Vietnam National University, Ho Chi Minh City
University of Technology
Faculty of Computer Science and Engineering

OPERATING SYSTEMS (C02017)

ASSIGNMENT

SIMPLE OPERATING SYSTEM

Instructor: Nguyễn Minh Tâm
Class: L04
Students: Lê Hoàng Tân - 2313050
Đỗ Đăng Khoa - 2311581
Nguyễn Minh Trí - 2313602
Trương Hoàng Nam - 2312202
Trịnh Tiến Đạt - 2310712

Ho Chi Minh City - 2025

Contents

1 Introduction

Operating systems play a fundamental role in modern computer systems by managing hardware resources and providing essential services to user applications. Key responsibilities of an operating system include process scheduling, memory management, and the abstraction of hardware complexity. Understanding how these components interact is crucial for comprehending the internal behavior and performance characteristics of an operating system.

This assignment focuses on the design and implementation of a simplified operating system simulator, namely *LamiaAtrium*. The simulator is designed to model core operating system mechanisms, including multi-core CPU scheduling, paging-based virtual memory management, and system call handling. Rather than aiming for a fully functional production operating system, the simulator emphasizes conceptual correctness and clarity, enabling students to explore operating system principles through hands-on implementation.

A major objective of this work is the implementation of a Multi-Level Queue (MLQ) scheduler to manage multiple processes with different priorities in a multi-CPU environment. The MLQ scheduler demonstrates how priority-based scheduling policies affect process execution order, fairness, and CPU utilization. Through execution logs and scheduling timelines, the behavior of the scheduler can be systematically analyzed and evaluated.

Another central component of this assignment is the memory management subsystem. The system employs paging-based virtual memory to isolate the address spaces of processes while efficiently utilizing physical memory resources. Furthermore, this work extends the paging mechanism to support a 64-bit virtual address space using a multi-level page table structure. By adopting demand allocation for page tables, the design significantly reduces memory overhead compared to flat page table approaches, while preserving correct address translation semantics.

The scope of this report includes the architectural design of the simulated operating system, detailed descriptions of the scheduling and memory management mechanisms, and an evaluation of the implemented multi-level paging scheme. Experimental results obtained from execution traces are analyzed to verify correctness and to discuss the advantages and limitations of the proposed design. Through this assignment, practical insights into operating system design trade-offs are gained, particularly in the context of scheduling policies and virtual memory management.

2 Overall System Architecture

This section describes the architectural components of the simulated operating system, the interactions between them, and the flow of control through the system. The simulator implements key operating system mechanisms including process management, CPU scheduling, memory management, and system call handling in a modular and extensible manner.

2.1 High-Level Components

At a high level, the simulator consists of the following major components:

- **Kernel:** The core module responsible for initializing system state, managing global data structures, coordinating scheduling, and handling systemwide resources.
- **Process Control Blocks (PCBs):** Each process is represented by a PCB, which stores process-specific state including registers, priority, memory management structures, and ready/sleep status.
- **Scheduler:** Implements a Multi-Level Queue (MLQ) scheduling policy to manage process execution order across one or more simulated CPUs.
- **Memory Management Unit (MMU):** Handles virtual memory translation, page table management, page fault resolution, and interaction with physical memory and swap devices.
- **Physical Memory Subsystem:** Includes simulated RAM (MEMRAM) and Swap (MEMSWAP) managed by 'memphy' interfaces, and a page replacement mechanism for handling memory pressure.
- **Loader and System Call Interface:** Responsible for loading program definitions, creating PCB instances, and facilitating controlled transitions between user and kernel simulated contexts.

2.2 Component Interaction

The simulator follows a modular architecture where components interact through well-defined interfaces. A process life cycle in the simulator typically proceeds as follows:

1. The **Loader** reads input definitions for processes, initializes PCBs, sets priority, and enqueues them into the scheduler's ready queues.
2. The **Scheduler** selects a ready process according to the MLQ policy and dispatches it onto an available CPU.
3. The **CPU** executes instructions from the dispatched process. When a memory access (read/write) or system call occurs, control is transferred to the corresponding subsystem.
4. The **MMU** handles page translation using either a flat or multi-level paging structure. On a page fault, the MMU interacts with the physical memory subsystem to allocate frames or swap pages in/out.

5. The **Physical Memory Subsystem** provides free frame allocation and supports swap operations. Page replacement decisions update page tables accordingly to reflect physical memory changes.

6. When a process completes execution, the scheduler removes it from the ready queue, releases allocated resources, and the CPU becomes available for other processes.

3 Background Requirements

3.1 Scheduler

In this operating system, the process scheduler is implemented using the Multilevel Queue (MLQ) scheduling policy, tailored for multiprocessor environments. The MLQ design organizes the ready processes into multiple queues, each associated with a distinct priority level. The number of priority levels is defined by the constant `MAX_PRIO`, and each priority level has its own independent ready queue.

When a new program is initiated, the loader is responsible for creating a process and assigning it a Process Control Block (PCB). The process is then inserted into the appropriate ready queue based on its assigned priority, using the `enqueue()` function. This queuing mechanism ensures that processes are grouped and managed according to their priority, facilitating more predictable and controlled CPU scheduling.

CPU scheduling within each priority queue follows the Round-Robin (RR) policy, allowing processes to share CPU time fairly within their respective queues. The `get_proc()` function is responsible for selecting the next process to execute. It implements the MLQ policy by traversing the priority queues in a fixed pattern. Each queue is allocated a fixed number of CPU time slots, determined by the formula `slot = MAX_PRIO - prio`, where prio is the priority of the queue. This allocation mechanism gives higher-priority queues more frequent access to the CPU.

If a queue exhausts its allocated time slots or becomes empty, the scheduler proceeds to the next queue in the sequence. This round-based traversal of queues continues cyclically, ensuring that all processes receive CPU time while respecting their relative priority. The design balances responsiveness for high-priority tasks with fairness across the system, and simplifies implementation by avoiding dynamic queue grouping, as seen in the actual Linux kernel.

This MLQ scheduling model demonstrates a structured and priority-aware method for process management, making it suitable for educational operating systems or simplified kernel prototypes.

Execution Result and Gantt Chart

Figure 1 and Figure 2 show the output of the scheduler in test case `sched_0` when simulating two processes with different priorities (PRIO 4 and PRIO 0). As expected, the process with higher priority (lower PRIO number) is scheduled and finished first.

Table 1 illustrates the Gantt diagram of the same scenario. The process with higher priority (Proc 2) preempts the CPU and runs from time slot 6 to 12, after which Proc 1 resumes execution until it finishes at slot 23.

In summary, MLQ provides a balanced, dynamic scheduling approach suited for diverse systems.

```
Time slot   0
ld_routine
    Loaded a process at input/proc/s0, PID: 1 PRIO: 4
Time slot   1
    CPU 0: Dispatched process  1
Time slot   2
Time slot   3
    CPU 0: Put process  1 to run queue
    CPU 0: Dispatched process  1
Time slot   4
    Loaded a process at input/proc/s1, PID: 2 PRIO: 0
Time slot   5
    CPU 0: Put process  1 to run queue
    CPU 0: Dispatched process  2
Time slot   6
Time slot   7
    CPU 0: Put process  2 to run queue
    CPU 0: Dispatched process  2
Time slot   8
Time slot   9
    CPU 0: Put process  2 to run queue
    CPU 0: Dispatched process  2
Time slot  10
Time slot  11
    CPU 0: Put process  2 to run queue
    CPU 0: Dispatched process  2
Time slot  12
    CPU 0: Processed  2 has finished
    CPU 0: Dispatched process  1
Time slot  13
Time slot  14
    CPU 0: Put process  1 to run queue
    CPU 0: Dispatched process  1
Time slot  15
```

