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

OPERATING SYSTEM (CO2017)

Assignment

"Simple Operating System"

Instructor(s): Vũ Thành Tài

Students: Nguyễn Hồ Quang Khải - 2352538 ***Leader***
Võ Hoàng Nguyên - 2352844
Phạm Quang Huy - 2352405
Hoàng Thiên Kim - 2352660
Lê Mạnh Hưng - 2352428

HO CHI MINH CITY, APRIL 2025

Contents

List of Figures

List of Tables

Member list & Workload

No.	Fullname	Student ID	Problems	% done
1	Nguyễn Hồ Quang Khải	2352538	- Implement mm-vm.c, mm-memphy.c, mm.c files - Implement libmem.c file - Implement sys-kiall file	60%
2	Phạm Quang Huy	2352405		0%
3	Võ Hoàng Nguyên	2352844	- Implement queue.c and sched.c - Implement report	30%
4	Hoàng Thiên Kim	2352660		0%
5	Lê Mạnh Hưng	2352428	- Implement report	10%

Table 1: Member list & workload

1 Introduction

1.1 General

This assignment focuses on simulating a simplified operating system (Simple OS), providing students with practical experience and a deep understanding of core OS concepts. The main objective of this simulation is to illustrate fundamental OS mechanisms, including process scheduling, memory management, and system call handling.

The implemented Simple OS consists of three critical modules:

- **Scheduler Module.** Responsible for managing CPU resources efficiently, allocating CPU time to multiple processes based on their priority using the Multi-Level Queue (MLQ) scheduling policy. This module ensures fair and effective distribution of CPU time among processes.
- **Memory Management Module.** This module manages both virtual memory (logical memory seen by processes) and physical memory (RAM and swap devices). It utilizes a paging-based memory allocation mechanism to translate virtual addresses used by processes into physical addresses stored in main memory or swapped out to secondary storage. It ensures isolation among processes, enhances memory usage efficiency, and handles memory allocation requests dynamically.
- **System Call Module.** This module provides a fundamental interface allowing user-level applications to request services from the OS kernel. Students learn to implement and handle system calls, facilitating operations such as process management and memory operations directly from user-level processes to the OS kernel space.

By integrating these modules into a cohesive unit, the Simple OS simulation not only demonstrates core operating system principles but also equips students with practical skills to comprehend how an actual operating system organizes, coordinates, and manages hardware and software resources. Through this hands-on experience, students can clearly visualize the roles, interactions, and functionalities of crucial OS components, bridging theoretical knowledge with practical system-level programming.

1.2 Source Code Structure and Tools

Source Code Structure

The provided source code for this assignment is clearly structured into multiple header files (`.h`) and source files (`.c`). Each file serves a distinct role within the Simple Operating System implementation. Specifically:

Header files:

- `timer.h`: Defines timing functionalities used by the system.
- `cpu.h`: Contains definitions for simulating virtual CPU operations.
- `queue.h`: Declares functions for managing queues, used primarily to store process control blocks (PCBs).
- `sched.h`: Defines functions used by the scheduler module for process management.
- `mem.h`: Declares essential functions related to virtual memory management.
- `loader.h` (obsoleted): Previously defined functions for loading programs from disk to memory.
- `common.h`: Contains common structures and definitions used throughout the OS.
- `bitopts.h`: Provides bit-level operations essential for memory management and flags handling.
- `os-mm.h`, `mm.h`: Define data structures and fundamental concepts required for paging-based memory management.
- `os-cfg.h`: Contains definitions for enabling or disabling specific OS features through configuration constants.

Source files:

- `timer.c`: Implements the timing mechanism used for managing time slices and scheduling processes.
- `cpu.c`: Implements virtual CPU operations, executing instructions from processes.
- `queue.c`: Implements queue management functions (`enqueue`, `dequeue`) for handling PCBs based on priority.
- `paging.c` (obsoleted): Originally used for validating the virtual memory mechanism functionality.

- `os.c`: The main file containing the program's entry point, initializing and running the OS.
- `loader.c`: Implements program loader functionality, initializing processes from disk-based instructions.
- `sched.c`: Implements the Multi-Level Queue (MLQ) scheduling algorithm, responsible for selecting processes to execute.
- `mem.c` (obsoleted): Previous implementation of virtual memory and RAM management mechanisms.
- `mm.c`, `mm-vm.c`, `mm-memphy.c`: Implement paging-based virtual-to-physical memory management, including page swapping and address translation functionalities.

Makefile: A configuration file used to compile the source code into executable files using the `make` command.

Input/Output folders:

- `input`: Contains various configuration and process instruction files to simulate different OS scenarios.
- `output`: Provides reference output examples for validating the correctness of the implemented OS modules.

Tools and Environment

This assignment has been implemented using the following development tools and environment:

- **Operating System:** Linux Ubuntu (version 20.04 LTS)
- **Compiler:** GNU Compiler Collection (`gcc`)
- **Debugger:** GNU Debugger (`gdb`) for debugging and validating the source code
- **Build Automation:** GNU Make (`make`), utilized with the provided Makefile to compile and link source code conveniently.
- **Text Editor:** Visual Studio Code for source code editing and management.

