries[p4d_idx
        ], alloc);
    if (!pud_table)
        return NULL;

    // Walk PUD -> PMD
```

```
    struct pmd_t *pmd_table = get_next_level((uint64_t *)&pud_table->entries[pud_idx
        ], alloc);
    if (!pmd_table)
        return NULL;

    // Walk PMD -> PT
    struct pt_t *pt_table = get_next_level((uint64_t *)&pmd_table->entries[pmd_idx],
        alloc);
    if (!pt_table)
        return NULL;

    // Return pointer to the specific Page Table Entry (Leaf)
    return (uint64_t *)&pt_table->entries[pt_idx];
}
```

Listing 12: Page Walk Logic (Simplified)

4. Hardware Simulation (`mm-memphy.c`) RAM is simulated not as a physical chip but as a BYTE array.

```
/* mm-memphy.c */
int MEMPHY_read(struct memphy_struct *mp, addr_t addr, BYTE *value) {
    if (mp == NULL) return -1;
    /* Direct access to storage array at index addr */
    *value = mp->storage[addr];
    return 0;
}

int MEMPHY_write(struct memphy_struct *mp, addr_t addr, BYTE data) {
    if (mp == NULL) return -1;
    mp->storage[addr] = data;
    return 0;
}
```

Listing 13: Physical Memory Access

b. Algorithm Description The system executes address translation via a strict procedure, simulating the hardware MMU (Memory Management Unit).

Step 1: Address Decoding The computer views the virtual address not as a large integer, but as a collection of indices.

- **Input:** 64-bit Virtual Address (VA).
- **Algorithm:** Use Bitwise AND (&) with a Mask to isolate specific bits, then Bitwise Right Shift (>>) to normalize the value into an integer index.
- **Example:** To get the PGD Index, the system extracts the highest 9 bits (56-48). This value determines the position in the PGD table.

Step 2: Page Walk (Tree Traversal) This is the core algorithm. Conceptually, the page table is a 5-level tree.

1. **Root:** Start from `mm->pgd` (stored in the CR3 register on real CPUs).
2. **Traversal:**
   - Use PGD Index to select an entry in PGD. This entry contains the physical address of the P4D table.
   - Use P4D Index to select an entry in P4D. This points to the PUD table.
   - Continue similarly through PUD and PMD.
3. **Leaf:** At the final level (PT), use the PT Index to retrieve the **PTE**.

Logic Flow: CR3 → [PGD] → [P4D] → [PUD] → [PMD] → [PT] → Frame.

Step 3: PTE Handling & State Check Once the PTE is retrieved, the system checks the status bits:

- **Case A: Bit Present = 1 (Hit):** The page is in RAM. The system extracts the 40-bit FPN (Frame Page Number) from the PTE.
- **Case B: Bit Present = 0 & Swapped = 1 (Page Fault):** The data is not in RAM but in Swap. The system triggers `__mm_swap_page`: Find a free frame → Read data from Swap → Update PTE (Present=1, Swapped=0).
- **Case C: Bit Present = 0 & Swapped = 0 (Invalid):** Accessing unallocated memory → Segmentation Fault.

Step 4: Physical Address Composition After obtaining the frame number (FPN), the final physical address is calculated:

$$PhysicalAddress = (FPN \times PageSize) + Offset$$

In code, this is optimized using bitwise operations:

- `FPN << 12`: Left shift 12 bits (equivalent to multiplying by 4096).
- `| Offset`: Bitwise OR with Offset (lower 12 bits of VA) to combine them.

Step 5: Physical Access Finally, the Physical Address (PA) is passed to `mm-memphy.c`.

- The module treats RAM as a massive array `storage[]`.
- It accesses `storage[PA]` to read or write the actual data byte.

Conclusion The successful implementation of the 5-Level Paging mechanism demonstrates memory management at the finest granularity. The Page Walk algorithm, combined with precise Bitwise operations, ensures the system can translate addresses within the vast 64-bit space while transparently handling complex scenarios like Page Faults and Swapping.

3 Interprets the results of running tests

3.1 Scheduling

3.1.1 The result input file of scheduler (sched)

The following output describes the behavior of the scheduler over multiple time slots, showing how processes are loaded, dispatched, and processed by the CPU.

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

```
        CPU 1: Dispatched process  2
Time slot   17
Time slot   18
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  3
Time slot   19
Time slot   20
        CPU 1: Put process  2 to run queue
        CPU 1: Dispatched process  2
Time slot   21
Time slot   22
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  3
        CPU 1: Processed  2 has finished
        CPU 1 stopped
Time slot   23
Time slot   24
Time slot   25
Time slot   26
        CPU 0: Processed  3 has finished
        CPU 0 stopped
```

Explanation of Output

The output demonstrates the behavior of the scheduler across multiple time slots and the corresponding actions performed by the CPU. Below is an analysis of key steps:

- **Time Slot 0 - 4 (Loading and Initial Dispatching):**
  - *Loading:* Processes are loaded sequentially. Process 1 (Prio 0) loads at slot 0; Process 2 (Prio 1) at slot 1; Process 3 (Prio 1) at slot 3; and Process 4 (Prio 0) at slot 4.
  - *Dispatching:* CPU 1 picks up Process 1 at slot 1. CPU 0 picks up Process 2 at slot 2. This shows the workload being distributed across both CPUs immediately as processes arrive.
- **Time Slot 5 - 6 (Preemption and Context Switching):**
  - *Preemption:* At slot 5, Process 1 (on CPU 1) exhausts its time slice and is put back into the run queue. CPU 1 immediately dispatches the newly loaded Process 4 (Prio 0).
  - *Process Migration:* At slot 6, Process 2 (on CPU 0) is preempted. CPU 0 then picks up Process 1 (which was previously on CPU 1). This clearly demonstrates *process migration* in the SMP environment.
- **Time Slot 9 - 13 (Re-dispatching):**

- *Round-Robin Behavior:* Processes 4 and 1 continue to execute. At slot 9 and 13, CPU 1 puts Process 4 back to the queue but immediately re-dispatches it (likely because it has the highest priority/quota at that moment). Similarly, CPU 0 re-dispatches Process 1 at slot 10.

- **Time Slot 14 - 16 (Process Completion):**
  - *Completion:* Process 1 finishes execution on CPU 0 at slot 14. CPU 0 then switches to Process 3 (Prio 1).
  - *Completion:* Process 4 finishes execution on CPU 1 at slot 16. CPU 1 then switches to Process 2 (Prio 1), which had been waiting since slot 6.

- **Time Slot 22 - 26 (Finalization):**
  - *Termination:* Process 2 finishes at slot 22, and CPU 1 stops as there are no more processes for it. Process 3 continues on CPU 0 until slot 26, where it finishes, and CPU 0 finally stops.

3.1.2 Scheduling: Gantt chart

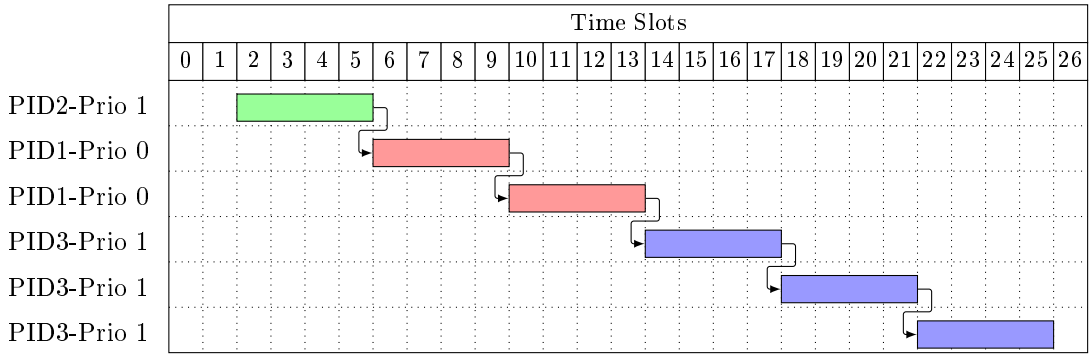

Figure 4: Gantt Chart for CPU 0

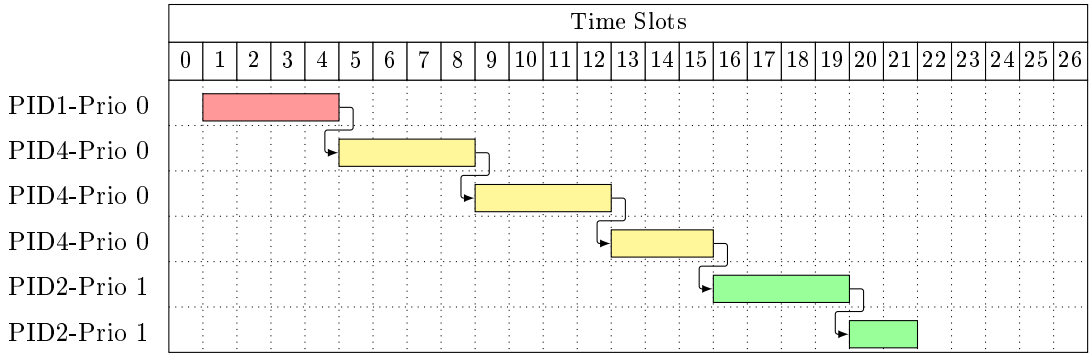

Figure 5: Gantt Chart for CPU 1

3.2 Memory

3.2.1 Result of the input file of Memory (os_0_mlq_paging)

```
1     Time slot   0
2 ld_routine
```

```
    Loaded a process at input/proc/p0s, PID: 1 PRIO: 0
Time slot   1
    CPU 1: Dispatched process  1
    Loaded a process at input/proc/p1s, PID: 2 PRIO: 15
Time slot   2
    CPU 0: Dispatched process  2
liballoc:152
--- Page Table Dump ---
print_pgtbl:
 PGD=0x739974001000 P4D=0x739974003000 PUD=0x739974005000 PMD=0x739974007000 PT=0
     x739974009000
0000000000000000: 0000000000000001
Time slot   3
    Loaded a process at input/proc/p1s, PID: 3 PRIO: 0
liballoc:152
--- Page Table Dump ---
print_pgtbl:
 PGD=0x739974001000 P4D=0x739974003000 PUD=0x739974005000 PMD=0x739974007000 PT=0
     x739974009000
0000000000000000: 0000000000000001
Time slot   4
libfree:168
--- Page Table Dump ---
print_pgtbl:
 PGD=0x739974001000 P4D=0x739974003000 PUD=0x739974005000 PMD=0x739974007000 PT=0
     x739974009000
0000000000000000: 0000000000000001
Time slot   5
liballoc:152
--- Page Table Dump ---
print_pgtbl:
 PGD=0x739974001000 P4D=0x739974003000 PUD=0x739974005000 PMD=0x739974007000 PT=0
     x739974009000
0000000000000000: 0000000000000001
Time slot   6
    Loaded a process at input/proc/p0s, PID: 4 PRIO: 0
libwrite:343
--- Page Table Dump ---
print_pgtbl:
 PGD=0x739974001000 P4D=0x739974003000 PUD=0x739974005000 PMD=0x739974007000 PT=0
     x739974009000
0000000000000000: 0000000000000001
Time slot   7
    CPU 1: Put process  1 to run queue
```