Figure 1: Log Output (p1) of test case `sched_0`

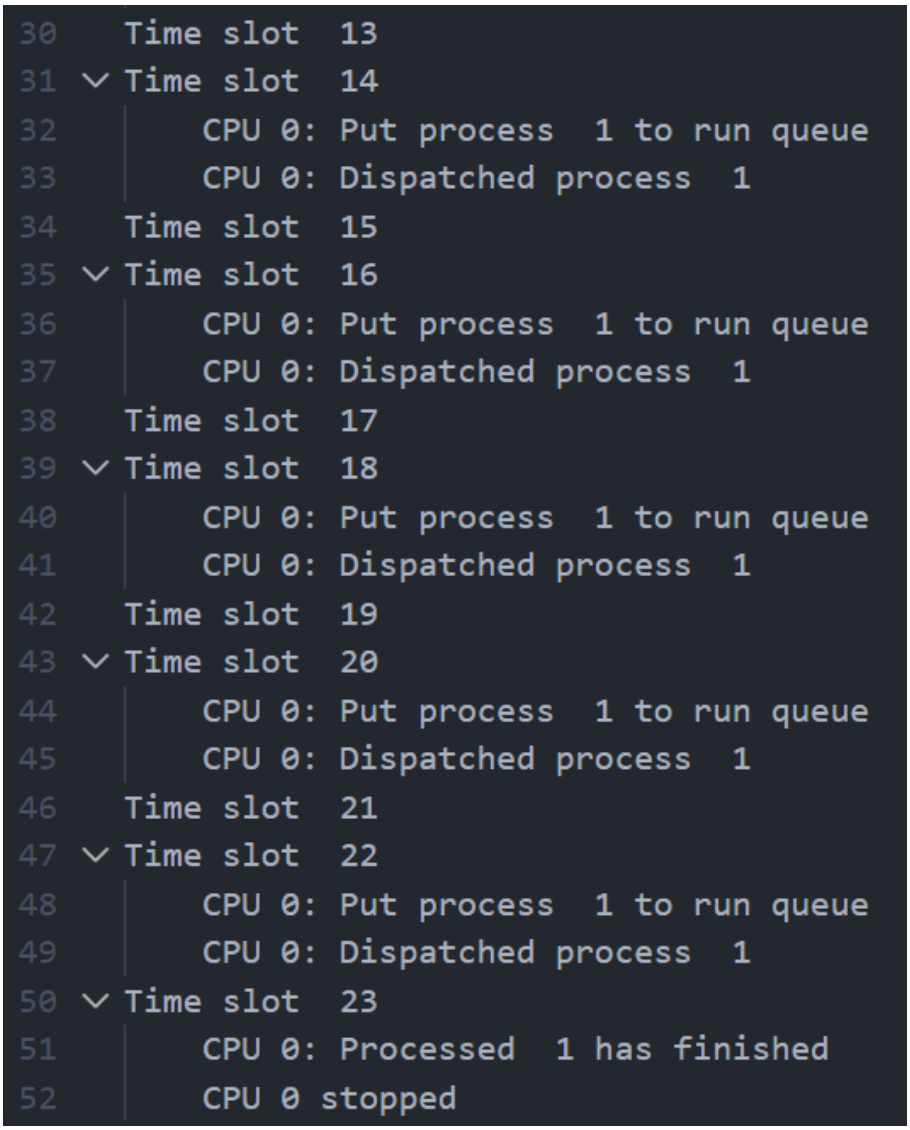

Figure 2: Log Output (p2) of test case `sched_0`

Time Slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
CPU 0	–	P1	P1	P1	P1	P2	P2	P2	P2	P2	P2	P2	P1	P1	P1	P1	P1	P1	P1	P1	P1	P1	P1	finish

Table 1: Gantt chart of MLQ Scheduling

3.2 System Call

System calls provide the fundamental interface between user-space applications and the operating system kernel. In this simulated operating system, system calls are not invoked directly by user programs but are accessed through wrapper functions provided by the user library (e.g., `libmem.c`).

3.2.1 Mechanism Description

The system call mechanism operates by passing arguments via simulated registers (`struct sc_regs`). Each system call is identified by a unique index number.

The process of handling a system call involves the following steps:

- The user process calls a library function (e.g., `liballoc`, `libread`).
- The library function prepares the arguments into a `sc_regs` structure and invokes the `syscall()` interface.
- On the kernel side (implemented in `syscall.c`), the `syscall` function acts as a dispatcher. Based on the system call number (`nr`), it routes the request to the appropriate handler using a switch-case statement or a lookup table (`sys_call_table`).

3.2.2 Implemented System Calls

Based on the current source code, the operating system supports the following key system calls:

1. **listsyscall (Index 0):**
   - **Purpose:** Displays a list of all system calls supported by the operating system.
   - **Implementation:** The handler `__sys_listsyscall` (in `sys_listsyscall.c`) iterates through the system call table and prints the name of each registered syscall.
2. **memmap (Index 17):**
   - **Purpose:** This is the core memory management system call, handling heap allocation, physical memory I/O, and page swapping.
   - **Implementation:** The handler `__sys_memmap` (in `sys_mem.c`) inspects the operation code (`regs->a1`) to determine the specific action:
     - `SYSMEM_MAP_OP`: Maps virtual pages to physical frames.
     - `SYSMEM_INC_OP`: Increases the heap size (adjusting the `sbrk` boundary).
     - `SYSMEM_IO_READ` / `SYSMEM_IO_WRITE`: Performs direct read/write operations on the physical RAM via `MEMPHY_read`/`MEMPHY_write`.
     - `SYSMEM_SWP_OP`: Executes page swapping between RAM and swap devices.

3.2.3 Security and User/Kernel Separation

A critical requirement in operating system design is the isolation of kernel memory from user processes. In this simulation, this separation is enforced through the mechanism of passing Process IDs (PIDs) rather than direct PCB pointers.

- **Problem:** If User Space were allowed to pass a direct pointer to `struct pcb_t` into the Kernel, a malicious program could pass a forged address to access unauthorized memory.
- **Solution:** Library functions only transmit the `PID` (Process ID) to the Kernel.
- **Implementation:** Inside `sys_mem.c`, the kernel uses the helper function `find_proc_safe(krnl, pid)` to validate the request. It searches for the corresponding PCB in the kernel's own `running_list` or `ready_queue`. The operation proceeds only if a valid, kernel-managed PCB is found, ensuring system integrity.

3.3 Memory Management and Paging

In this simple operating system simulation, memory management is one of the core modules, responsible for allocating and maintaining memory usage for concurrent processes while ensuring isolation and efficient utilization of physical memory resources. To achieve this, the system adopts a virtual memory model combined with a paging-based memory management scheme, which is commonly used in modern operating systems like Linux.

3.3.1 The Virtual Memory Mapping in Each Process

Each process in the system has its own virtual address space, which is managed through a list of contiguous virtual memory areas (`vm_area_struct`). These areas represent segments such as code, heap, or stack, depending on their use. Although the virtual address space appears continuous to the process, it is actually backed by discrete memory frames in physical RAM.

Each virtual memory area is defined by its `[vm_start, vm_end]` boundaries. The current heap boundary is tracked by the sbrk value. Within the area between vm start and sbrk, the OS manages a list of memory regions (`vm_rg_struct`) representing allocated chunks and a list of free regions (`vm_freerg_list`) to support dynamic allocation (`alloc`) and deallocation (`free`) operations during runtime.

The overall virtual memory of a process is encapsulated by the `mm_struct` structure in `os-mm.h`, which contains:

- **pgd**: the page table directory that handles address translation
- **mmap**: a pointer to the list of vm areas
- **symgrtbl**: a fixed-size symbol table used for managing variable memory regions
- **fifo_pgn**: a list to support page replacement policies like FIFO.

Otherwise, if we turn on the macro for 64-bit configuration in `os-cfg.h`, we will have:

- **pgd**: page global directory
- **p4d**: page level 4 directory
- **pud**: page upper directory
- **pmd**: page middle directory
- **pt**: page table.

3.3.2 The System's Physical Memory

The physical memory is implemented using two types of devices: RAM (main memory) and SWAP (secondary memory). RAM is directly accessible by the CPU and is used for storing active memory pages. When RAM becomes full, the system swaps out pages to the SWAP device, which can be larger but is slower and not directly accessible by the CPU.

Both RAM and SWAP are simulated using the `memphy_struct` structure `os-mm.h`. This includes:

- **storage**: a pointer to the actual memory array
- **maxsz**: the maximum size of the memory device
- **rdmflg**: a flag indicating whether access is random or sequential
- **free_fp_list and used_fp_list**: linked lists that track free and used memory frames.

The system allows configuration of one RAM device and up to four SWAP devices. This mimics real-world systems where RAM is expensive and fast, while SWAP is cheaper but slower, used mainly to extend the usable memory space. The use of paging allows each process to have isolated and flexible memory allocation, with frames from RAM and SWAP being assigned dynamically.

3.3.3 Paging-Based Address Translation Scheme

Address translation in this system is based on a single-level paging scheme. Each logical address generated by the CPU is split into:

- **Page number**: used to look up the physical frame in the page table
- **Page offset**: added to the frame's base address to obtain the actual physical address.