Using these tools, the team ensures efficient implementation, compilation, and debugging of the Simple Operating System components, enabling thorough testing and validation of OS functionalities according to assignment requirements.

1.3 Objectives and Purpose

The main objective of this assignment is to provide a practical understanding of core Operating System (OS) concepts, including process scheduling, memory management, and system call handling. Through the simulation of a simple OS, students gain hands-on experience with the key components of an OS and their interrelationships. The primary goals of this assignment are:

- **Process Scheduling:** To implement and understand process scheduling mechanisms, specifically the Multi-Level Queue (MLQ) scheduling algorithm, which categorizes processes based on their priority. The objective is to ensure that high-priority processes are given preferential treatment while maintaining fairness in CPU time distribution.
- **Memory Management:** To simulate the management of both virtual and physical memory. The purpose is to provide a clear understanding of how memory is allocated, how paging works, and how segmentation and swapping can be used to optimize memory usage in an OS.
- **System Calls:** To implement a basic system call mechanism, allowing user-level processes to request services from the OS kernel. This helps students grasp how system calls are used to interface between user applications and the OS kernel.
- **Thread Synchronization:** To understand and apply thread synchronization techniques in a multi-threaded environment, ensuring safe access to shared resources like queues and memory. This aspect focuses on avoiding race conditions and data corruption in a concurrent system.
- **OS Simulation:** To create a simplified OS simulation that integrates process scheduling, memory management, and system calls into a cohesive system. This helps students connect theoretical knowledge with practical implementation in the context of OS design.

The purpose of this assignment is not only to simulate core OS functionality but also to give students an in-depth understanding of how real-world operating systems manage resources and ensure process execution efficiently and fairly.

2 Implementation

2.1 Scheduler

The scheduler in the Simple Operating System is responsible for managing CPU resources efficiently and ensuring that processes are executed according to their priority. This is achieved using the **Multi-Level Queue (MLQ)** scheduling algorithm, which categorizes processes into different queues based on their priority.

2.1.1 Key Components of the Scheduler

- **Ready Queue and Run Queue:**
  - **Ready Queue:** Holds processes that are ready to be executed, waiting for CPU time.
  - **Run Queue:** Contains processes that are currently being executed by the CPU.
- **Multi-Level Queue (MLQ):** The system uses multiple ready queues, each corresponding to a specific priority level. Processes are assigned to these queues based on their priority, with higher-priority processes placed in queues with lower numerical values (indicating higher priority) and lower-priority processes in queues with higher numerical values.
- **Process Control Block (PCB):** Each process is represented by a **Process Control Block (PCB)**, which stores the process's state, priority, and other metadata required for scheduling.

2.1.2 Functions for Queue Management

- **Enqueue and Dequeue:**
  - **Enqueue:** Adds a process to the appropriate queue based on its priority.
  - **Dequeue:** Removes and returns a process from the queue, prioritizing the one with the highest priority (from the queue with the lowest numerical value).

2.1.2.1 Enqueue Function:

The `enqueue` function adds a new process to the queue, ensuring that the process is inserted at the correct position based on the current size of the queue.

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

2.1.2.2 Dequeue Function: The `dequeue` function removes and returns the highest-priority process from the queue. If the queue is empty, it returns NULL.

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

- **MLQ Scheduling:** In **MLQ Scheduling**, the scheduler iterates through priority queues to fetch processes for execution, selecting the process with the highest priority (from the queue with the lowest numerical value). The `get_mlq_proc` function retrieves processes from the appropriate queue while respecting remaining slots and queue availability.

2.1.2.3 MLQ Scheduling Process: The `get_mlq_proc` function selects the highest-priority process from the queues. If the current queue is empty or out of slots, it moves to the next queue. The function uses a mutex lock to ensure thread safety during the process retrieval.

```c
struct pcb_t *get_mlq_proc(void) {
    static unsigned long cur_prio = 0;
```

```
    static int remain_slot = 0;
    struct pcb_t *proc = NULL;
    pthread_mutex_lock(&queue_lock);
    int all_empty = 1;
    for (unsigned long prio = 0; prio < MAX_PRIO; prio++) {
        if (!empty(&mlq_ready_queue[prio])) {
            all_empty = 0;
            break;
        }
    }
    if (all_empty) {
        pthread_mutex_unlock(&queue_lock);
        return NULL;
    }
    if (remain_slot == 0 || empty(&mlq_ready_queue[cur_prio]))
        {
        for (unsigned long i = 0; i < MAX_PRIO; i++) {
            unsigned long next_prio = (cur_prio + i) %
               MAX_PRIO;
            if (!empty(&mlq_ready_queue[next_prio])) {
                cur_prio = next_prio;
                remain_slot = MAX_PRIO - cur_prio;
                break;
            }
        }
    }
    if (!empty(&mlq_ready_queue[cur_prio])) {
        proc = dequeue(&mlq_ready_queue[cur_prio]);
        remain_slot--;
        if (remain_slot == 0) {
            cur_prio = (cur_prio + 1) % MAX_PRIO;
        }
    }
```

```
34
35      pthread_mutex_unlock(&queue_lock);
36      return proc;
37  }
```

- **Thread Synchronization:** To ensure thread safety when accessing shared resources (queues), a **mutex lock** (`queue_lock`) is employed, preventing race conditions in a multi-threaded environment.