```
    CPU 1: Dispatched process  3
Time slot   8
    CPU 0: Put process  2 to run queue
    CPU 0: Dispatched process  4
Time slot   9
liballoc:152
--- Page Table Dump ---
print_pgtbl:
 PGD=0x73996c001000 P4D=0x73996c003000 PUD=0x73996c005000 PMD=0x73996c007000 PT=0
     x73996c009000
0000000000000000: 0000000000001001
Time slot  10
liballoc:152
--- Page Table Dump ---
print_pgtbl:
 PGD=0x73996c001000 P4D=0x73996c003000 PUD=0x73996c005000 PMD=0x73996c007000 PT=0
     x73996c009000
0000000000000000: 0000000000001001
Time slot  11
libfree:168
--- Page Table Dump ---
print_pgtbl:
 PGD=0x73996c001000 P4D=0x73996c003000 PUD=0x73996c005000 PMD=0x73996c007000 PT=0
     x73996c009000
0000000000000000: 0000000000001001
Time slot  12
liballoc:152
--- Page Table Dump ---
print_pgtbl:
 PGD=0x73996c001000 P4D=0x73996c003000 PUD=0x73996c005000 PMD=0x73996c007000 PT=0
     x73996c009000
0000000000000000: 0000000000001001
Time slot  13
libwrite:343
--- Page Table Dump ---
print_pgtbl:
 PGD=0x73996c001000 P4D=0x73996c003000 PUD=0x73996c005000 PMD=0x73996c007000 PT=0
     x73996c009000
0000000000000000: 0000000000001001
    CPU 1: Put process  3 to run queue
    CPU 1: Dispatched process  1
libread:304
Time slot  14
    CPU 0: Put process  4 to run queue
```

```
    CPU 0: Dispatched process  3
Time slot  15
Time slot  16
Time slot  17
Time slot  18
    CPU 0: Processed  3 has finished
    CPU 0: Dispatched process  4
libread:304
Time slot  19
    CPU 1: Put process  1 to run queue
    CPU 1: Dispatched process  1
libfree:168
--- Page Table Dump ---
print_pgtbl:
 PGD=0x739974001000 P4D=0x739974003000 PUD=0x739974005000 PMD=0x739974007000 PT=0
     x739974009000
0000000000000000: 0000000000000001
Time slot  20
Time slot  21
    CPU 1: Processed  1 has finished
    CPU 1: Dispatched process  2
Time slot  22
Time slot  23
Time slot  24
    CPU 0: Put process  4 to run queue
    CPU 0: Dispatched process  4
libfree:168
--- Page Table Dump ---
print_pgtbl:
 PGD=0x73996c001000 P4D=0x73996c003000 PUD=0x73996c005000 PMD=0x73996c007000 PT=0
     x73996c009000
0000000000000000: 0000000000001001
Time slot  25
    CPU 1: Processed  2 has finished
    CPU 1 stopped
Time slot  26
    CPU 0: Processed  4 has finished
    CPU 0 stopped
```

3.2.2 Explain the status of the memory allocation in heap and data segments

- **Heap Segment:** It is the memory area for initialization (Dynamic Allocation) while the program is running. It has no fixed size and can be expanded.

- **Data Segment:** In this simplified simulation model, the "Data Segment" refers to the actual byte values written into the allocated Heap memory frames. While the Heap defines the "container" (virtual addresses and size), the Data represents the "content" stored in the corresponding Physical Frames (RAM).

3.2.3 Chronological Analysis of Memory Allocation

The following analysis tracks the state of memory allocation across specific time slots where memory intervention occurs. Two distinct processes are identified by their Page Global Directory (PGD) addresses:

- **Process A (p0s):** PGD Address `0x739974001000`
- **Process B (p1s):** PGD Address `0x73996c001000`

Start Explain the status of the memory allocation in heap and data segments.

- **Time Slot 2: Initial Allocation (Process A)**
  - **Event:** `liballoc:152` triggered by Process A.
  - **Action:** The process requests memory allocation. The OS expands the Heap and creates a new virtual memory region starting at Virtual Address 0.
  - **Page Table Dump:** `00...00: 00...01`
  - **Analysis:** The Page Table Entry (PTE) value `0x1` indicates:
    - **Present Bit (Bit 0):** 1 (Page is in RAM).
    - **Frame Page Number (FPN):** 0 (Derived from the upper bits).

  Process A is assigned **Physical Frame 0**.

- **Time Slot 4: Logical Deallocation (Process A)**
  - **Event:** `libfree:168`.
  - **Action:** Process A calls `free`.
  - **Heap Status:** The memory region is logically marked as "free" and added to the `vm_freerg_list` for future reuse.

- **Time Slot 5: Re-allocation (Process A)**
  - **Event:** `liballoc:152`.
  - **Action:** Process A requests allocation again.
  - **Heap Status:** The OS checks the `vm_freerg_list`, finds the region freed in Time Slot 4, and reuses it.
  - **Mapping:** The mapping remains `00...01` (Physical Frame 0).

- **Time Slot 6: Data Write (Process A)**
  - **Event:** `libwrite:343`.
  - **Action:** Data is written to the allocated address.

- **Data Status:** The actual value is stored in **Physical Frame 0**.

- **Time Slot 9: Allocation for New Process (Process B)**
  - **Event:** `liballoc:152`.
  - **Context Switch:** The PGD changes to `0x73996c001000`, indicating a context switch to Process B.
  - **Page Table Dump:** `00...00: 00...1001`
  - **Analysis:** The PTE value `0x1001` indicates:
    * **Present Bit:** 1.
    * **FPN:** 1 (The value `0x1000` at bit offset 12 corresponds to index 1).

    Since Frame 0 is occupied by Process A, Process B is assigned the next available resource, **Physical Frame 1**.

- **Time Slot 11 & 13: Free and Write (Process B)**
  - **Time Slot 11 (Free):** Process B frees its memory region. The region enters Process B's private free list.
  - **Time Slot 13 (Write):** Process B writes data. This data is stored in **Physical Frame 1**.

- **Time Slot 19 & 24: Final Deallocation** Both processes perform their final `libfree` operations (Slot 19 for Process A, Slot 24 for Process B), releasing their respective logical memory regions.

3.2.4 Summary of Result Flow

The table below summarizes the correlation between Time Slots, Processes, and Physical Memory status.

Slot	PGD (Process)	Event	Physical Frame	Logical Heap Status
2	...7400 (Proc A)	Alloc	Frame 0	New Region created
4	...7400 (Proc A)	Free	Frame 0	Region moved to Free List
5	...7400 (Proc A)	Alloc	Frame 0	Region reused from Free List
6	...7400 (Proc A)	Write	Frame 0	Data written to Frame 0
9	...6c00 (Proc B)	Alloc	**Frame 1**	New Region created
11	...6c00 (Proc B)	Free	Frame 1	Region moved to Free List
13	...6c00 (Proc B)	Write	Frame 1	Data written to Frame 1

Table 1: Memory Allocation Flow Summary

4 Answer question

4.1 Question 1

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

Answer:

The mechanism used to pass a complex argument to a system call is Indirection via pointers. Since CPU registers are very limited in number and size, they cannot hold large or complex data directly. Instead, the user-space program places the data in its own memory and passes only the memory address (pointer) of that data in a register. When the system call is invoked, the kernel receive the pointer from the register. But this method have a problem that the passed address maybe wrong or bad address (pointing to kernel memory or unmapped memory) leading to the kernel could crash. To handle this problem, the kernel check if the address provided is actually within the User space memory area. Then the kernel copy the data from User space into Kernel Space buffer, the Kernel perform on its own safe copy of the data.

4.2 Question 2

Question: Which mechanisms does the operating system use to manage system calls that become unresponsive?

Answer:

Operating systems use a multi-layered approach to manage system calls that become unresponsive. The mechanisms used to manage this fall into three categories:

- Interruption Mechanisms: when the system call blocks, the operating system provides ways for the user or the application to forcefully break. In Linux most of blocking system calls place the process in a task interruptible state, when the process receives a signal like Ctrl + C, the kernel prematurely wakes the process.
- Timeout/Watchdogs: The kernel monitors itself to ensure that a single stuck system call does not freeze the entire computer. Many blocking system calls accept a timeout parameter. The kernel sets an internal timer; if the condition is not satisfied before the timer expires, the call returns with an error. In addition, the operating system runs a background kernel thread that scans all process in every constant time (120 seconds in Linux), if it find a process stuck in Uninterruptible Sleep (D state) for more than this duration, it logs a "Hung Task" error to the kernel log with a stack trace.
- Asynchronous Avoidance: Modern operating systems encourage developers to avoid "blocking" system calls entirely to prevent unresponsiveness. Applications can set file descriptors to non-blocking mode (O_NONBLOCK). Its mean that if a system call would freeze, it returns immediately with an error instead of waiting. In addition, there are some API (Application Programming Interface) like io_uring, epoll,.. in Linux that allow applications to submit system calls to a queue and retrieve results later, ensuring the main application thread never actually block.

4.3 Question 3

Question: Evaluate the comparative advantages of the scheduling algorithm implemented in this assignment in relation to other scheduling algorithms you have learned. What is the complexity of this algorithm?

Answer:

The scheduling algorithm implemented in this assignment is a **Non-Feedback Multilevel Queue (MLQ)** augmented with a **Slot-based Round Robin** mechanism for inter-queue dispatching. The design employs a static array of priority queues, where each priority level is assigned a specific execution quota (slot). Based on the theoretical framework provided in *Operating System Concepts (10th Edition)* and the specific logic in `sched.c`, we present a detailed comparative evaluation and complexity analysis below.

1. **Comparative Evaluation against Standard Algorithms**

   a. **Comparison with First-Come, First-Served (FCFS)**

   - **Theoretical Background:** FCFS is the most basic non-preemptive scheduling algorithm. The CPU is allocated to the process at the head of the ready queue. The critical drawback identified in operating system theory is the *"Convoy Effect"*. This phenomenon occurs when a CPU-intensive process occupies the processor for an extended period, causing all subsequent I/O-bound processes to wait in the ready queue. This leads to extremely low I/O device utilization and a high Average Waiting Time.
   - **Assignment Implementation Analysis:** The implemented algorithm in the assignment is inherently **Preemptive**.
     - *Mechanism:* The simulation framework in `os.c` enforces a `time_slot`. Even if a process has not completed its CPU burst, it is forced to yield the CPU and is placed back into the `mlq_ready_queue` via `put_mlq_proc()`.
     - *Code Evidence:* In `sched.c`, the scheduler manages `slot[i]`. When a process consumes its time slice, the scheduler is invoked again to pick a new candidate.
   - **Comparative Advantage:** Our Slot-based MLQ algorithm completely eliminates the Convoy Effect. By forcing time-sharing (Round Robin within queues), short processes and I/O-bound processes are guaranteed frequent CPU access. This significantly improves the system's **Responsiveness**, making it suitable for interactive environments, whereas FCFS is strictly limited to Batch Systems.

   b. **Comparison with Shortest-Job-First (SJF)**

   - **Theoretical Background:** SJF is theoretically optimal; it is proven to yield the minimum possible average waiting time for a given set of processes. Ideally, the scheduler selects the process with the shortest next CPU burst. However, the textbook highlights a fundamental implementation flaw: *Infeasibility*. In a real Short-Term Scheduler, the OS cannot know the length of the next CPU burst. Implementing SJF requires relying on exponential averaging ($\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$) to predict the future, which adds computational overhead and is not always accurate.

- **Assignment Implementation Analysis:** Our implementation prioritizes **Determinism** and **Feasibility** over theoretical optimality.
  - *Mechanism:* Instead of guessing the future, the system uses a static priority value (`proc->prio`) defined in the `pcb_t` structure.
  - *Code Evidence:* The `get_mlq_proc()` function iterates through a fixed array `mlq_ready_queue[i]`. The decision is based purely on the current state (priority and slot), requiring no complex mathematical prediction.
- **Comparative Evaluation:** While our algorithm may not achieve the theoretically minimum waiting time of SJF, it is practically implementable with $O(1)$ decision overhead (relative to process count). It avoids the runtime cost of predicting CPU bursts, making the kernel faster and more predictable.

c. Comparison with Pure Priority Scheduling

- **Theoretical Background:** In standard Priority Scheduling, the CPU is always allocated to the highest priority process. The fatal flaw of this design is *"Indefinite Blocking"* or *"Starvation"*. If a steady stream of high-priority processes arrives, low-priority processes may never execute. The standard solution is "Aging" (gradually increasing the priority of waiting processes), which adds complexity to the system.
- **Assignment Implementation Analysis (The Slot Mechanism):** This is the most innovative aspect of the assignment's design. It solves starvation **without** changing process priorities (Aging).
  - *Code Logic:*