  Each page table entry (PTE) is a 32-bit value that contains:

- Frame number (bits 0–12)
- Swapped indicator (bit 30)
- Page number Present bit (bit 31)
- Other metadata like dirty bit, swap offset, and swap type.

Each process maintains its own page table with thousands of entries (e.g., 16,000 for 22-bit address with 256B pages), ensuring memory space isolation. When memory is accessed and the required page is not present in RAM, a page fault occurs. The OS must handle this by swapping in the required page from SWAP, potentially swapping out another page if RAM is full.

This dynamic and demand-based allocation allows efficient use of limited physical memory while providing the illusion of a large continuous memory space to each process.

3.3.4 64-bit Addressing and Multi-level Paging Extension

In addition to the original 32-bit single-level paging model, this version extends the simulator to support a 64-bit virtual addressing scheme with **multi-level page tables**. The goal is to model how modern x86-64 systems scale virtual memory, where a single flat page table would be prohibitively large. To reduce page-table memory overhead, multi-level page tables can be combined with sparse allocation or demand allocation (only allocating page-table structures for actually-used virtual regions).

5-level paging structure (x86-64 style)

The 64-bit scheme is organized as a 5-level page table hierarchy, where each level index is extracted from the virtual address and used to traverse the corresponding directory. In particular:

- Level 5: Page Global Directory (PGD) uses bits 56–48
- Level 4: Page Level-4 Directory (P4D) uses bits 47–39
- Level 3: Page Upper Directory (PUD) uses bits 38–30
- Level 2: Page Middle Directory (PMD) uses bits 29–21
- Level 1: Page Table (PT) uses bits 20–12
- Page offset uses bits 11–0 (4KB page offset)

This hierarchical split expands the theoretical virtual address space (up to 128 PiB) and matches the common 4KB page organization.

Address translation (conceptual pipeline)

Given a virtual address VA, the MMU performs:

1. Extract indices: `pgd_idx`, `p4d_idx`, `pud_idx`, `pmd_idx`, `pt_idx` from the corresponding bit ranges.
2. Traverse directories: PGD → P4D → PUD → PMD → PT.
3. At the leaf PT entry, obtain the target physical frame number if present; otherwise, trigger a page fault and perform swapping/mapping similarly to the paging workflow described for the 32-bit mode.

This is consistent with the paging-based memory pipeline where pages may be swapped between RAM and SWAP and page tables are updated to reflect physical frame residency.

Module placement and implementation note

All core virtual-memory structures are supported in the memory-management modules, and the assignment explicitly states that 64-bit address support is placed in `mm64.c`. In the provided implementation skeleton, the page table root for 64-bit mode is represented with a 64-bit entry type (e.g., allocating the PGD as an array of `uint64_t` entries when `MM64` is enabled).

Design choice: sparse / demand allocation

Because a 64-bit flat page table would consume excessive memory, the simulator should allocate intermediate page-table levels on demand. This minimizes memory usage by only creating directories for active virtual regions, trading off extra traversal steps for lower storage overhead.

With the 64-bit multi-level paging extension, the simulator models a realistic hierarchy of page-table directories, while preserving the original paging workflow (mapping, swapping, and page-table updates) described in the memory subsystem.

3.3.5 Wrapping-Up All Paging-Oriented Implementations

To support modular and configurable memory features, the system uses compile-time macros defined in os-cfg.h, such as:

```
#define MM_PAGING
#define MM_FIXED_MEMSZ
#define MM64 1
```

These settings allow enabling or disabling features like paging or fixed memory sizes. For instance, `MM_FIXED_MEMSZ` ensures compatibility with legacy input files, while disabling it allows dynamic configuration of RAM and SWAP sizes through input files.

When `MM_PAGING` is enabled, additional fields related to memory management are added to the pcb t structure, including pointers to `mm_struct`, `memphy_struct` for RAM, and SWAP devices.

If `MM64` is defined, `mm_struct` will have the fields: `pgd`, `p4d`,... as we mentioned above and allow us to work with multi-level paging.

Multiple memory modules work together to manage memory:

- **libmem.c**: provides the interface for memory operations (alloc, free, read, write).
- **mm-vm.c**: handles virtual memory area mapping and allocation.
- **mm-memphy.c** manages physical memory frame allocation and I/O.
- **mm64.c** : coordinates the overall memory logic including page table updates and swapping.

Basic memory operations:

- **ALLOC**: requests a chunk of memory. If no free region is found, it triggers a system call to expand the heap and map new pages.
- **FREE**: adds the released region back to the free list for future use (physical frames are not immediately reclaimed to avoid fragmentation).
- **READ/WRITE**: involves checking if the page is present, swapping in if necessary, and accessing the data. These operations may trigger multiple internal mechanisms (e.g., memory swapping, I/O access, page table updates).

By implementing and integrating these components, the system provides a complete simulation of how memory management works in a real OS, helping students understand the challenges and mechanisms involved in virtual memory and paging systems.

4 Implementation

The source code was implemented in C language across several files, each corresponding to a specific module of the operating system. Below is a brief overview of the implementation:

4.1 Scheduler

The Scheduler is responsible for determining the execution order of processes and allocating CPU time based on priority. In this assignment, we implemented a Multi-Level Queue (MLQ) scheduler with a priority-based time slicing mechanism. This design ensures that high-priority processes receive more CPU resources while preventing starvation for lower-priority ones through a defined slot quota system.

4.1.1 Core Data Structures

To implement the MLQ algorithm efficiently, we utilized the following key data structures in `sched.c`:

- **Priority Queues**: An array of queues, where each index corresponds to a specific priority level. This allows for $O(1)$ access to the ready queue of any given priority.

```
static struct queue_t mlq_ready_queue[MAX_PRIO];
```

- **Time Slot Management**:
  - **slot**: An array specifying the time slots for each priority level, with higher priorities receiving more slots.

```
        static int slot[MAX_PRIO];
```

  - **current_slot**: An array tracking the number of slots actually consumed by each priority queue in the current round.

```
        static int current_slot[MAX_PRIO];
```

  - **current_prior**: An integer tracking the current priority level being served, enabling the scheduler to resume its search from where it left off (stateful behavior).

```
        static int current_prior = 0;
```

- **Synchronization**: A mutex lock is employed to ensure thread safety, protecting shared scheduler data structures from race conditions in a multi-CPU simulation.

```
static pthread_mutex_t queue_lock;
```

4.1.2 Initialization

The initialization process, handled in `init_scheduler()`, sets up the ready queues and assigns time slot quotas. The quota allocation follows an inverse relationship with priority values:

```
slot[i] = MAX_PRIO - i;
```

Since a lower priority value indicates higher importance in this system (e.g., priority 0 is the highest), this formula assigns a larger number of time slots to high-priority processes, granting them more CPU time before the scheduler yields to other queues.

4.1.3 Process Selection Algorithm

1. **Queue Traversal**: The scheduler iterates through priority levels starting from `current_prior`.
2. **Selection Criteria**: A process is selected from the current priority queue if the queue is not empty and has not exhausted its allocated time slots.
3. **Dispatching**: If a candidate is found, it is dequeued, and the `current_slot` counter is incremented.
4. **Round Reset**: If the scheduler scans through all priority levels without finding a runnable process that fits the quota, it implies a new scheduling round is needed. All `current_slot` counters are reset.

```
struct pcb_t * get_mlq_proc(void) {
    struct pcb_t * proc = NULL;
    pthread_mutex_lock(&queue_lock);
    int processed_priorities = 0;
    while(processed_priorities < MAX_PRIO)
    {
        if(!empty(&mlq_ready_queue[current_prior]) &&
            current_slot[current_prior] < slot[current_prior])
        {
            proc = dequeue(&mlq_ready_queue[current_prior]);
            current_slot[current_prior]++;
            if(proc) enqueue(&running_list, proc);
            pthread_mutex_unlock(&queue_lock);
            return proc;
        }
        processed_priorities++;
        current_prior++;
        if(current_prior >= MAX_PRIO) current_prior = 0;
    }
    current_prior = 0;
    for(int i = 0; i < MAX_PRIO; i++)
```

```
    {
        current_slot[i] = 0;
    }
    pthread_mutex_unlock(&queue_lock);
    return NULL;
}
```

4.1.4 Process Management Functions

The lifecycle of a process within the scheduler is managed by `put_mlq_proc()` and `add_mlq_proc()`.

- `put_mlq_proc()`: Invoked when a process finishes its time slice. It removes the process from the `running_list` (using `purgequeue`) and enqueues it back to the appropriate priority queue.

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

4.1.5 Integration with OS Core

The scheduler is tightly integrated with other OS components. The CPU simulation calls `get_proc()` to fetch instructions. By maintaining the `running_list`, the scheduler enables the memory management unit to safely validate and retrieve the calling process's PCB during system calls, ensuring strict separation between user and kernel spaces.

4.2 System Call

The system call interface serves as the secure gateway for user processes to access kernel services. The implementation is divided into the central dispatcher, specific handlers, and the registration mechanism.

4.2.1 System Call Architecture

The core architecture is implemented in `syscall.c`, acting as a multiplexer for all incoming kernel requests.

- **Dispatcher:** The function `syscall()` serves as the central entry point. It accepts the system call number (`nr`), the caller's PID, and the register state (`regs`).
- **Routing Logic:** Instead of a manual switch-case, the dispatcher uses C preprocessor macros to generate the routing logic dynamically from `syscalltbl.lst`. This ensures that the dispatch table and function prototypes remain consistent.
- **Default Handler:** Any unimplemented or undefined system call number is routed to `__sys_ni_syscall`, which safely returns without performing any action.

```
1 int syscall(struct krnl_t *krnl, uint32_t pid, uint32_t nr, struct sc_regs*
  ↪  regs)
2 {
3     switch (nr) {
4     #include "syscalltbl.lst"
5     default: return __sys_ni_syscall(krnl, regs);
6     }
7 };
```

4.2.2 Key System Call Handlers

We have implemented specific handlers to support process management and memory operations:

1. **Memory Management Handler (`memmap`):** Located in `sys_mem.c`, the `__sys_memmap` function is the most complex handler. It acts as a sub-dispatcher for memory operations based on the opcode in register `a1`:
   - **Heap Management (`SYSMEM_INC_OP`):** Invokes `inc_vma_limit` to adjust the process's break pointer (`sbrk`).
   - **Page Swapping (`SYSMEM_SWP_OP`):** Calls `__mm_swap_page` to handle data movement between RAM and Swap devices.
   - **Physical I/O (`SYSMEM_IO_READ/WRITE`):** Allows direct read/write access to the simulated physical RAM (`mram`) via `MEMPHY_read` and `MEMPHY_write`.

- **Security Enforcement:** A helper function, `find_proc_safe(krnl, pid)`, is used to resolve the `pcb_t` pointer from the provided PID. This ensures that the kernel operates on valid process structures managed by the scheduler, preventing illegal access via raw pointers from user space.

```
int __sys_memmap(struct krnl_t *krnl, uint32_t pid, struct sc_regs*
↪ regs)
{
    if (!krnl) return -1;
    int memop = regs->a1;
    int ret = 0;
    struct pcb_t *caller = NULL;
    addr_t paddr = regs->a2;
    switch (memop) {
    case SYSMEM_MAP_OP:
    case SYSMEM_INC_OP:
    case SYSMEM_SWP_OP:
            caller = find_proc_safe(krnl, pid);
            if (!caller) return -1;
            if (memop == SYSMEM_MAP_OP) ret = vmap_pgd_memset(caller,
            ↪  regs->a2, regs->a3);
            else if (memop == SYSMEM_INC_OP) ret =
            ↪  inc_vma_limit(caller, regs->a2, regs->a3);
            else ret = __mm_swap_page(caller, regs->a2, regs->a3);
            break;
    case SYSMEM_IO_READ:
            if (krnl->mram) {
                BYTE data;
                if (MEMPHY_read(krnl->mram, paddr, &data) < 0)
                    ret = -1;
                else {
                    if (caller && regs->a3 < 10) {
                        caller->regs[regs->a3] = data;
                    }
                }
            } else ret = -1;
            break;
    case SYSMEM_IO_WRITE:
            if (krnl->mram) {
                BYTE data = (BYTE)regs->a3;
                if (MEMPHY_write(krnl->mram, paddr, data) < 0)
                    ret = -1;
            } else ret = -1;
```

```
                break;
        default:
                return -1;
        }
        return ret;
    }
```

2. **System Information Handler (`listsyscall`):** Implemented in `sys_listsyscall.c`, this handler iterates through the `sys_call_table` and prints the names of all registered system calls. This is useful for verifying the system configuration and debugging.

```
int __sys_listsyscall(struct krnl_t *krnl, uint32_t pid, struct
↪  sc_regs* reg)
{
   for (int i = 0; i < syscall_table_size; i++)
       printf("%s\n",sys_call_table[i]);
   return 0;
}
```

4.2.3 System Call Registration

To maintain modularity, system calls are registered in a separate list file (`syscalltbl.lst`). The implementation in `syscall.c` uses macro expansion techniques:

- **Macro `__SYSCALL`:** This macro is defined twice. First, to generate external function declarations for all handlers. Second, to generate the `case` statements inside the `syscall()` switch block.
- **Benefit:** This design allows adding new system calls by simply adding a single line to the list file, automatically updating both the dispatch logic and the lookup table.

4.3 Memory Management Unit & Multi-level Paging

The Memory Management Unit (MMU), implemented in `mm64.c` and `mm-memphy.c`, is responsible for handling the 5-level paging mechanism, virtual-to-physical address translation, and frame management. The system is designed to support a vast 64-bit address space efficiently through dynamic allocation strategies.

4.3.1 Core Data Structures

To support the 64-bit architecture, the system utilizes the following key data structures:

- **Memory Management Structure (mm_struct)**:This structure serves as the root of the memory management system for each process. It contains a pointer to the Page Global Directory (PGD), which is the top level of the 5-level paging hierarchy, and a list of virtual memory areas (mmap).
- **5-Level Page Tables**: The system defines a hierarchical paging structure consisting of PGD (Level 5), P4D (Level 4), PUD (Level 3), PMD (Level 2), and PT (Level 1). Each entry in these tables is a 64-bit address (addr_t), allowing for a deep and scalable translation tree.
- **Frame Structure (framephy_struct)**: A linked list data structure used to track physical frames allocated in RAM. Each node contains the Frame Page Number (FPN) and a pointer to the next frame.

```
struct mm_struct {
#ifdef MM64
   uint64_t *pgd;
   uint64_t *p4d;
   uint64_t *pud;
   uint64_t *pmd;
   uint64_t *pt;
#else
   uint32_t *pgd;
#endif
   struct vm_area_struct *mmap;
   struct vm_rg_struct symrgtbl[PAGING_MAX_SYMTBL_SZ];
   struct pgn_t *fifo_pgn;
};

struct framephy_struct {
   addr_t fpn;
   struct framephy_struct *fp_next;
   struct mm_struct* owner;
};
```

4.3.2 Initialization strategy

The memory management system is initialized using the `init_mm()` function, which sets up the page directory and creates an initial virtual memory area (VMA) for the process.

```
int init_mm(struct mm_struct *mm, struct pcb_t *caller)
{
    struct vm_area_struct * vma = malloc(sizeof(struct vm_area_struct));
    mm->pgd = (addr_t*)malloc(512 * sizeof(addr_t));
    __sync_fetch_and_add(&total_pgtbl_size, 512 * sizeof(addr_t));
    memset(mm->pgd, 0, 512 * sizeof(addr_t));

    mm->p4d = NULL;
    mm->pud = NULL;
    mm->pmd = NULL;
    mm->pt = NULL;

    vma->vm_id = 0;
    vma->vm_start = 0;
    vma->vm_end = BIT_ULL(PAGING_CPU_BUS_WIDTH);
    vma->sbrk = vma->vm_start;

    vma->vm_freerg_list = NULL;
    vma->vm_next = NULL;
    vma->vm_mm = mm;
    mm->mmap = vma;

    for (int i=0; i<PAGING_MAX_SYMTBL_SZ; i++) {
        mm->symrgtbl[i].rg_start = 0;
        mm->symrgtbl[i].rg_end = 0;
        mm->symrgtbl[i].rg_next = NULL;
    }
    return 0;
}
```

4.3.3 Page Table Entry Management & Dynamic Walking

The core logic of the MMU resides in the `__get_pte()` function, which implements the page walk through the 5 levels.

- **On-demand Allocation**: The function checks for the existence of intermediate page tables (P4D, PUD, PMD, PT) during traversal. If a table is missing, it dynamically allocates memory using `malloc` and initializes it with `memset`. This Dynamic Allocation strategy ensures that memory is consumed only for the address ranges actually used by the process, significantly reducing storage overhead compared to a static allocation approach.