```
// sched.c : get_mlq_proc()
if (slot[i] > 0) {
    proc = dequeue(...);
    slot[i]--; // Consume budget
    break;     // Dispatch immediate
} else {
    slot[i] = MAX_PRIO - i; // Refill budget
    continue; // SKIP this queue, force check lower priority
}
```

  - *Operational Theory:* The system assigns a "CPU Budget" to each priority level based on the formula $Budget = MAX\_PRIO - Prio$.
  - *Example:* Priority 0 gets 140 slots. Priority 1 gets 139 slots. Even if Priority 0 has infinite tasks, after 140 dispatches, the `slot[0]` becomes 0. The scheduler is mathematically forced to skip Priority 0 and serve Priority 1.
- **Comparative Advantage:** The implementation achieves **Starvation Freedom** through a "Share-based" approach. It guarantees that every priority level, no matter how low, eventually receives a turn when higher levels exhaust their quotas. This is a robust and fairness-oriented improvement over standard Priority Scheduling.

d. Comparison with Standard Multilevel Queue (Partitioned)

- **Theoretical Background:** Standard MLQ partitions the ready queue into distinct groups, typically "Foreground" (Interactive) and "Background" (Batch). The scheduling between queues is often Fixed-Priority (leading to starvation) or Time-Slicing with coarse granularity (e.g., 80% CPU for Foreground, 20% for Background).
- **Assignment Implementation Analysis:** Our implementation offers a **Fine-Grained Proportional Share**.
  - *Granularity:* Instead of just 2 or 3 partitions, the system supports `MAX_PRIO` (140) distinct levels.
  - *Distribution:* The slot allocation decreases linearly $(140, 139, \ldots, 1)$.
- **Comparative Advantage:** This linear distribution creates a smoother degradation of service. It allows the OS to differentiate processes with high precision (e.g., a process with priority 10 is slightly favored over 11), providing more flexibility than the rigid Foreground/Background binary of standard MLQ.

e. Comparison with Multilevel Feedback Queue (MLFQ)

- **Theoretical Background:** MLFQ is considered the most general and powerful scheduling algorithm. Its key feature is *Feedback*: processes move between queues. A process using too much CPU moves down; a process waiting too long moves up (Aging). This allows the OS to learn the nature of the process (I/O-bound vs CPU-bound) dynamically.
- **Assignment Implementation Limitation:** The assignment specification explicitly states a non-feedback design.
  - *Code Evidence:* In `put_mlq_proc()`, a process is always enqueued back to `mlq_ready_queue[proc->prio]`. The `prio` field is never modified by the scheduler.
- **Disadvantage/Trade-off:** The lack of feedback makes the system less adaptive. If a high-priority process becomes CPU-intensive, it continues to enjoy a large slot quota (140 slots), potentially degrading overall system throughput compared to an MLFQ which would demote such a process. However, this simplifies the implementation significantly and reduces the overhead of priority recalculation.

2. Algorithmic Complexity Analysis

The performance of the scheduler is determined by the operations on the `queue_t` structure (array-based) and the search logic in `sched.c`. Let N be the number of processes in a specific queue, and K be the number of priority levels ($K = \texttt{MAX\_PRIO} = 140$).

- **Enqueue Operation ($O(1)$):**
  - *Code Reference:* `queue.c: enqueue()`.
  - *Analysis:* The function inserts a process at the tail of the array: `q->proc[q->size] = proc`. This requires direct index access and incrementing the size counter.
  - *Complexity:* Constant time, $O(1)$.
- **Dequeue Operation ($O(N)$):**
  - *Code Reference:* `queue.c: dequeue()`.

- *Analysis:* The scheduler retrieves the process at the head (index 0). Because the queue is implemented as a contiguous memory block (array), removing the first element creates a "gap". The loop `for (i = 0; i < q->size - 1; i++)` is required to shift all remaining $N-1$ elements to the left to preserve data contiguity.
- *Complexity:* Linear time relative to queue size, $O(N)$.

- **Scheduler Selection Logic ($O(K+N)$):**
  - *Code Reference:* `sched.c: get_mlq_proc()`.
  - *Analysis:* The routine employs a linear search across priority levels: `for (int i = 0; i < MAX_PRIO; i++)`. In the worst-case scenario (e.g., when the system is idle or all high-priority slots are exhausted), the loop iterates K times. Upon finding a valid queue, it calls `dequeue()`.
  - *Calculation:* $Cost = Search\_Cost + Retrieval\_Cost = O(K) + O(N)$.
  - **Total Complexity:** $O(K+N)$.

4.4 Question 4

Question: Discuss the architectural and operational implications of implementing multiple memory segments in this simple OS.

Answer:

Implementing multiple memory segments fundamentally structures how the OS manages the virtual address space on top of the physical paging system. The implications can be divided into architectural changes and operational behaviors.

1. **Architectural Implications**

   From an architectural perspective, this adds a logical organization layer above the raw paging mechanism:

   a. **Data Structure Transformation:** In the design, a segment is represented by a `vm_area_struct` inside the per-process `mm_struct`. Implementing multiple segments means that instead of a single VMA (current `vm_id = 0` for "data/heap"), `mm->mmap` becomes a linked list of VMAs. For example:
   - `vm_id = 0`: Code / Read-only data
   - `vm_id = 1`: Heap
   - `vm_id = 2`: Stack
   - `vm_id = 3`: Shared regions, etc.

   b. **Logical Abstraction Layer:** Architecturally, this creates a logical layer on top of the 5-level page tables:
   - The **5-level paging** (PGD–P4D–PUD–PMD–PT) remains unchanged and is solely responsible for translating virtual page numbers to physical frames.
   - The **Segmentation layer** decides *which* virtual ranges exist and *what* they are used for. Each VMA tracks its virtual interval `[vm_start, vm_end)`, current growth point `sbrk`, and a free-region list for local holes.

c. **Structured Extensibility:** This makes the address space structured and extensible. Different regions can later be assigned different protection policies (e.g., code as read-only, stack as non-executable, or shared segments mapped into multiple processes) while still sharing the same page table root (`mm->pgd`).

2. **Operational Implications**

On the operational side, supporting multiple segments changes the workflow of memory operations:

a. **Segment-Aware Operations:** All core memory functions (`__alloc`, `__free`, `__read`, `__write`) become segment-aware. They take a `vmaid` parameter to select the correct VMA via `get_vma_by_num(mm, vmaid)` before performing any work. The allocator only searches and updates the free-region list of that specific segment, and `inc_vma_limit` grows only that targeted VMA when more virtual space is needed.

b. **Bookkeeping vs. Isolation:** There is a trade-off involving increased bookkeeping for better isolation. Every time the kernel extends a segment, it must check for overlaps with other VMAs (`validate_overlap_vm_area`) before calling `vm_map_ram` to map new pages. While this adds overhead (walking the VMA list, extra validation), it guarantees that distinct segments (e.g., Heap vs. Stack) do not collide and grow independently.

c. **Contextual Paging and Swapping:** While low-level page-table operations in `mm64.c` (allocating frames, filling PTEs, swapping victim pages) remain structurally the same, the context of each page becomes clearer. The OS knows whether a page belongs to code, heap, or stack based on the VMA it falls into. This facilitates segment-specific policies (e.g., preventing code pages from being swapped out or prioritizing stack pages) without altering the core paging algorithms.

In summary, while multiple segments slightly increase complexity in the memory manager, the OS gains a much more organized virtual address space and a natural architectural hook to implement protection and per-segment policies on top of the underlying 5-level paging mechanism.

4.5 Question 5

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

Answer:

Based on the Simple OS implementation, dividing the address into more than 2 levels fundamentally changes the paging structure and introduces several important consequences:

1. **Current 2-Level Implementation**

In the Simple OS's 2-level paging system (defined in `common.h`), the 20-bit address space is divided as follows:

- **First level (Segment):** 5 bits (`FIRST_LV_LEN`) = 32 entries

- **Second level (Page):** 5 bits (`SECOND_LV_LEN`) = 32 entries
- **Offset:** 10 bits (`OFFSET_LEN`) = 1KB pages

This creates a hierarchical structure with `page_table_t` containing pointers to `trans_table_t` entries, requiring 2 memory accesses for address translation. The total virtual address space is limited to 1 MB (2^{20} bytes).

```
// common.h
/* Address geometry (two-level page tables):
* ADDRESS_SIZE: total VA bits.
* OFFSET_LEN:   bits within a page (page size = 1 << OFFSET_LEN).
* FIRST_LV_LEN: bits used by first-level index (L1).
* SECOND_LV_LEN:bits used by second-level index (L2).
* SEGMENT_LEN/PAGE_LEN: aliases to L1/L2 lengths.
*/
#define ADDRESS_SIZE 20
#define OFFSET_LEN 10
#define FIRST_LV_LEN 5
#define SECOND_LV_LEN 5
#define SEGMENT_LEN FIRST_LV_LEN
#define PAGE_LEN SECOND_LV_LEN
```

Listing 14: 20-Bit Address Space

2. 5-Level Paging (64-bit Implementation)

When moving to 5-level paging in the 64-bit mode (`mm64.h` and `mm64.c`), the address is divided into 5 distinct levels:

- **Level 5 - PGD (Page Global Directory):** bits 56-48 (9 bits = 512 entries)
- **Level 4 - P4D (Page Level 4 Directory):** bits 47-39 (9 bits = 512 entries)
- **Level 3 - PUD (Page Upper Directory):** bits 38-30 (9 bits = 512 entries)
- **Level 2 - PMD (Page Middle Directory):** bits 29-21 (9 bits = 512 entries)
- **Level 1 - PT (Page Table):** bits 20-12 (9 bits = 512 entries)
- **Offset:** bits 11-0 (12 bits = 4KB pages)

This expands the virtual address space from 1 MB (20-bit) to 128 PiB (57-bit canonical address space), as implemented in `mm64.c`.

```
// mm64.h
#define MM64_BITS_PER_LONG 64

#define PAGING64_CPU_BUS_WIDTH 57 /* 57-bit canonical VA space */
#define PAGING64_PAGESZ 4096      /* 4KB or 12-bits PAGE NUMBER */

// ...
```

```
8
9  /* OFFSET */
10 #define PAGING64_ADDR_OFFST_HIBIT 11
11 #define PAGING64_ADDR_OFFST_LOBIT 0
12
13 /* PT */
14 #define PAGING64_ADDR_PT_HIBIT 20
15 #define PAGING64_ADDR_PT_LOBIT 12
16 #define PAGING64_ADDR_PT_SHIFT 12
17
18 /* PMD */
19 #define PAGING64_ADDR_PMD_HIBIT 29
20 #define PAGING64_ADDR_PMD_LOBIT 21
21
22 /* PUD */
23 #define PAGING64_ADDR_PUD_HIBIT 38
24 #define PAGING64_ADDR_PUD_LOBIT 30
25
26 /* P4D */
27 #define PAGING64_ADDR_P4D_HIBIT 47
28 #define PAGING64_ADDR_P4D_LOBIT 39
29
30 /* PGD */
31 #define PAGING64_ADDR_PGD_HIBIT 56
32 #define PAGING64_ADDR_PGD_LOBIT 48
```

Listing 15: 64-Bit 5 Level Paging

3. Key Consequences of Multi-Level Paging

a. Address Space Expansion: The 5-level scheme supports up to 128 petabytes of virtual memory, compared to the 2-level system's 1 MB limit. The function `get_pd_from_address()` in `mm64.c` (lines 149-157) extracts all 5 directory indices from a single address using bit masks and shifts, enabling this massive expansion. Each level uses 9 bits, allowing 512 entries per directory table.

```
1  // mm64.c
2  /*
3  * get_pd_from_address - Parse address to 5 page directory level
4  */
5  int get_pd_from_address(addr_t addr, addr_t *pgd, addr_t *p4d, addr_t *pud,
       addr_t *pmd, addr_t *pt)
6  {
7      *pgd = (addr & PAGING64_ADDR_PGD_MASK) >> PAGING64_ADDR_PGD_LOBIT;
8      *p4d = (addr & PAGING64_ADDR_P4D_MASK) >> PAGING64_ADDR_P4D_LOBIT;
9      *pud = (addr & PAGING64_ADDR_PUD_MASK) >> PAGING64_ADDR_PUD_LOBIT;
10     *pmd = (addr & PAGING64_ADDR_PMD_MASK) >> PAGING64_ADDR_PMD_LOBIT;
```

```
    *pt = (addr & PAGING64_ADDR_PT_MASK) >> PAGING64_ADDR_PT_LOBIT;
    return 0;
}
```

Listing 16: get_pd_from_address() in mm64.c

b. Increased Translation Overhead: Each address translation requires traversing all 5 levels (PGD → P4D → PUD → PMD → PT), requiring 5 memory accesses instead of 2. The implementation in `get_page_table_entry()` (Listing 12) demonstrates this hierarchical traversal. Modern CPUs use Translation Lookaside Buffers (TLB) to cache recent translations, mitigating this overhead in practice.

c. Sparse Allocation and Memory Efficiency: The implementation uses **on-demand allocation** through the `get_next_level()` function (lines 72-96 of `mm64.c`). Page tables are allocated only when actually needed, not for the entire address space. This sparse allocation strategy is critical because:

- Theoretical maximum page table size would be $512^5 = 35$ trillion entries (unfeasible)
- Actual memory usage is much smaller since only used address ranges allocate tables
- Each table is 4KB (512 entries × 8 bytes), allocated via `alloc_aligned_table()`
- The `mm_struct->pgd` pointer starts as NULL and is allocated on first use

```
// mm64.c
/*
* Helper: Get or Allocate Next Level (Directory Navigation)
*/
void *get_next_level(uint64_t *entry, int alloc)
{
    /* Check Real Hardware Present Bit (Bit 0) */
    if (*entry & X86_PRESENT)
    {
        /* Mask off the bottom 12 bits to get the pointer */
        return (void *)(*entry & X86_ADDR_MASK);
    }

    if (!alloc)
    return NULL;

    /* Allocate 4KB Aligned Table */
    void *new_table = alloc_aligned_table();
    if (new_table == NULL)
    return NULL;

    /* * Write the Entry:
    * Pointer (Already 4KB aligned, so bottom 12 bits are 0)
    * OR with X86 Flags (Present + Write + User)
```

```
25      */
26      *entry = (uint64_t)new_table | X86_PRESENT | X86_WRITE | X86_USER;
27
28      return new_table;
29  }
```

Listing 17: get_next_level() in mm64.c

d. Page Table Entry Operations Complexity: All PTE operations must traverse the full 5-level hierarchy:

- `pte_set_fpn()` (lines 192-208): Walks 5 levels to set frame number

```
1   // mm64.c
2   /*
3   * pte_set_fpn - Set PTE entry for on-line page
4   */
5   int pte_set_fpn(struct pcb_t *caller, addr_t pgn, addr_t fpn)
6   {
7       addr_t addr = pgn << PAGING64_ADDR_PT_SHIFT;
8       uint64_t *pte = get_page_table_entry(caller->mm, addr, 1);
9       if (pte == NULL)
10      return -1;
11
12      *pte = 0; // Clear old data
13
14      // Mark page present in RAM
15      SETBIT(*pte, PAGING_PTE_PRESENT_MASK);
16      CLRBIT(*pte, PAGING_PTE_SWAPPED_MASK);
17
18      SETVAL(*pte, fpn, PAGING_PTE_FPN_MASK, PAGING_PTE_FPN_LOBIT);
19
20      return 0;
21  }
```

Listing 18: pte_set_fpn() in mm64.c

- `pte_set_swap()` (lines 171-187): Walks 5 levels to set swap information

```
1   // mm64.c
2   /*
3   * pte_set_swap - Set PTE entry for swapped page
4   */
5   int pte_set_swap(struct pcb_t *caller, addr_t pgn, int swptyp, addr_t
        swpoff)
6   {
7       addr_t addr = pgn << PAGING64_ADDR_PT_SHIFT;
8       uint64_t *pte = get_page_table_entry(caller->mm, addr, 1);
```

```
    if (pte == NULL)
    return -1;

    *pte = 0; // Clear old data

    // Swapped pages keep Present=0 and only carry swap metadata
    SETBIT(*pte, PAGING_PTE_SWAPPED_MASK);

    SETVAL(*pte, swptyp, PAGING_PTE_SWPTYP_MASK, PAGING_PTE_SWPTYP_LOBIT)
        ;
    SETVAL(*pte, swpoff, PAGING_PTE_SWPOFF_MASK, PAGING_PTE_SWPOFF_LOBIT)
        ;

    return 0;
}
```

Listing 19: pte_set_swap() in mm64.c

- `pte_get_entry()` (lines 211-218): Walks 5 levels to read PTE

```
// mm64.c
/* Get PTE page table entry */
uint64_t pte_get_entry(struct pcb_t *caller, addr_t pgn)
{
    addr_t addr = pgn << PAGING64_ADDR_PT_SHIFT;
    uint64_t *pte = get_page_table_entry(caller->mm, addr, 0);
    if (pte == NULL)
    return 0;
    return *pte;
}
```

Listing 20: pte_get_entry() in mm64.c

- `vmap_pgd_memset()` (lines 234-246): Must walk 5 levels for each page mapped

```
// mm64.c
/*
* vmap_pgd_memset - map a range of page at aligned address
*/
int vmap_pgd_memset(struct pcb_t *caller, addr_t addr, int pgnum)
{
    for (int i = 0; i < pgnum; i++)
    {
        addr_t current_addr = addr + (i * PAGING64_PAGESZ);
        uint64_t *pte = get_page_table_entry(caller->mm, current_addr, 1)
            ;
        if (pte)
```

```
12          {
13              *pte = 0; // Clear the LEAF entry (Not Present)
14          }
15      }
16      return 0;
17  }
```

Listing 21: vmap_pgd_memset() in mm64.c

Each operation calls `get_page_table_entry()` which performs the full traversal, allocating intermediate directories as needed.

e. **Implementation Complexity:** The 5-level implementation is significantly more complex than the 2-level system:

- `init_mm()` (lines 358-386 in mm64.c): Initializes the root PGD pointer as NULL, with VM area structures

```
1   // mm64.c
2   /* init_mm */
3   int init_mm(struct mm_struct *mm, struct pcb_t *caller)
4   {
5       struct vm_area_struct *vma0 = malloc(sizeof(struct vm_area_struct));
6       mm->pgd = NULL;
7
8       vma0->vm_id = 0;
9       vma0->vm_start = 0;
10      vma0->vm_end = vma0->vm_start;
11      vma0->sbrk = vma0->vm_start;
12      vma0->vm_freerg_list = NULL;
13
14      struct vm_rg_struct *first_rg = init_vm_rg(vma0->vm_start, vma0->
            vm_end);
15      enlist_vm_rg_node(&vma0->vm_freerg_list, first_rg);
16
17      vma0->vm_next = NULL;
18      vma0->vm_mm = mm;
19      mm->mmap = vma0;
20
21      for (int i = 0; i < PAGING_MAX_SYMTBL_SZ; i++)
22      {
23          mm->symrgtbl[i].rg_start = 0;
24          mm->symrgtbl[i].rg_end = 0;
25          mm->symrgtbl[i].rg_next = NULL;
26      }
27
28      mm->fifo_pgn = NULL;
```

```
29
30      return 0;
31  }
```

Listing 22: init_mm() in mm64.c

- `get_page_table_entry()`: Implements recursive tree traversal with dynamic allocation (Listing 12)
- `print_pgtbl_recursive()` (lines 546-595 in mm64.c): Recursive function to traverse and print all 5 levels
- Address extraction in `get_pd_from_address()`: Complex bit manipulation across 5 levels (Listing 16)

The code structure shows separate directory types (`pgd_t`, `p4d_t`, `pud_t`, `pmd_t`, `pt_t`) defined in `os-mm.h`, each containing 512 entries.

```
1  // os-mm.h
2  #define MM_PAGING
3  #define PAGING_MAX_MMSWP 4 /* max number of supported swapped space */
4  #define PAGING_MAX_SYMTBL_SZ 30
5
6  /* 64-bit Paging Constants */
7  #define PAGING64_ENTRY_COUNT 512 // 9 bits per level = 512 entries
8
9  /*
10 * @bksysnet: in long address mode of 64bit or original 32bit
11 * the address type need to be redefined
12 */
13
14 #ifdef MM64
15 typedef uint64_t addr_t;
16 typedef char BYTE;
17
18 // Page Table (Level 1) - Points to Physical Frame
19 struct pt_t
20 {
21     uint64_t entries[PAGING64_ENTRY_COUNT];
22 };
23
24 // Page Middle Directory (Level 2) - Points to PT
25 struct pmd_t
26 {
27     uint64_t entries[PAGING64_ENTRY_COUNT];
28 };
29
30 // Page Upper Directory (Level 3) - Points to PMD
```

```
31 struct pud_t
32 {
33     uint64_t entries[PAGING64_ENTRY_COUNT];
34 };
35
36 // Page Level-4 Directory (Level 4) - Points to PUD
37 struct p4d_t
38 {
39     uint64_t entries[PAGING64_ENTRY_COUNT];
40 };
41
42 // Page Global Directory (Level 5) - Points to P4D
43 struct pgd_t
44 {
45     uint64_t entries[PAGING64_ENTRY_COUNT];
46 };
```

Listing 23: Directory Structs defined in os-mm.h

f. Storage Space Trade-offs: The implementation demonstrates careful design decisions to balance:

- **Traversal time:** 5 memory accesses per translation (mitigated by TLB)
- **Memory accesses:** Each level requires one access, but sparse allocation reduces total accesses
- **Storage space:** Only allocated tables consume memory, not theoretical maximum

The `vmap_page_range()` function (lines 251-278 of mm64.c) shows how pages are mapped efficiently, tracking mapped pages in the `fifo_pgn` list for replacement algorithms.