- **Statistics Tracking**: The implementation includes counters `total_pgtbl_size` and `memory_access_count` within the page walk logic to measure the storage cost and access overhead of the multi-level paging system.

```c
static unsigned long total_pgtbl_size = 0;
static unsigned long memory_access_count = 0;

static addr_t *__get_pte(struct mm_struct *mm, addr_t pgn, int alloc) {
    if (!mm || !mm->pgd) return NULL;

    __sync_fetch_and_add(&memory_access_count, 5);
    int pgd_idx = (pgn >> 36) & 0x1FF;
    int p4d_idx = (pgn >> 27) & 0x1FF;
    int pud_idx = (pgn >> 18) & 0x1FF;
    int pmd_idx = (pgn >> 9)  & 0x1FF;
    int pt_idx  = (pgn >> 0)  & 0x1FF;

    /* Level 5: PGD */
    if (mm->pgd[pgd_idx] == 0) {
        if (!alloc) return NULL;
        mm->pgd[pgd_idx] = (addr_t)malloc(512 * sizeof(addr_t));
        memset((void *)mm->pgd[pgd_idx], 0, 512 * sizeof(addr_t));
        __sync_fetch_and_add(&total_pgtbl_size, 512 * sizeof(addr_t));
    }
    addr_t *p4d = (addr_t *)mm->pgd[pgd_idx];

    /* Level 4: P4D */
    if (p4d[p4d_idx] == 0) {
        if (!alloc) return NULL;
        p4d[p4d_idx] = (addr_t)malloc(512 * sizeof(addr_t));
        memset((void *)p4d[p4d_idx], 0, 512 * sizeof(addr_t));
        __sync_fetch_and_add(&total_pgtbl_size, 512 * sizeof(addr_t));
    }
    addr_t *pud = (addr_t *)p4d[p4d_idx];

    /* Level 3: PUD */
    if (pud[pud_idx] == 0) {
        if (!alloc) return NULL;
        pud[pud_idx] = (addr_t)malloc(512 * sizeof(addr_t));
        memset((void *)pud[pud_idx], 0, 512 * sizeof(addr_t));
        __sync_fetch_and_add(&total_pgtbl_size, 512 * sizeof(addr_t));
    }
    addr_t *pmd = (addr_t *)pud[pud_idx];
```

```

    /* Level 2: PMD */
    if (pmd[pmd_idx] == 0) {
        if (!alloc) return NULL;
        pmd[pmd_idx] = (addr_t)malloc(512 * sizeof(addr_t));
        memset((void *)pmd[pmd_idx], 0, 512 * sizeof(addr_t));
        __sync_fetch_and_add(&total_pgtbl_size, 512 * sizeof(addr_t));
    }
    addr_t *pt = (addr_t *)pmd[pmd_idx];

    /* Level 1: PT - Return pointer to the PTE entry */
    return &pt[pt_idx];
}
```

4.3.4 Memory Mapping and Frame Allocation

The mapping between virtual pages and physical frames is handled by:

- **alloc_pages_range()**: Allocates a contiguous sequence of free physical frames from the RAM device (mram) by retrieving nodes from the free_fp_list.
- **vmap_page_range()**: Iterates through a range of virtual addresses, retrieves the corresponding Page Table Entry (PTE) using `__get_pte()`, and updates it with the allocated Frame Page Number (FPN). It also enlists the mapped pages into a `fifo_pgn` list to support page replacement algorithms.

```
addr_t alloc_pages_range(struct pcb_t *caller, int req_pgnum, struct
↪  framephy_struct **frm_lst)
{
    int pgit, count = 0;
    struct framephy_struct *newfp_str;
    addr_t fpn;

    *frm_lst = NULL;

    for(pgit = 0; pgit < req_pgnum; pgit++) {
        if(MEMPHY_get_freefp(caller->krnl->mram, &fpn) == 0) {
            newfp_str = (struct framephy_struct*)malloc(sizeof(struct
            ↪  framephy_struct));
            newfp_str->fpn = fpn;
            newfp_str->fp_next = *frm_lst;
            *frm_lst = newfp_str;
            count++;
        } else {
```

```
            /* Rollback if failed */
            while(*frm_lst) {
                struct framephy_struct *tmp = *frm_lst;
                *frm_lst = (*frm_lst)->fp_next;
                MEMPHY_put_freefp(caller->krnl->mram, tmp->fpn);
                free(tmp);
            }
            return -3000; /* Out of memory */
        }
    }
    return 0;
}

addr_t vmap_page_range(struct pcb_t *caller,
                    addr_t addr,
                    int pgnum,
                    struct framephy_struct *frames,
                    struct vm_rg_struct *ret_rg)
{
    struct framephy_struct *fpit = frames;
    int pgit = 0;
    addr_t pgn = PAGING_PGN(addr);

    if (ret_rg) {
        ret_rg->rg_start = addr;
        ret_rg->rg_end = addr + pgnum * PAGING64_PAGESZ;
    }

    for (pgit = 0; pgit < pgnum; pgit++) {
       if (fpit == NULL) break;
       pte_set_fpn(caller, pgn + pgit, fpit->fpn);
       enlist_pgn_node(&caller->mm->fifo_pgn, pgn + pgit);
       fpit = fpit->fp_next;
    }
    return 0;
}
```

4.3.5 Swap and Debugging Utilities

The MMU includes utilities for swapping pages and debugging:

- **__swap_cp_page()**: Copies page content from a source frame to a destination frame.
- **print_pgtbl()**: For inspecting page tables.

```c
int __swap_cp_page(struct memphy_struct *mpsrc, addr_t srcfpn,
                struct memphy_struct *mpdst, addr_t dstfpn)
{
  int cellidx;
  addr_t addrsrc, addrdst;
  for(cellidx = 0; cellidx < PAGING64_PAGESZ; cellidx++)
  {
    addrsrc = srcfpn * PAGING64_PAGESZ + cellidx;
    addrdst = dstfpn * PAGING64_PAGESZ + cellidx;

    BYTE data;
    MEMPHY_read(mpsrc, addrsrc, &data);
    MEMPHY_write(mpdst, addrdst, data);
  }
  return 0;
}

int print_pgtbl(struct pcb_t *caller, addr_t start, addr_t end)
{
    printf("--- PCB %d Page Table ---\n", caller->pid);
    if (caller && caller->mm && caller->mm->pgd) {
        /* Start traversal from Level 5 (PGD) */
        print_pgtbl_recursive((void *)caller->mm->pgd, 5, 0);
    }
    printf("----------------------------------------\n");
    return 0;
}
```

4.3.6 Integration with OS Core

The MMU integrates with the OS core as follows:

- **Process Control Block (PCB)**: Each process (`pcb_t`) contains an `mm_struct` for its memory management, including the page directory and VMA list.
- **Memory Device**: The `memphy_struct` abstracts RAM and swap storage, with functions like `MEMPHY_read()` and `MEMPHY_write()` for data access.
- **Scheduler**: The MMU supports the scheduler by ensuring processes have access to mapped memory, with page faults triggering frame allocation or swapping.

4.4 Virtual Memory Module

The Virtual Memory Module (`mm-vm.c`) manages the high-level organization of the process's address space, specifically the Virtual Memory Areas (VMAs).

4.4.1 Core Data Structures

The virtual memory module relies on data structures defined in the memory management unit, with additional focus on VMAs and memory regions:

- **vm_area_struct**: Represents a contiguous segment of virtual memory (e.g., the Data segment). It tracks the current heap boundary (`sbrk`) and maintains a list of free memory regions (`vm_freerg_list`) for reuse.
- **vm_rg_struct**: Represents a specific allocated or free region within a VMA, defined by `rg_start` and `rg_end`.

```
struct vm_area_struct {
   unsigned long vm_id;
   addr_t vm_start;
   addr_t vm_end;
   addr_t sbrk;
/*
 * Derived field
 * unsigned long vm_limit = vm_end - vm_start
 */
   struct mm_struct *vm_mm;
   struct vm_rg_struct *vm_freerg_list;
   struct vm_area_struct *vm_next;
};

struct vm_rg_struct {
   addr_t rg_start;
   addr_t rg_end;

   struct vm_rg_struct *rg_next;
};
```

4.4.2 VMA Retrieval

The module provides a function to retrieve a VMA by its identifier, enabling access to specific memory areas.