```
1  /*
2  * vmap_page_range - map a range of page at aligned address
3  */
4  addr_t vmap_page_range(struct pcb_t *caller,
5  addr_t addr,
6  int pgnum,
7  struct framephy_struct *frames,
8  struct vm_rg_struct *ret_rg)
9  {
10     struct framephy_struct *fpit = frames;
11     int pgit = 0;
12     addr_t pgn;
13
14     ret_rg->rg_start = addr;
15     ret_rg->rg_end = addr + (pgnum * PAGING64_PAGESZ);
16
17     for (pgit = 0; pgit < pgnum; pgit++)
18     {
19         if (fpit == NULL)
```

```
            return -1;

            pgn = (addr + (pgit * PAGING64_PAGESZ)) >> PAGING64_ADDR_PT_SHIFT;

            pte_set_fpn(caller, pgn, fpit->fpn);
            enlist_pgn_node(&caller->mm->fifo_pgn, pgn);

            fpit = fpit->fp_next;
        }

        return 0;
}
```

Listing 24: vmap_page_range() in os-mm.h

4. **Design Strategies Used:**

   The implementation employs several optimization strategies to mitigate the overhead of 5-level paging:

   - **Demand Allocation:** Tables allocated only when first accessed (`get_next_level()` with `alloc=1`)
   - **Sparse Page Tables:** No allocation for unused address ranges
   - **4KB Alignment:** Efficient memory allocation using `posix_memalign()` for hardware compatibility
   - **Present Bit Checking:** Fast detection of absent entries using `PAGING_PRESENT` bit (bit 0)
   - **FIFO Page Tracking:** Efficient page replacement tracking via `fifo_pgn` linked list

In summary, moving from 2-level to 5-level paging in this Simple OS enables massive address space expansion (from 1 MB to 128 PiB) but requires more complex address parsing, slower translation (5 memory accesses vs. 2), and sophisticated memory management strategies. The implementation successfully balances these trade-offs through sparse allocation, on-demand table creation, and efficient tree traversal algorithms, making it essential for supporting modern 64-bit operating systems and large-scale applications.

4.6 Question 6

Question: What are the advantages and disadvantages of segmentation with paging?

Answer:

In the Simple OS implementation, segmentation with paging is realized through the `vm_area_struct` (VM areas) acting as logical segments, which are then divided into fixed-size pages managed by the page table hierarchy. This hybrid approach combines the benefits of both memory management techniques.

1. **Implementation in Simple OS**

   The OS uses `vm_area_struct` (defined in `os-mm.h`) (Listing 6) as segment-like structures, each containing:

   - `vm_id`: Identifier for the VM area (logical segment identifier)

- `vm_start` and `vm_end`: Boundaries defining the logical segment's virtual address range
- `sbrk`: Break pointer for dynamic memory allocation within the segment (heap growth)
- `vm_freerg_list`: Linked list of free regions within the segment for efficient reuse
- `vm_mm`: Back-pointer to the `mm_struct` for accessing page tables

Each VM area is then paged using the hierarchical page table structure (`mm_struct->pgd` in 64-bit mode, as defined in `os-mm.h`), where pages are mapped to non-contiguous physical frames. The symbol table (`symrgtbl`) tracks memory regions (variables) within VM areas, similar to how segments track logical units. The `mm_struct->mmap` field maintains a linked list of all VM areas for a process.

```
/*
* Memory management struct
* Represents the memory management information for a process
*/
struct mm_struct
{
    /* TODO: The structure of page diractory need to be justify
    *       as your design. The single point is draft to avoid
    *       compiler noisy only, this design need to be revised
    */
    #ifdef MM64
    struct pgd_t *pgd;
    #else
    uint32_t *pgd;
    #endif

    struct vm_area_struct *mmap; // Linked list of memory areas

    /* Currently we support a fixed number of symbol */
    struct vm_rg_struct symrgtbl[PAGING_MAX_SYMTBL_SZ]; // Symbol region table

    /* list of free page */
    struct pgn_t *fifo_pgn; // FIFO page number list
};
```

Listing 25: mm_struct in os-mm.h

2. Advantages

a. Eliminates External Fragmentation: In `vm_map_ram()` (lines 322-337 of `mm64.c`) and `vmap_page_range()` (Listing 25), physical memory is allocated in fixed-size 4KB frames. The `framephy_struct` linked list (defined in `os-mm.h` lines 155-162) manages non-contiguous frames, eliminating external fragmentation entirely since segments are paged. The `alloc_pages_range()` function (lines 283-317) allocates frames individually and links them, allowing a VM area to span multiple non-contiguous physical frames.

```c
// mm64.c
/*
* vm_map_ram - do the mapping all vm are to ram storage device
*/
addr_t vm_map_ram(struct pcb_t *caller, addr_t astart, addr_t aend, addr_t
    mapstart, int incpgnum, struct vm_rg_struct *ret_rg)
{
    struct framephy_struct *frm_lst = NULL;
    addr_t ret_alloc = 0;

    ret_alloc = alloc_pages_range(caller, incpgnum, &frm_lst);

    if (ret_alloc == -3000)
    return -1;
    if (ret_alloc < 0)
    return -1;

    vmap_page_range(caller, mapstart, incpgnum, frm_lst, ret_rg);

    return 0;
}
```

Listing 26: vm_map_ram() in mm64.c

```c
// mm64.c
/*
* alloc_pages_range - allocate req_pgnum of frame in ram
*/
addr_t alloc_pages_range(struct pcb_t *caller, int req_pgnum, struct
    framephy_struct **frm_lst)
{
    int pgit;
    addr_t fpn;

    *frm_lst = NULL;
    struct framephy_struct *tail = NULL;

    for (pgit = 0; pgit < req_pgnum; pgit++)
    {
        if (MEMPHY_get_freefp(caller->krnl->mram, &fpn) == 0)
        {
            struct framephy_struct *new_node = malloc(sizeof(struct
                framephy_struct));
            new_node->fpn = fpn;
```

```
19              new_node->fp_next = NULL;
20              new_node->owner = caller->mm;
21
22              if (*frm_lst == NULL)
23              {
24                  *frm_lst = new_node;
25                  tail = new_node;
26              }
27              else
28              {
29                  tail->fp_next = new_node;
30                  tail = new_node;
31              }
32          }
33          else
34          {
35              return -3000;
36          }
37      }
38      return 0;
39  }
```

Listing 27: alloc_pages_range() in mm64.c

b. Logical Memory Organization: VM areas provide logical organization of memory, separating different memory regions (code, data, heap) conceptually. Functions like `get_vma_by_num()` (lines 12-31 of `mm-vm.c`) traverse the VM area list to find segments by ID, and `validate_overlap_vm_area()` (lines 76-102) ensures distinct logical regions don't overlap, allowing the OS to treat different memory areas as separate segments while using paging for physical allocation.

```
1   // mm-vm.c
2   /*get_vma_by_num - get vm area by numID
3   *@mm: memory region
4   *@vmaid: ID vm area to alloc memory region
5   *
6   */
7   struct vm_area_struct *get_vma_by_num(struct mm_struct *mm, int vmaid)
8   {
9       struct vm_area_struct *pvma = mm->mmap;
10
11      if (mm->mmap == NULL)
12      return NULL;
13
14      // Iterate through the entire list
15      while (pvma != NULL)
```

```
    {
        // Check for exact match
        if (pvma->vm_id == vmaid)
        return pvma;

        pvma = pvma->vm_next;
    }

    // If we reach here, the ID was not found
    return NULL;
}
```

Listing 28: get_vma_by_num() in mm-vm.c

```
// mm-vm.c
/*validate_overlap_vm_area
*@caller: caller
*@vmaid: ID vm area to alloc memory region
*@vmastart: vma end
*@vmaend: vma end
*
*/
int validate_overlap_vm_area(struct pcb_t *caller, int vmaid, addr_t vmastart
    , addr_t vmaend)
{
    if (vmastart >= vmaend)
    {
        return -1; // Invalid range
    }

    struct vm_area_struct *vma = caller->mm->mmap;

    // Traverse the linked list of VM Areas
    while (vma != NULL)
    {
        // Skip self if we are checking against our own ID (though typically
            we check new vs old)
        if (vma->vm_id != vmaid)
        {
            // Check for overlap: OVERLAP(Start1, End1, Start2, End2)
            // Logic: (Start1 < End2) && (Start2 < End1)
            if ((vmastart < vma->vm_end) && (vma->vm_start < vmaend))
            {
                return -1; // Overlap detected
            }
```

```
30          }
31          vma = vma->vm_next;
32      }
33
34      return 0;
35  }
```

Listing 29: validate_overlap_vm_area() in mm-vm.c

c. **Efficient Handling of Large Segments:** The `inc_vma_limit()` function (lines 110-168 of `mm-vm.c`) demonstrates how large VM areas can grow dynamically. Since these segments are paged, they don't require contiguous physical memory. The function:

- Calculates page-aligned size using `DIV_ROUND_UP(inc_amt, PAGING_PAGESZ)`
- Calls `vm_map_ram()` which allocates frames via `alloc_pages_range()` and maps them via `vmap_page_range()`
- Updates the `sbrk` pointer to track the new segment boundary

```
1               // mm-vm.c
2   /*inc_vma_limit - increase vm area limits to reserve space for new
        variable
3   *@caller: caller
4   *@vmaid: ID vm area to alloc memory region
5   *@inc_sz: increment size
6   *
7   */
8   int inc_vma_limit(struct pcb_t *caller, int vmaid, addr_t inc_sz)
9   {
10      struct vm_rg_struct *newrg = malloc(sizeof(struct vm_rg_struct));
11      int inc_amt = (int)inc_sz;
12
13      // Retrieve current VMA
14      struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);
15      if (!cur_vma)
16      {
17          free(newrg);
18          return -1;
19      }
20
21      // Calculate number of pages needed for this increase
22      int incnumpage = DIV_ROUND_UP(inc_amt, PAGING_PAGESZ);
23
24      // Calculate new boundaries
25      struct vm_rg_struct *area = get_vm_area_node_at_brk(caller, vmaid,
            inc_sz, inc_amt);
26      if (!area)
27      {
```

```
28          free(newrg);
29          return -1;
30      }
31
32      // Store old sbrk to define the map start
33      addr_t old_sbrk = cur_vma->sbrk;
34
35      // Validate that expanding this VMA won't collide with another VMA
36      if (validate_overlap_vm_area(caller, vmaid, area->rg_start, area->
            rg_end) < 0)
37      {
38          free(newrg);
39          free(area);
40          return -1; /* Overlap and failed allocation */
41      }
42
43      // Call the Hardware Mapper:
44      // 1. Allocates physical frames (alloc_pages_range)
45      // 2. Updates Page Tables (vmap_page_range)
46      if (vm_map_ram(caller, area->rg_start, area->rg_end,
47      old_sbrk, incnumpage, newrg) < 0)
48      {
49          free(newrg);
50          free(area);
51          return -1; /* Map the memory to MEMRAM failed (OOM) */
52      }
53
54      // Update the VMA's Break Pointer (sbrk)
55      // This officially "commits" the expansion
56      cur_vma->sbrk += inc_sz;
57      cur_vma->vm_end += inc_sz; // Assuming vm_end tracks the used limit
            in this simple model
58
59      // Cleanup: we used newrg in vm_map_ram via pointer,
60      // but the actual region tracking is often added to the free list
61      // or symbol table by the caller (libmem).
62      free(newrg);
63      free(area);
64
65      return 0;
66  }
```

Listing 30: inc_vma_limit() in mm-vm.c

This allows segments to grow without requiring contiguous physical memory allocation.

d. Reduced Internal Fragmentation: The `vm_freerg_list` within each VM area tracks free regions, allowing efficient reuse of freed memory within segments. The `get_free_vmrg_area()` function (lines 384-437 of `libmem.c`) searches the free list for regions large enough to satisfy allocation requests, reducing internal fragmentation. The symbol table (`symrgtbl`) with `PAGING_MAX_SYMTBL_SZ = 30` entries enables tracking of variables within logical segments, and the `__alloc()` function first attempts to reuse free regions before expanding the segment.

```
// libmem.c
/* get_free_vmrg_area - get a free vm region */
int get_free_vmrg_area(struct pcb_t *caller, int vmaid, int size, struct
    vm_rg_struct *newrg)
{
    struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);
    if (!cur_vma)
    return -1;

    struct vm_rg_struct *rgit = cur_vma->vm_freerg_list;
    if (rgit == NULL)
    return -1;

    newrg->rg_start = newrg->rg_end = -1;

    // Traverse free list
    while (rgit != NULL)
    {
        if (rgit->rg_start + size <= rgit->rg_end)
        {
            /* Found fit */
            newrg->rg_start = rgit->rg_start;
            newrg->rg_end = rgit->rg_start + size;

            /* Update free node */
            if (rgit->rg_start + size < rgit->rg_end)
            {
                rgit->rg_start += size;
            }
            else
            {
                /* Perfectly matches, remove this free node */
                struct vm_rg_struct *nextrg = rgit->rg_next;
                if (nextrg)
                {
                    rgit->rg_start = nextrg->rg_start;
                    rgit->rg_end = nextrg->rg_end;
                    rgit->rg_next = nextrg->rg_next;
```

```
38                      free(nextrg);
39                  }
40                  else
41                  {
42                      // This was the last node, mark as invalid/empty
43                      rgit->rg_start = rgit->rg_end = 0;
44                      rgit->rg_next = NULL;
45                      // Ideally we should remove rgit from the list head if it
                            's the head
46                      // But standard linked list removal is tricky with just '
                            rgit' ptr
47                      // Assuming list management handles this or next alloc
                            handles 0-size
48                  }
49              }
50              return 0;
51          }
52          rgit = rgit->rg_next;
53      }
54      return -1; // No fit found
55  }
```

Listing 31: get_free_vmrg_area() in libmem.c

e. **Support for Dynamic Growth:** The `sbrk` pointer in each VM area allows segments to grow dynamically. The `get_vm_area_node_at_brk()` function (lines 47-67 of `mm-vm.c`) creates new regions starting at the current `sbrk` position. When `get_free_vmrg_area()` fails (line 45 of `libmem.c`, inside `__alloc()` function), the system expands the heap via `inc_vma_limit()`, which calls `vm_map_ram()` to allocate and map new pages. This enables processes to extend their segments on demand without pre-allocation.

```
1   // mm-vm.c
2   struct vm_rg_struct *get_vm_area_node_at_brk(struct pcb_t *caller, int vmaid,
        addr_t size, addr_t alignedsz)
3   {
4       struct vm_rg_struct *newrg;
5
6       // Retrieve the VMA (usually ID 0 for data/heap)
7       struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);
8       if (!cur_vma)
9       return NULL;
10
11      newrg = malloc(sizeof(struct vm_rg_struct));
12
13      // The new region starts where the current Heap ends (sbrk)
```

```
    newrg->rg_start = cur_vma->sbrk;

    // It ends after 'size' bytes
    newrg->rg_end = newrg->rg_start + size;

    newrg->rg_next = NULL;

    return newrg;
}
```

Listing 32: get_vm_area_node_at_brk() in mm-vm.c

```
// libmem.c
/* __alloc - allocate a region memory */
int __alloc(struct pcb_t *caller, int vmaid, int rgid, addr_t size, addr_t *
    alloc_addr)
{
    pthread_mutex_lock(&mmvm_lock);
    struct vm_rg_struct rgnode;

    // Check if we can reuse a free region
    if (get_free_vmrg_area(caller, vmaid, size, &rgnode) == 0)
    {
        caller->mm->symrgtbl[rgid].rg_start = rgnode.rg_start;
        caller->mm->symrgtbl[rgid].rg_end = rgnode.rg_end;
        *alloc_addr = rgnode.rg_start;
        pthread_mutex_unlock(&mmvm_lock);
        return 0;
    }

    /* If get_free_vmrg_area FAILED, we must expand the heap */
    struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);
    if (!cur_vma)
    {
        pthread_mutex_unlock(&mmvm_lock);
        return -1;
    }

    addr_t inc_sz = PAGING_PAGE_ALIGNSZ(size);
    // Usually aligning to page size is good practice, or use exact size if
        your inc_vma handles it.
    // The provided skeleton had some ifdef logic, let's stick to standard
        alignment.

    addr_t old_sbrk = cur_vma->sbrk;
```

```
31
32      /* SYSCALL to increase limit */
33      struct sc_regs regs;
34      regs.a1 = SYSMEM_INC_OP;
35      regs.a2 = vmaid;
36      regs.a3 = inc_sz; // Request expansion
37
38      // Note: In real syscall, we don't pass PCB, but here we simulate it via
            wrapper
39      // The wrapper 'syscall' takes (krnl, pid, nr, regs)
40      if (syscall(caller->krnl, caller->pid, 17, &regs) < 0)
41      {
42          pthread_mutex_unlock(&mmvm_lock);
43          return -1; // Failed to expand
44      }
45
46      /* Successful increase limit */
47      caller->mm->symrgtbl[rgid].rg_start = old_sbrk;
48      caller->mm->symrgtbl[rgid].rg_end = old_sbrk + size;
49      *alloc_addr = old_sbrk;
50
51      // The remaining space (inc_sz - size) should technically be added to
            free list
52      // to avoid internal fragmentation, but for this simple assignment we
            might skip it
53      // or add logic:
54      if (inc_sz > size)
55      {
56          struct vm_rg_struct *fragment = malloc(sizeof(struct vm_rg_struct));
57          fragment->rg_start = old_sbrk + size;
58          fragment->rg_end = old_sbrk + inc_sz;
59          fragment->rg_next = NULL;
60          enlist_vm_freerg_list(caller->mm, fragment);
61      }
62
63      pthread_mutex_unlock(&mmvm_lock);
64      return 0;
65  }
```

Listing 33: __alloc() in libmem.c

3. Disadvantages

a. **Increased Translation Complexity:** Address translation requires two conceptual steps: first identifying the VM area (segment) that contains the address, then translating the address through the

page table hierarchy. In practice, the implementation uses the page table directly (via `get_page_table_entry()` in `mm64.c`, Listing 12), but VM area validation adds overhead. In 64-bit mode, address translation involves extracting indices from PGD, P4D, PUD, PMD, and PT levels (as in `get_pd_from_address()`, lines 149-157 of `mm64.c`, Listing 16), requiring 5 memory accesses. Additionally, functions like `__read()` (lines 279-290 of `libmem.c`) must first validate the address is within the correct VM area region via `get_symrg_byid()` before performing page translation.

```
// libmem.c
/* __read - read value in region memory */
int __read(struct pcb_t *caller, int vmaid, int rgid, addr_t offset, BYTE *
    data)
{
    struct vm_rg_struct *currg = get_symrg_byid(caller->mm, rgid);
    if (!currg)
    return -1;

    // Bounds check
    if (currg->rg_start + offset >= currg->rg_end)
    return -1;

    return pg_getval(caller->mm, currg->rg_start + offset, data, caller);
}
```

Listing 34: __read() in libmem.c

b. Memory Overhead for Management Structures: Each process maintains a linked list of VM areas (`mm_struct->mmap`), each with its own free region list (`vm_freerg_list`). Additionally, the hierarchical page table structure (`mm_struct->pgd` in 64-bit mode) must be maintained, with each directory level consuming 4KB when allocated. The `symrgtbl` array (`PAGING_MAX_SYMTBL_SZ = 30`) adds further overhead for tracking memory regions within segments. Each VM area structure itself consumes memory, and the free region lists within each VM area add additional overhead compared to a pure paging system.

c. Complex Overlap Validation: The `validate_overlap_vm_area()` function in `mm-vm.c` (Listing 29) must check all VM areas to prevent overlaps, requiring traversal of the entire `mmap` linked list. This adds O(n) overhead during memory allocation operations, as seen in `inc_vma_limit()` and `__alloc()` when expanding segments. The overlap check logic (lines 15-18 below) compares the new region against all existing VM areas, which becomes expensive as the number of VM areas grows.

```
// mm-vm.c
int validate_overlap_vm_area(struct pcb_t *caller, int vmaid, addr_t vmastart
    , addr_t vmaend)
{
    /*
    ...
    */
```

```
7
8       while (vma != NULL)
9       {
10          // Skip self if we are checking against our own ID (though typically
                we check new vs old)
11          if (vma->vm_id != vmaid)
12          {
13              // Check for overlap: OVERLAP(Start1, End1, Start2, End2)
14              // Logic: (Start1 < End2) && (Start2 < End1)
15              if ((vmastart < vma->vm_end) && (vma->vm_start < vmaend))
16              {
17                  return -1; // Overlap detected
18              }
19          }
20          vma = vma->vm_next;
21      }
22
23      return 0;
24  }
```

Listing 35: Overlap check logic

d. Page Table Management Complexity: The `mm_struct` must maintain both VM area structures and page table structures simultaneously. In 64-bit mode, the structure contains the root PGD pointer (`mm->pgd`), and the initialization in `init_mm()` (`mm64.c`, Listing 22) shows the complexity of managing both segment-like VM areas (initializing `mmap`, `symrgtbl`) and hierarchical page tables (initializing `pgd` as NULL). The `vmap_page_range()` function must walk the 5-level page table hierarchy while also tracking which VM area the pages belong to, adding complexity to memory mapping operations.

e. Swapping Complexity: The swapping mechanism operates at the page level via `pg_getpage()` (lines 177-237 of `libmem.c`), but the VM area structure adds complexity because:

- The FIFO page replacement (`fifo_pgn` list, line 142 of `os-mm.h`) operates at the page level without direct VM area context
- When a page is swapped out, the system must ensure the page table entry is properly marked (via `pte_set_swap()`, lines 171-187 of `mm64.c`)
- The `find_victim_page()` function (lines 352-381 of `libmem.c`) selects victims from the FIFO list without considering VM area boundaries, which could lead to swapping pages from different logical segments

The `__mm_swap_page()` function (lines 33-37 of `mm-vm.c`) handles the actual swap copy, but the coordination between VM area tracking and page-level swapping adds complexity.