- **get_vma_by_num()**: Retrieves the VMA with the specified ID from the process's memory management structure.

```c
struct vm_area_struct *get_vma_by_num(struct mm_struct *mm, int vmaid)
{
  if (mm == NULL) return NULL;
  struct vm_area_struct *pvma = mm->mmap;
  if (mm->mmap == NULL)
    return NULL;
  int vmait = pvma->vm_id;
  while (vmait < vmaid)
  {
    if (pvma == NULL)
      return NULL;
    pvma = pvma->vm_next;
    vmait = pvma->vm_id;
  }
  return pvma;
}
```

4.4.3 Memory Region Allocation

The module supports dynamic expansion of the process's data segment:

- **inc_vma_limit()**: When the free list cannot satisfy an allocation request, this function is called to expand the heap. It calculates the number of required pages, invokes `vm_map_ram()` to map new physical frames, and advances the sbrk pointer.
- **get_vm_area_node_at_brk()**: Creates a new memory region metadata node starting at the current break point, ensuring contiguous allocation.

```c
int inc_vma_limit(struct pcb_t *caller, int vmaid, addr_t inc_sz)
{
  struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);
  if(cur_vma == NULL){
    return -1;
  }
  addr_t aligned_inc_sz = PAGING_PAGE_ALIGNSZ(inc_sz);
  int incnumpage = aligned_inc_sz / PAGING_PAGESZ;
  addr_t old_sbrk = cur_vma->sbrk;
  addr_t new_sbrk = old_sbrk + aligned_inc_sz;

  if(new_sbrk > cur_vma->vm_end){
    return -1;
  }

```

```
  struct vm_rg_struct *newrg = malloc(sizeof(struct vm_rg_struct));
  newrg->rg_start = old_sbrk;
  newrg->rg_end = new_sbrk;
  newrg->rg_next = NULL;

  if(vm_map_ram(caller, cur_vma->vm_start, cur_vma->vm_end, old_sbrk,
  ↪  incnumpage, newrg) < 0){
    free(newrg);
    return -1;
  }
  cur_vma->sbrk = new_sbrk;
  return 0;
}

struct vm_rg_struct *get_vm_area_node_at_brk(struct pcb_t *caller, int vmaid,
↪  addr_t size, addr_t alignedsz)
{
  struct vm_rg_struct * newrg;
  struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);
  if(cur_vma == NULL || cur_vma->sbrk + alignedsz > cur_vma->vm_end){
    return NULL;
  }
  newrg = malloc(sizeof(struct vm_rg_struct));
  if (newrg == NULL) {
    return NULL;
  }
  newrg->rg_start = cur_vma->sbrk;
  newrg->rg_end = newrg->rg_start + alignedsz;
  newrg->rg_next = NULL;
  return newrg;
}
```

4.4.4 Memory Region Validation

To ensure data integrity, the function `validate_overlap_vm_area()` checks if a requested memory region overlaps with any existing regions in the VMA before proceeding with allocation.

```
int validate_overlap_vm_area(struct pcb_t *caller, int vmaid, addr_t vmastart,
↪  addr_t vmaend)
{
  if (vmastart >= vmaend) return -1;
  struct vm_area_struct *vma = caller->mm->mmap;
  if (vma == NULL)
```

```
  {
    return 0;
  }
  while(vma != NULL){
    if(vma->vm_id == vmaid){
      vma = vma->vm_next;
      continue;
    }
    if(OVERLAP(vmastart, vmaend, vma->vm_start, vma->vm_end)){
      return -1;
    }
    vma = vma->vm_next;
  }
  return 0;
}
```

4.4.5 Page Swapping

The module supports swapping pages between RAM and swap storage to manage memory efficiently:

- **__mm_swap_page()**: Swaps a page from a victim frame in RAM to a swap frame, using the `swap_cp_page()` function from the MMU.

```
int __mm_swap_page(struct pcb_t *caller, addr_t vicfpn , addr_t swpfpn)
{
  __swap_cp_page(caller->krnl->mram, vicfpn, caller->krnl->active_mswp,
  ↪  swpfpn);
  return 0;
}
```

4.4.6 Integration with OS Core

The virtual memory module integrates with the OS core as follows:

- **Process Control Block (PCB)**: Each process (`pcb_t`) contains a memory management structure (`mm_struct`) with a list of VMAs, accessed via `krnl->mm`.
- **Memory Management Unit**: The module uses MMU functions like `vm_map_ram()` and `__swap_cp_page()` for memory mapping and swapping.
- **Scheduler**: The module supports the scheduler by allocating memory for processes and handling page faults through swapping or allocation.

4.5 Memory Module Library

The Memory Module Library in `libmem.c` provides core memory management functionality including allocation, deallocation, page management, and read/write operations. It implements paging-based virtual memory with support for swapping and process memory isolation.

4.5.1 Core Data Structures

The library uses several key data structures for memory management:

- **Memory Management Structure (mm_struct)**: Contains 5-level paging scheme, free page list, and symbol table.
- **Page Node (pgn_t)**: FIFO page replacement tracking.

```
struct mm_struct {
#ifdef MM64
   uint64_t *pgd;
   uint64_t *p4d;
   uint64_t *pud;
   uint64_t *pmd;
   uint64_t *pt;
#else
   uint32_t *pgd;
#endif
   struct vm_area_struct *mmap;
   /* Currently we support a fixed number of symbol */
   struct vm_rg_struct symrgtbl[PAGING_MAX_SYMTBL_SZ];
   /* list of free page */
   struct pgn_t *fifo_pgn;
};

struct pgn_t {
   addr_t pgn;
   struct pgn_t *pg_next;
};
```

4.5.2 Synchronization

To ensure thread safety in a multi-core environment, critical sections within the library (such as accessing the free list or symbol table) are protected using a global mutex `mmvm_lock`.

```
static pthread_mutex_t mmvm_lock = PTHREAD_MUTEX_INITIALIZER;
```

4.5.3 Memory Allocation Strategy

liballoc() / __alloc(): The allocation logic follows a First-Fit strategy. It first attempts to find a suitable block in the `vm_freerg_list` using `get_free_vmrg_area()`. If no block is found, it invokes the `SYSMEM_INC_OP` system call to request the kernel to expand the heap.

```
int liballoc(struct pcb_t *proc, addr_t size, uint32_t reg_index)
{
  addr_t addr;
  int val = __alloc(proc, 0, reg_index, size, &addr);
  if (val == -1)
  {
    printf("liballoc:-1\n");
    return -1;
  }
  addr_t offset = addr - proc->mm->mmap->vm_start;
  printf("liballoc:%llu\n", (unsigned long long)offset);

#ifdef IODUMP
#ifdef PAGETBL_DUMP
  print_pgtbl(proc, 0, -1);
#endif
#endif
  return val;
}
```

4.5.4 Deallocation and Merging

- **libfree() / __free()**: This function releases a memory region. A robust validation check ensures that the region address is valid (non-zero).
- **Fragmentation Handling**: Upon deallocation, the code not only adds the region to the free list but also performs Sorting (by address) and Merging of adjacent free regions. This advanced logic helps reduce external fragmentation within the heap.

```
int libfree(struct pcb_t *proc, uint32_t reg_index)
{
  struct vm_rg_struct *rg = get_symrg_byid(proc->mm, reg_index);
  addr_t offset = 0;
  if (rg) {
    offset = rg->rg_start - proc->mm->mmap->vm_start;
  }
  int val = __free(proc, 0, reg_index);
  if (val == -1)
```

```
  {
    printf("libfree:-1\n");
    return -1;
  }
  printf("libfree:%llu\n", (unsigned long long)offset);
#ifdef IODUMP
#ifdef PAGETBL_DUMP
  print_pgtbl(proc, 0, -1);
#endif
#endif
  return val;
}
```

4.5.5 I/O Operations

The **libread()** and **libwrite()** functions provide memory access. Instead of accessing the physical RAM array directly (which is prohibited for user space), these functions translate virtual offsets to physical addresses and invoke `SYSMEM_IO_READ` or `SYSMEM_IO_WRITE` system calls to perform the actual data transfer.

```
int libread(
    struct pcb_t *proc,
    uint32_t source,
    addr_t offset,
    uint32_t* destination)
{
  BYTE data;
  int val = __read(proc, 0, source, offset, &data);
  printf("libread:%llu\n",
       (unsigned long long)offset);
  *destination = data;
#ifdef IODUMP
#ifdef PAGETBL_DUMP
  print_pgtbl(proc, 0, -1);
#endif
#endif

  return val;
}

int libwrite(
    struct pcb_t *proc,
    BYTE data,
```

```
    uint32_t destination,
    addr_t offset)
{
  int val = __write(proc, 0, destination, offset, data);
  printf("libwrite:%llu\n",
       (unsigned long long)offset);
  if (val == -1)
    return -1;

#ifdef IODUMP
#ifdef PAGETBL_DUMP
  print_pgtbl(proc, 0, -1);
#endif
  MEMPHY_dump(proc->krnl->mram);
#endif
  return val;
}
```

4.5.6 Page Management

- **find_victim_page()**: Page replacement by FIFO algorithm.
- **pg_getpage()**: Retrieves page, handles swapping.
- **pg_getval() / pg_setval()**: Memory access primitives.

```
int pg_getpage(struct mm_struct *mm, int pgn, int *fpn, struct pcb_t *caller)
{
  uint32_t pte = pte_get_entry(caller, pgn);
  if (!PAGING_PAGE_PRESENT(pte))
  {
    addr_t vicpgn;
    if (find_victim_page(caller->mm, &vicpgn) == -1) return -1;
    struct sc_regs regs;
    regs.a1 = SYSMEM_SWP_OP;
    regs.a2 = vicpgn; // Victim PGN
    regs.a3 = pgn;    // Target PGN
    syscall(caller->krnl, caller->pid, 17, &regs);
    enlist_pgn_node(&caller->mm->fifo_pgn, pgn);
  }
  *fpn = PAGING_FPN(pte_get_entry(caller, pgn));
  return 0;
}
```

4.5.7 Integration with OS Core

A key feature of the implementation is the strict enforcement of User/Kernel Separation:

- **User Space (libmem.c)**: Never passes `struct pcb_t` pointers directly to the kernel. All operations are performed via `syscall()` using abstract operation codes (`SYSMEM_MAP_OP`, `SYSMEM_INC_OP`, etc.).
- **Kernel Space (sys_mem.c)**: The syscall handler `__sys_memmap` receives the Process ID (pid) from the user. It uses a secure helper function `find_proc_safe(krnl, pid)` to traverse the kernel's process lists (`ready_queue, running_list`) and resolve the valid PCB. This prevents malicious user programs from compromising kernel integrity by passing invalid pointers.

5 Questions

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

Answer: In systems with limited register availability, passing complex arguments such as large structures, arrays, or strings directly via registers is impractical due to size constraints. Instead, the standard approach is to pass these arguments **by reference**:

- The complex data is first stored in a known region of memory accessible to the process (e.g., a heap-allocated buffer).
- A pointer to the memory address containing this data is then passed to the system call through a register.
- Inside the kernel (or system call handler), this pointer is dereferenced to access and manipulate the actual data.

This technique allows efficient communication of large or structured data without exceeding the limited number of registers (e.g., a0, a3 in RISC-like systems). It also ensures that only a constant-sized value (a pointer) is passed via the syscall interface.

In the context of this assignment: When a user program calls a system call such as memmap, complex arguments (like a region ID referencing a name string) are passed using a register that holds the **address** of the data stored in virtual memory. The system call handler then uses that address to read from the process's memory, simulating how a real OS manages user-to-kernel data exchange.

Conclusion: Passing complex arguments by reference allows system calls to work efficiently and flexibly, even in architectures with limited register space. It is a common design pattern in real-world operating systems.

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

Answer: If a system call takes too long to execute (i.e., it is a long-running or blocking system call), it can negatively impact the performance and responsiveness of the operating system in several ways:

- **CPU Occupation**: A long system call may monopolize the CPU, preventing other ready processes from being scheduled, especially in single-CPU environments.
- **Blocking Behavior**: If the system call is blocking and runs in kernel mode, the calling process may be stuck in an uninterruptible wait state, delaying overall system responsiveness.
- **Reduced System Throughput**: Other system calls or background processes may have to wait longer, reducing the number of tasks completed over time.

- **Potential Deadlocks or Starvation**: In a poorly synchronized environment, such as the multi-core simulation in this assignment, a long system call that holds locks may block other CPUs, leading to deadlock or starvation.

  Mitigation Strategies:

- **Use of Timeouts**: Set a maximum execution time or implement watchdogs to interrupt or abort excessively long-running system calls.
- **Non-blocking and Asynchronous Design**: Where applicable, system calls can return immediately and perform their work in the background using worker threads or deferred execution.
- **Improved Scheduling**: Prioritize short jobs or system calls via scheduling policies (e.g., MLQ in this assignment) to ensure fair CPU usage and avoid starvation.
- **Kernel Yielding**: Allow long system calls to yield CPU voluntarily at checkpoints to give other processes execution opportunities.

Conclusion: Long execution time in system calls can harm overall system performance and user experience. Proper design with scheduling awareness, timeouts, and non-blocking mechanisms is essential, especially in multi-process, multi-CPU systems like the one simulated in this assignment.

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

Answer: The MLQ (Multilevel Queue Scheduling) algorithm offers several advantages compared to traditional scheduling methods:

- **Better Performance for Mixed Workloads**: MLQ can separate processes into different queues based on their type or priority (e.g., system processes, interactive processes, batch jobs). This design allows the scheduler to treat each category differently, optimizing performance for mixed workloads. For example, interactive tasks can be scheduled with shorter time slices to improve responsiveness, while CPU-bound tasks can be scheduled less frequently to avoid resource hogging.
- **Avoiding Starvation (within a queue)**: In contrast to Priority Scheduling or SJF (Shortest Job First), which may lead to starvation of lower-priority or long- running jobs, MLQ uses Round Robin (RR) within each queue. This ensures fairness among processes of the same priority level, helping prevent starvation as long as all queues are given CPU time in a round-robin fashion.
- **Improved Flexibility and Control**: With multiple queues and configurable time slices or CPU slots per queue (as seen in this assignment's implementation using slot = MAX PRIO - prio), the system can prioritize urgent or important tasks without completely ignoring lower-priority ones. It also enables better tuning in systems with diverse resource demands.

- **Fairness and Efficiency**: While giving priority to more critical processes, MLQ still ensures that all active queues eventually get CPU time. Compared to non- preemptive algorithms like FCFS (First-Come First-Serve), MLQ is more responsive and avoids long wait times caused by a single long process.
- **Scalability in Multi-CPU Environments**: The MLQ model in the assignment is designed to support multiple CPUs. By using independent ready queues and dispatching logic, MLQ allows parallel execution, improving system throughput and efficiency in multiprocessor systems.
- **Closer to Real OS Behavior**: The MLQ algorithm resembles older Linux scheduler implementations, making it an excellent teaching model for understanding real- world scheduling policies before advancing to more complex models like Completely Fair Scheduler (CFS).

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

Answer: The use of multiple memory segments (also known as multiple `vm_area_struct`) in this simple operating system brings several important advantages:

- **Logical Separation of Data**: By dividing the virtual memory into distinct segments (e.g., heap, stack, code), we logically organize memory usage. This separation reflects how real-world OSes manage different types of memory access, improving both clarity and maintainability of memory operations.
- **Better Memory Management**: The OS can handle allocation and deallocation more effectively by maintaining separate free lists and allocation tables per segment. This helps minimize fragmentation and allows targeted operations (e.g., allocate only in heap area).
- **Security and Isolation**: Each segment can be protected by different access permissions (e.g., code segment as read-only, stack with guard pages), helping detect and prevent invalid memory access.
- **Support for Dynamic Memory Allocation**: With multiple segments, especially a well-defined heap segment (using sbrk boundary), we can implement dynamic allocation strategies such as malloc/free simulation inside the virtual memory space.
- **Scalability and Extendability**: This design allows future extension of the system to include features like memory-mapped files, shared memory, or separate segments for different regions (e.g., text, data, BSS), similar to ELF-format in modern Linux.
- **Easier Page Management**: When using paging (as implemented in this assignment), each segment can be mapped independently to physical memory, allowing page-level protection and swapping per segment, rather than managing the whole space as one big chunk.

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

Answer: Dividing the address into more than two levels in a paging memory management system results in a hierarchical (multi-level) paging structure. This change introduces both benefits and drawbacks:

- **Advantages**:
  - **Scalability for large address spaces**: Multi-level paging is necessary when supporting large virtual address spaces (e.g., 32-bit or 64-bit systems), where a single page table would require too much memory.
  - **Efficient use of memory (space saving)**: Page tables at each level are allocated only when needed (on-demand), which avoids reserving large continuous memory for unused address space.
  - **Modular and hierarchical structure**: This structure organizes address translation cleanly into levels, simplifying expansion and region-specific control over memory (e.g., region-based swapping).
- **Disadvantages**:
  - **Increased memory access time**: Translating a virtual address requires traversing multiple levels of page tables. For example, a 3-level system may require 3 memory accesses before locating the physical address, increasing latency.
  - **Higher implementation complexity**: Multi-level translation logic is more complex to design and maintain, especially in low-level systems or simulations like this simple OS.
  - **TLB (Translation Lookaside Buffer) pressure**: Because more page table entries are involved, TLB misses may become more frequent, which can degrade performance unless TLB is large and well-optimized.