```
1 // libmem.c
2 /* find_victim_page - FIFO replacement policy */
```

```
3  int find_victim_page(struct mm_struct *mm, addr_t *retpgn)
4  {
5      struct pgn_t *pg = mm->fifo_pgn;
6      if (!pg)
7      return -1; // Empty list
8
9      struct pgn_t *prev = NULL;
10
11     // Traverse to the end of the list (Oldest item)
12     while (pg->pg_next)
13     {
14         prev = pg;
15         pg = pg->pg_next;
16     }
17
18     *retpgn = pg->pgn;
19
20     // Unlink
21     if (prev)
22     {
23         prev->pg_next = NULL;
24     }
25     else
26     {
27         mm->fifo_pgn = NULL; // Only one item was in list
28     }
29
30     free(pg);
31     return 0;
32 }
```

Listing 36: find_victim_page() in libmem.c

f. Synchronization Overhead: The `__alloc()` function uses `pthread_mutex_lock(&mmvm_lock)` to synchronize access to VM areas and page tables. This mutex protects both segment-level operations (VM area list traversal, free region management) and page-level operations (page table updates). The `__write()` function (line 312) also acquires this lock, and `__read()` performs page table operations that may require synchronization. This single mutex protects multiple data structures, potentially creating contention points that wouldn't exist in a pure paging system where only page tables need protection.

g. Free Region List Management Overhead: The `get_free_vmrg_area()` function (`libmem.c`) must traverse the free region list within a VM area to find suitable regions, which can be O(n) in the number of free regions. When regions are freed via `__free()`, the function must create new free region nodes and insert them into the list, potentially fragmenting the free list. The

`enlist_vm_freerg_list()` function adds regions to the head of the list, which is efficient but doesn't merge adjacent free regions, potentially leading to fragmentation.

In summary, the Simple OS's segmentation-with-paging approach (VM areas + page tables) provides logical memory organization and eliminates external fragmentation, but at the cost of increased translation complexity, memory overhead for management structures, O(n) overlap validation, complex coordination between segment and page-level operations, and synchronization overhead. The hybrid system requires maintaining both VM area metadata and page table structures, with operations spanning both abstraction levels, making the implementation more complex than a pure paging system.

4.7 Question 7

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:

The Simple OS uses a multi-threaded architecture where multiple CPU threads (from `cpu_routine()` in `os.c`) and a loader thread (from `ld_routine()`) execute concurrently, sharing critical data structures. Without proper synchronization, severe system failures would occur.

1. **Synchronization Mechanisms in Simple OS**

   The OS implements three main synchronization mechanisms:

   - **Timer Synchronization** (`timer.c`): Uses mutexes (`event_lock`, `timer_lock`) and condition variables (`event_cond`, `timer_cond`) to coordinate time slot advancement. The `timer_routine()` waits for all devices to complete before incrementing `_time`, and `next_slot()` synchronizes CPU/loader threads with the timer.
   - **Scheduler Synchronization** (`sched.c`): Uses `queue_lock` mutex to protect shared scheduler state including `mlq_ready_queue[]`, `slot_remaining[]`, `current_queue_index`, and `running_list`. Functions like `get_mlq_proc()`, `put_mlq_proc()`, and `add_mlq_proc()` are all protected by this mutex.
   - **Memory Management Synchronization** (`libmem.c`): Uses `mmvm_lock` mutex to protect VM area structures and page table operations. Functions like `__alloc()`, `__write()`, and `__read()` acquire this lock before accessing memory management structures.

2. **Consequences of Missing Synchronization**

   a. **Ready Queue Corruption:** Without `queue_lock` protection, multiple CPU threads could simultaneously call `get_mlq_proc()` and `put_mlq_proc()`, leading to race conditions on `mlq_ready_queue[]`. This could result in:

      - **Lost Processes:** A process dequeued by one CPU thread might be simultaneously dequeued by another, causing one CPU to receive `NULL` or an invalid process pointer.
      - **Duplicate Process Execution:** The same process could be dispatched to multiple CPUs simultaneously, leading to inconsistent state and data corruption.

- **Queue Structure Corruption:** Concurrent enqueue/dequeue operations could corrupt the queue's internal structure (size, pointers), causing segmentation faults or infinite loops.

b. **Scheduler State Inconsistency:** The scheduler maintains state variables `slot[]` and `mlq_ready_queue` that must be updated atomically. Without synchronization:

- **Incorrect Priority Scheduling:** Multiple threads reading and updating `mlq_ready_queue` simultaneously could cause the scheduler to skip priority levels or process them out of order, violating the MLQ scheduling policy.
- **Slot Counter Corruption:** The `slot[MAX_PRIO]` counter could be incorrectly decremented by multiple threads, causing processes to receive incorrect time slices or be prematurely moved to lower priority queues.

c. **Timer Desynchronization:** The timer mechanism in `timer_routine()` coordinates all threads by waiting for each device's `done` flag before advancing time slots. Without proper mutex protection:

- **Time Slot Skips:** If multiple threads call `next_slot()` simultaneously, the timer might advance before all threads complete their work, causing some threads to miss time slots or execute out of order.
- **Deadlock in Timer:** The condition variable waits in `timer_routine()` (lines 18-20 below) and `next_slot()` (lines 61-63 below) could deadlock if mutexes are not properly acquired, freezing the entire system.
- **Inconsistent Time Perception:** Different CPU threads might observe different values of `current_time()` if the timer advances while threads are reading it, leading to incorrect process scheduling decisions.

```
1  // timer.c
2  static void *timer_routine(void *args)
3  {
4      while (!timer_stop)
5      {
6          printf("Time slot %3llu\n", current_time());
7          int fsh = 0;
8          int event = 0;
9          /* Wait for all devices have done the job in current
10         * time slot */
11         struct timer_id_container_t *temp;
12         for (temp = dev_list; temp != NULL; temp = temp->next)
13         {
14             pthread_mutex_lock(&temp->id.event_lock);
15             while (!temp->id.done && !temp->id.fsh)
16             {
17                 pthread_cond_wait(
18                 &temp->id.event_cond,
19                 &temp->id.event_lock);
20             }
```

```
21              if (temp->id.fsh)
22              {
23                  fsh++;
24              }
25              event++;
26              pthread_mutex_unlock(&temp->id.event_lock);
27          }
28
29          /* Increase the time slot */
30          _time++;
31
32          /* Let devices continue their job */
33          for (temp = dev_list; temp != NULL; temp = temp->next)
34          {
35              pthread_mutex_lock(&temp->id.timer_lock);
36              temp->id.done = 0;
37              pthread_cond_signal(&temp->id.timer_cond);
38              pthread_mutex_unlock(&temp->id.timer_lock);
39          }
40          if (fsh == event)
41          {
42              break;
43          }
44      }
45      pthread_exit(args);
46  }
47
48  void next_slot(struct timer_id_t *timer_id)
49  {
50      /* Tell to timer that we have done our job in current slot */
51      pthread_mutex_lock(&timer_id->event_lock);
52      timer_id->done = 1;
53      pthread_cond_signal(&timer_id->event_cond);
54      pthread_mutex_unlock(&timer_id->event_lock);
55
56      /* Wait for going to next slot */
57      pthread_mutex_lock(&timer_id->timer_lock);
58      while (timer_id->done)
59      {
60          pthread_cond_wait(
61          &timer_id->timer_cond,
62          &timer_id->timer_lock);
63      }
64      pthread_mutex_unlock(&timer_id->timer_lock);
```

```
65 }
```

Listing 37: timer_routine() and next_slot() in timer.c

d. **Memory Management Corruption:** Without `mmvm_lock` protection, concurrent memory operations could corrupt critical structures:

- **VM Area List Corruption:** Multiple threads accessing `mm_struct->mmap` (the VM area linked list) simultaneously could corrupt list pointers, causing processes to access wrong memory regions or crash when traversing the list.
- **Page Table Race Conditions:** Concurrent calls to `pte_set_fpn()` and `pte_get_entry()` could result in reading partially updated page table entries, causing incorrect address translations and memory access violations.
- **Symbol Table Corruption:** The `symrgtbl[]` array could be corrupted if multiple threads allocate memory regions simultaneously, leading to processes accessing incorrect memory addresses or overwriting each other's data.

e. **Process State Corruption:** In `cpu_routine()`, each CPU thread maintains local state (`proc`, `time_left`) but accesses shared process control blocks. Without synchronization:

- **Concurrent Process Execution:** The same `pcb_t` structure could be modified by multiple CPU threads simultaneously, corrupting the program counter (`pc`), registers, or memory mappings.
- **Inconsistent Process Termination:** A process might be freed by one CPU thread while another is still executing it, leading to use-after-free errors and system crashes.

f. **Loader-CPU Race Conditions:** The `ld_routine()` adds processes to the ready queue via `add_proc()` while CPU threads are simultaneously removing processes via `get_proc()`. Without synchronization:

- **Process Addition During Dequeue:** A process might be added to the queue while another thread is in the middle of dequeuing, causing the new process to be lost or the dequeue operation to fail.
- **Incomplete Process Initialization:** A CPU thread might dispatch a process before the loader finishes initializing its memory structures (`mm_struct`), leading to null pointer dereferences or uninitialized memory access.

3. **System-Wide Failures**

Without synchronization, the Simple OS would experience cascading failures:

- **Non-Deterministic Behavior:** The same input could produce different outputs on each run due to race conditions, making debugging and testing extremely difficult.
- **System Crashes:** Segmentation faults from corrupted pointers, null pointer dereferences, or invalid memory accesses would cause the entire OS simulation to crash.
- **Resource Leaks:** Lost processes or corrupted queue structures could prevent proper cleanup, leading to memory leaks and resource exhaustion.

- **Incorrect Execution Results:** Even if the system doesn't crash, processes might execute incorrectly due to corrupted state, producing wrong outputs that are difficult to detect.

4. Experimental Demonstration: Removing Timer Synchronization

To demonstrate the consequences of missing synchronization, we removed all mutex protection from `timer_routine()` (`timer.c`) and `next_slot()` functions. This eliminates the critical synchronization mechanism that coordinates time slot advancement across all CPU threads and the loader thread.

a. Experimental Setup

The mutexes `event_lock` and `timer_lock` (defined per device in `struct timer_id_t`), along with their associated condition variables (`event_cond` and `timer_cond`), were removed from the timer synchronization code. Specifically:

- In `timer_routine()`: Removed mutex locks at lines 32, 44, 53, 56 that protect the `done` flag checks and updates

```
1  // timer.c
2  static void *timer_routine(void *args)
3  {
4              /*...*/
5          for (temp = dev_list; temp != NULL; temp = temp->next)
6          {
7  //line 32    pthread_mutex_lock(&temp->id.event_lock);
8              while (!temp->id.done && !temp->id.fsh)
9              {
10                 pthread_cond_wait(
11                 &temp->id.event_cond,
12                 &temp->id.event_lock);
13             }
14             if (temp->id.fsh)
15             {
16                 fsh++;
17             }
18             event++;
19 //line 44    pthread_mutex_unlock(&temp->id.event_lock);
20         }
21             /*...*/
22         for (temp = dev_list; temp != NULL; temp = temp->next)
23         {
24 //line 53    pthread_mutex_lock(&temp->id.timer_lock);
25             temp->id.done = 0;
26             pthread_cond_signal(&temp->id.timer_cond);
27 //line 56    pthread_mutex_unlock(&temp->id.timer_lock);
28         }
29             /*...*/
30 }
```

Listing 38: Removed muxtex timer_routine()

- In `next_slot()`: Removed mutex locks at lines 69, 72, 75, 82 that coordinate between threads and the timer

```
// timer.c
void next_slot(struct timer_id_t *timer_id)
{
            /* Tell to timer that we have done our job in current slot */
//line 69   pthread_mutex_lock(&timer_id->event_lock);
            timer_id->done = 1;
            pthread_cond_signal(&timer_id->event_cond);
//line 72   pthread_mutex_unlock(&timer_id->event_lock);

            /* Wait for going to next slot */
//line 75   pthread_mutex_lock(&timer_id->timer_lock);
            while (timer_id->done)
            {
                pthread_cond_wait(
                &timer_id->timer_cond,
                &timer_id->timer_lock);
            }
//line 82   pthread_mutex_unlock(&timer_id->timer_lock);
}
```

Listing 39: Removed muxtex next_slot()

This allows multiple threads (CPU threads from `cpu_routine()` in `os.c` and the loader thread from `ld_routine()`) to access and modify the timer state (`_time` of `timer.c`) and device `done` flags concurrently without any protection.

b. Observed Output

The following output was captured from a test run with timer synchronization removed:

```
Time slot   0
ld_routine
    Loaded a process at input/proc/p0s, PID: 1 PRIO: 0

Time slot   1
    CPU 0: Dispatched process  1
    Loaded a process at input/proc/p1s, PID: 2 PRIO: 15
    CPU 1: Dispatched process  2

Time slot   2
liballoc:152
--- Page Table Dump ---
```

```
print_pgtbl:
 PGD=0x7fbab8001000 P4D=0x7fbab8003000 PUD=0x7fbab8005000 PMD=0x7fbab8007000
     PT=0x7fbab8009000
0000000000000000: 0000000000000001

Time slot   3
liballoc:152
--- Page Table Dump ---
print_pgtbl:
 PGD=0x7fbab8001000 P4D=0x7fbab8003000 PUD=0x7fbab8005000 PMD=0x7fbab8007000
     PT=0x7fbab8009000
0000000000000000: 0000000000000001
    Loaded a process at input/proc/p1s, PID: 3 PRIO: 0