Conclusion: Using more than two levels in paging enables support for massive virtual memory spaces and efficient use of memory, but introduces performance and complexity trade-offs. It is commonly used in modern operating systems (e.g., x86-64 with 5-level paging) where address space scalability is critical.

5.6 What are the advantages and disadvantages of segmentation with paging?

Answer

- **Advantages**:
  - **Efficient Memory Utilization**: Combines the benefits of both segmentation and paging, allowing flexible memory allocation and reducing fragmentation.

- **Protection and Sharing**: Segmentation provides logical grouping of memory, while paging ensures efficient use of physical memory with protection mechanisms at both levels.
- **Simplified Memory Management**: Paging reduces external fragmentation, while segmentation offers a clear logical structure.

- **Disadvantages**:
  - **Increased Complexity**: Managing both segments and pages introduces additional overhead in memory lookup and maintenance.
  - **Internal Fragmentation**: Paging may still cause internal fragmentation within each page.
  - **Overhead for Translation**: Multiple levels of address translation (segment and page tables) can increase access time and system complexity.

5.7 What happens if the synchronization is not handled in your Simple OS? Illustrate the problem of your simple OS (assignment outputs) by example if you have any.

Answer: If synchronization is not handled in the Simple OS (i.e., removing mechanisms like `pthread_mutex_lock` or `enter_critical`), **Race Conditions** will inevitably occur. Since the OS simulation runs on a multi-threaded environment (representing multi-core CPUs) where resources are shared, concurrent access without protection leads to data inconsistency, resource leakage, or system instability.

Specifically, in our implementation, the absence of synchronization would impact two critical subsystems:

- **Memory Management (`libmem.c`):** The `vm_freerg_list` (free virtual memory regions) and physical frame lists are shared resources. Without the `mmvm_lock`, two processes running on different CPUs could read the same "free" node simultaneously before it is updated. This results in the **same memory address** being allocated to two different processes, leading to data corruption.
- **Scheduler (`sched.c`):** The `ready_queue` is accessed by multiple CPUs via `get_proc()`. Without `queue_lock`, two CPUs might dequeue the **same process** simultaneously, violating the principle that a process can only execute on one CPU at a time.

Illustration by Example:

We can illustrate this problem by contrasting the observed correct behavior (with synchronization) from our output `os_1_mlq_paging.output` against a hypothetical failure scenario.

1. **Observed Correct Behavior (With Synchronization):** In `os_1_mlq_paging.output` at Time slot 7, multiple CPUs are active:

```
Time slot 7
...
CPU 1: Put process  2 to run queue
CPU 1: Dispatched process 2
CPU 0: Dispatched process 4
```

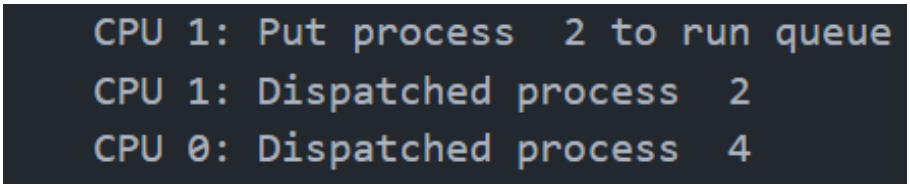

Figure 3: A part of log output of test case `os_1_mlq_paging`

2. **Hypothetical Failure (Without Synchronization):** If we removed `pthread_mutex_lock` in `src/libmem.c`, the following race condition could occur:
   - **Step 1:** CPU 2 (Process 2) calls `alloc`. It reads the free list and sees address `0` is available.
   - **Step 2:** Before CPU 2 updates the list to remove this node, CPU 3 (Process 1) also calls `alloc` and reads the *same* free list state.
   - **Step 3:** Both functions return `0`.

   Result: Both Process 1 and Process 2 believe they own the memory at `0`. When Process 1 writes data, it overwrites Process 2's data, causing a critical system failure.

Conclusion: The use of `pthread_mutex` in our implementation effectively prevents these race conditions, ensuring that shared resources like memory regions and process queues are accessed atomically.

6 Observation and Conclusion

6.1 Observation

Through simulation and testing with various scenarios, the following behaviors and metrics were observed:

- **Scheduler Behavior (MLQ):** The scheduler correctly implemented the Multi-Level Queue (MLQ) policy. As observed in the output logs (e.g., `sched_0.txt`), processes with higher priority (lower `prio` value) were consistently dispatched before lower-priority ones. The time-slicing mechanism within each priority queue ensured fairness, preventing starvation for processes with the same priority level.
- **Memory Management Efficiency (64-bit Paging):** The 5-level paging system demonstrated significant storage efficiency due to the **Dynamic Allocation** strategy.
  - In the `os_0_mlq_paging` test case, the page table storage required only **32,768 bytes** (= 32 KB).
  - In the more complex `os_1_mlq_paging` scenario, the storage size increased to **81,920 bytes** (= 80 KB).

```
============================================================
        MULTILEVEL PAGING STATISTICS (MM64)
============================================================
  [+] Page Table Storage Size : 32768 bytes
  [+] Memory Access Count     : 40 times
============================================================
```

Figure 4: Log Output of test case `os_0_mlq_paging`

```
============================================================
        MULTILEVEL PAGING STATISTICS (MM64)
============================================================
  [+] Page Table Storage Size : 81920 bytes
  [+] Memory Access Count     : 80 times
============================================================
```

Figure 5: Log Output of test case `os_1_mlq_paging`

The Memory Access Count was recorded at **40** times for the `os_0_mlq_paging` test case and **80** times for the `os_1_mlq_paging` test case reflecting the overhead of traversing the 5-level page table structure, which is a trade-off for the efficient memory usage.

This confirms that the system allocates page table levels (P4D, PUD, PMD, PT) only on demand, avoiding the prohibitive memory cost of statically allocating the entire 64-bit address space.

- **System Call Security:** The system successfully enforced User/Kernel separation. Memory operations initiated from user space correctly triggered system calls (e.g., `memmap`), and the kernel resolved the Process Control Block (PCB) using the **Process ID (PID)** rather than accepting unsafe pointers from user space.

6.2 Conclusion

This assignment provided hands-on experience with designing and implementing core components of a modern, 64-bit operating system. Key takeaways include:

- **Mastery of 64-bit Architecture:** We successfully designed a 5-level paging mechanism capable of handling a vast address space. The implementation of dynamic page table allocation highlighted the trade-off between memory efficiency (storage) and access overhead (multiple memory accesses per translation).
- **Process Management:** Implementing the MLQ scheduler reinforced concepts of process priority, context switching, and CPU resource distribution.
- **Concurrency Challenges:** The use of mutexes to protect shared resources in the memory library illustrated the complexities of synchronization in a multi-core environment.

Overall, the simulation successfully met the requirements for a robust, secure, and efficient operating system model.

For a better overview of our source code, check this github repository: Operating Systems Assignment

References

[1] Silberschatz, A., Galvin, P. B., & Gagne, G. *Operating System Concepts (10th Edition)*. Wiley, 2018.

[2] Tanenbaum, A. S., & Bos, H. *Modern Operating Systems (4th Edition)*. Pearson, 2014.

[3] Love, R. *Linux Kernel Development (3rd Edition)*. Addison-Wesley Professional, 2010.

[4] Bovet, D. P., & Cesati, M. *Understanding the Linux Kernel (3rd Edition)*. O'Reilly Media, 2005.

[5] Gorman, M. *Understanding the Linux Virtual Memory Manager*. Prentice Hall, 2004.

[6] Denning, P. J. *Virtual Memory*. ACM Computing Surveys (CSUR), 2(3), 153-189, 1970.

[7] Kernighan, B. W., & Ritchie, D. M. *The C Programming Language (2nd Edition)*. Prentice Hall, 1988.

[8] Free Software Foundation. *The GNU C Reference Manual*. https://www.gnu.org/software/gnu-c-manual/

[9] Stallman, R., & GNU Project Volunteers. *GNU Coding Standards*. https://www.gnu.org/prep/standards/