Time slot   4
libfree:168
--- Page Table Dump ---
print_pgtbl:
 PGD=0x7fbab8001000 P4D=0x7fbab8003000 PUD=0x7fbab8005000 PMD=0x7fbab8007000
     PT=0x7fbab8009000
0000000000000000: 0000000000000001

Time slot   5
liballoc:152
--- Page Table Dump ---
print_pgtbl:
 PGD=0x7fbab8001000 P4D=0x7fbab8003000 PUD=0x7fbab8005000 PMD=0x7fbab8007000
     PT=0x7fbab8009000
0000000000000000: 0000000000000001

Time slot   6
libwrite:343
--- Page Table Dump ---
print_pgtbl:
 PGD=0x7fbab8001000 P4D=0x7fbab8003000 PUD=0x7fbab8005000 PMD=0x7fbab8007000
     PT=0x7fbab8009000
0000000000000000: 0000000000000001
    Loaded a process at input/proc/p0s, PID: 4 PRIO: 0
    CPU 0: Put process  1 to run queue
    CPU 1: Put process  2 to run queue
    CPU 1: Dispatched process  3
    CPU 0: Dispatched process  4

Time slot   7
```

```
52
53 Time slot   8
54 liballoc:152
55 --- Page Table Dump ---
56 print_pgtbl:
57  PGD=0x7fbab800b000 P4D=0x7fbab800d000 PUD=0x7fbab800f000 PMD=0x7fbab8011000
       PT=0x7fbab8013000
58 0000000000000000: 0000000000001001
59
60 Time slot   9
61 liballoc:152
62 --- Page Table Dump ---
63 print_pgtbl:
64  PGD=0x7fbab800b000 P4D=0x7fbab800d000 PUD=0x7fbab800f000 PMD=0x7fbab8011000
       PT=0x7fbab8013000
65 0000000000000000: 0000000000001001
66
67 Time slot   10
68 libfree:168
69 --- Page Table Dump ---
70 print_pgtbl:
71  PGD=0x7fbab800b000 P4D=0x7fbab800d000 PUD=0x7fbab800f000 PMD=0x7fbab8011000
       PT=0x7fbab8013000
72 0000000000000000: 0000000000001001
73
74 Time slot   11
75 liballoc:152
76 --- Page Table Dump ---
77 print_pgtbl:
78  PGD=0x7fbab800b000 P4D=0x7fbab800d000 PUD=0x7fbab800f000 PMD=0x7fbab8013000
79 0000000000000000: 0000000000001001
80
81 Time slot   12
82 libwrite:343
83 --- Page Table Dump ---
84 print_pgtbl:
85  PGD=0x7fbab800b000 P4D=0x7fbab800d000 PUD=0x7fbab800f000 PMD=0x7fbab8011000
       PT=0x7fbab8013000
86 0000000000000000: 0000000000001001
87
88 Time slot   13
89     CPU 1: Put process  3 to run queue
90     CPU 1: Dispatched process  1
91 libread:304
```

```
92      CPU 0: Put process  4 to run queue
93      CPU 0: Dispatched process  3
94
95  Time slot   14
96
97  Time slot   15
98
99  Time slot   16
100
101 Time slot   17
102     CPU 0: Processed  3 has finished
103     CPU 0: Dispatched process  4
104 libread:304
105
106 Time slot   18
107
108 Time slot   19
109     CPU 1: Put process  1 to run queue
110     CPU 1: Dispatched process  1
111 libfree:168
112 --- Page Table Dump ---
113 print_pgtbl:
114  PGD=0x7fbab8001000 P4D=0x7fbab8003000 PUD=0x7fbab8005000 PMD=0x7fbab8007000
        PT=0x7fbab8009000
115 0000000000000000: 0000000000000001
116
117 Time slot   20
118
119 Time slot   21
120     CPU 1: Processed  1 has finished
121     CPU 1: Dispatched process  2
122
123 Time slot   22
124
125 Time slot   23
126     CPU 0: Put process  4 to run queue
127     CPU 0: Dispatched process  4
128 libfree:168
129 --- Page Table Dump ---
130 print_pgtbl:
131  PGD=0x7fbab800b000 P4D=0x7fbab800d000 PUD=0x7fbab800f000 PMD=0x7fbab8011000
        PT=0x7fbab8013000
132 0000000000000000: 0000000000001001
133
```

```
Time slot   24

Time slot   25
    CPU 0: Processed  4 has finished
    CPU 1: Processed  2 has finished
    CPU 0 stopped
    CPU 1 stopped
```

c. Analysis of the Results

While this particular run completed successfully, several critical observations highlight the dangers of missing synchronization:

- **Concurrent Operations in Same Time Slot:** At time slot 6, we observe multiple events occurring simultaneously within the same time slot: `libwrite:343`, process loading (PID 4), CPU 0 putting process 1 to run queue, CPU 1 putting process 2 to run queue, and both CPUs dispatching new processes (3 and 4). Without timer synchronization, the loader thread, CPU 0, and CPU 1 are all executing concurrently. The `next_slot()` function (called from `cpu_routine()` at lines 66, 99, 112 and from `ld_routine()` at lines 140, 155 in `os.c`) should coordinate with `timer_routine()` to ensure only one time slot is active at a time. Without mutexes, threads can read and modify the `done` flag and `_time` variable concurrently, causing operations from different logical time slots to interleave.
- **Potential Page Table Race Conditions:** While the page table dumps appear consistent in this run, without proper synchronization, concurrent access to page table structures could lead to race conditions. Multiple threads could simultaneously call `pte_set_fpn()` or `pte_get_entry()` (in `mm64.c`), which traverse the 5-level page table hierarchy via `get_page_table_entry()`. Without mutex protection, one thread could be updating a page table entry (via `get_next_level()` allocating new directory levels) while another thread reads it, potentially observing partially updated pointers or inconsistent states. The fact that this run shows consistent page tables does not guarantee correctness—race conditions are non-deterministic and may manifest differently in subsequent runs.
- **Concurrent Memory Operations:** At time slot 13, we see both CPUs performing operations simultaneously: CPU 1 putting process 3 to run queue and dispatching process 1, CPU 1 performing `libread:304`, CPU 0 putting process 4 to run queue and dispatching process 3. CPU 1 is reading memory while CPU 0 is dispatching a process. Without proper synchronization, these operations could interfere with each other, especially if they access shared scheduler structures or memory management data.
- **Non-Deterministic Execution Order:** The output shows that operations from different threads can appear in any order within a time slot. For example, at time slot 6, the process loading, CPU queue operations, and dispatches all occur in an unpredictable sequence. This non-determinism makes it impossible to reproduce bugs consistently, as the exact interleaving depends on thread scheduling by the operating system.

- **Time Slot Coordination Failure:** The `timer_routine()` function (lines 20-64 of `timer.c`) is designed to:
  i. Wait for all devices to set their `done` flag (lines 30-45)
  ii. Increment `_time` (line 48)
  iii. Signal all devices to continue (lines 51-57)

  Without mutex protection, multiple threads can call `next_slot()` simultaneously, and the timer thread can advance `_time` while other threads are still reading it or setting their `done` flags. This breaks the atomicity of time slot transitions, causing threads to operate on inconsistent time values.
- **The Illusion of Correctness:** The fact that this run completed successfully demonstrates a critical characteristic of race conditions: they are **non-deterministic**. The absence of visible errors in one run does not guarantee correctness. While this particular run shows consistent page table structures and no obvious data corruption, the concurrent operations observed (especially at time slots 6 and 13) indicate that threads are executing without proper coordination. Subsequent runs with the same input might:
  – Crash due to corrupted pointers or invalid memory accesses
  – Produce incorrect results due to data corruption
  – Deadlock if threads wait on condition variables without proper mutex protection (the condition variable waits in `timer_routine()` and `next_slot()` require mutexes to function correctly)
  – Exhibit different execution orders, making debugging and testing impossible

d. Why This Demonstrates the Problem

The removal of timer synchronization mutexes creates a **time-of-check to time-of-use (TOCTOU)** vulnerability and breaks the atomicity of time slot transitions. The synchronization protocol in `timer.c` works as follows:

i. Each device thread calls `next_slot()` which sets `done = 1` and signals the timer (lines 69-72)
ii. The device then waits for the timer to advance (lines 75-82)
iii. The timer waits for all devices to set `done = 1` (lines 30-45)
iv. The timer increments `_time` and resets all `done` flags (lines 48, 54-56)
v. The timer signals all devices to continue (line 55)

Without mutex protection, this protocol breaks:

- Thread A might read `_time = 5` and set `done = 1`
- Thread B might read `_time = 5` and also set `done = 1`
- The timer thread might increment `_time` to 6 while Thread A is still in the middle of its operation
- Thread A completes its operation thinking it's in time slot 5, but the timer has already advanced
- Multiple threads can observe different values of `_time` simultaneously

The concurrent operations observed at time slots 6 and 13 provide concrete evidence of this race condition: multiple threads are executing operations simultaneously without proper coordination, breaking the atomicity of time slot transitions.

In summary, synchronization is essential for the Simple OS's correctness. The mutex-protected critical sections in `timer.c` (protecting `_time` and `done` flags), `sched.c` (protecting queue operations), and `libmem.c` (protecting memory management structures) ensure that shared data structures are accessed atomically, preventing race conditions, data corruption, and system failures. The experimental removal of timer synchronization demonstrates that even when a run appears successful, the underlying race conditions create non-deterministic behavior. The concurrent operations observed in the output show that threads are not properly coordinated, which can lead to subtle bugs, data corruption, and unpredictable system behavior in subsequent runs.

5 Overall

In this assignment, we have successfully designed and implemented the core components of a simple Operating System simulation, focusing on a Symmetric Multiprocessing (SMP) environment. The project integrated three critical modules: a Slot-based Scheduler, a Hybrid Memory Management system, and a robust Synchronization mechanism.

For process management, we implemented a Slot-based Multi-Level Queue (MLQ) scheduling algorithm. Unlike standard priority scheduling which suffers from starvation, our design introduces a dynamic slot mechanism ($slot = MAX\_PRIO - prio$) combined with Round-Robin execution within each queue. This approach ensures a balance between honoring process priority and maintaining fairness, guaranteeing that lower-priority tasks eventually receive CPU resources even under heavy load. The implementation successfully handles context switching and load balancing across multiple simulated CPUs.

Regarding memory management, we constructed a sophisticated hybrid architecture that combines Segmentation with 5-Level Paging. A focal point of this implementation is the rigorous simulation of the hardware page walk mechanism.We programmed the traversal logic that iteratively resolves a 64-bit virtual address through the five hierarchical levels: PGD $\rightarrow$ P4D $\rightarrow$ PUD $\rightarrow$ PMD $\rightarrow$ PT. By utilizing precise bit-masking and shifting operations to extract directory indices at each level, our system accurately mimics the translation process of modern MMUs. This mechanism allows the OS to handle sparse memory allocation efficiently within a massive 128 PiB address space while ensuring strict isolation between user and kernel modes.

Furthermore, we addressed the critical challenge of concurrency in a multi-threaded architecture. We identified potential race conditions that could lead to data corruption or system crashes when multiple CPUs and the loader access shared resources simultaneously. By strictly applying synchronization mechanisms—specifically Mutex locks on the ready queue, memory structures, and the global timer—we ensured the atomicity of critical operations and the deterministic behavior of the system.

In conclusion, this assignment has provided a comprehensive hands-on experience in building OS kernel components. It bridged the gap between theoretical concepts—such as virtual memory translation, process scheduling, and thread safety—and practical system programming. The insights gained from handling the complexity of the 5-level page walking algorithm and SMP synchronization serve as a solid foundation for understanding the architecture of modern, high-performance operating systems.

6 Source Code

1. **Scheduler:** Link GitHub
2. **Memory Management:** Link GitHub