2.1.2.4 Thread Synchronization with Mutex Lock:

The mutex lock ensures that only one thread can access the shared queues at a time. This prevents multiple threads from modifying the queues simultaneously, which could lead to data corruption or inconsistencies.

```
1  pthread_mutex_t queue_lock;
2  pthread_mutex_init(&queue_lock, NULL);
3  pthread_mutex_lock(&queue_lock);
4  pthread_mutex_unlock(&queue_lock);
```

2.1.3 Scheduler Initialization

When initializing the scheduler, necessary data structures such as the ready queue and run queue are created. If **MLQ Scheduling** is enabled, the multi-level queues are also initialized.

```
1  void init_scheduler(void) {
2  #ifdef MLQ_SCHED
3      int i;
4      for (i = 0; i < MAX_PRIO; i++) {
5          mlq_ready_queue[i].size = 0;
6          slot[i] = MAX_PRIO - i;
7      }
8  #endif
9      ready_queue.size = 0;
10     run_queue.size = 0;
11     pthread_mutex_init(&queue_lock, NULL);
12 }
```

2.1.4 Conclusion

This section describes the implementation of the **Multi-Level Queue (MLQ)** scheduling algorithm, which manages process execution based on priority. By organizing processes into separate queues for each priority level, the scheduler ensures that higher-priority processes are executed before lower-priority ones. Thread safety is maintained through mutex locks, preventing race conditions in a multi-threaded environment.

2.2 Memory Management

2.2.1 File libmem.c (int _alloc)

Description: This function allocates memory for a process. If the process has exhausted its current memory (allocated during process creation), it requests additional memory from the OS. This memory is divided into two parts: one for running the process and another part that remains free for subsequent use.

```
int __alloc(struct pcb_t *caller,int vmaid,int rgid,int size,int *alloc_addr)
```

Steps to realization: Parameters:

- `caller`: The process requesting memory.
- `vmaid`: The ID of the virtual memory area the process is using.
- `rgid`: The ID of the memory region (symbol table).
- `size`: The size of the memory block requested.
- `alloc_addr`: A pointer to an integer that will store the address of the allocated memory.

Steps to realization:

1. Check for available free memory regions by calling `get_free_vmrg_area`.
2. If a sufficiently large free region is available, allocate memory using that region. The address of the allocated memory is stored in a register for the process, after that return value for function
3. If no sufficiently large free region is available, increase the virtual memory region by the required size and go to step 4 5 6 7 8.
4. Calculate the total memory needed (based on the requested size).

5. Using `inc_vma_limit`, increase the memory limit for the process's virtual memory area.
6. Allocate memory for the process.
7. Call the `sys_call` to increase the memory.
8. Update `alloc_addr`.

2.2.2 File libmem.c (int _free)

Description: This function frees previously allocated memory. Simply get information about the memory region, then free that region by adding it to the list of free regions (`enlist_vm_freerg_list`)

```
int _free(struct pcb_t *caller, int vmaid, int rgid)
```

Parameters:

- `caller`: The process requesting to free memory.
- `vmaid`: The ID of the virtual memory area the process is using.
- `rgid`: The ID of the memory region (symbol table).

Steps to realization:

1. Get information about the memory region using `get_symrg_byid`.
2. Remove information about the memory region including `rg_start`, `rg_end`, and `rg_next`.
3. Put the freed memory region into the free list by calling the function `enlist_vm_freerg_list`.

2.2.3 File libmem.c (int pg_getpage)

Description: This function performs the paging operation (paging) and swaps data between RAM and swap memory when the memory page is not present in RAM.

```
int pg_getpage(struct mm_struct *mm, int pgn, int *fpn, struct pcb_t *caller)
```

Parameters:

- `mm`: A pointer to the `mm_struct` structure for the process, containing information about the process's memory.
- `pgn`: The page number being accessed by the process.

- `fpn`: A pointer to an integer that will store the frame number where the page is stored.
- `caller`: The process requesting the page.

Steps to realization:

1. Get information about the page, check the page table to see if the page is in RAM. If not, search the swap memory for the corresponding page.
2. From the page table, take the entry corresponding to the page.
3. If we can find a free frame in memory, swap the page from RAM to secondary storage.
4. If we can find the frame in memory, return it.

2.2.4 File libmem.c (int pg_getval)

Description: This function reads data from a memory address at a given process.

```
int pg_getval(struct mm_struct *mm, int addr, BYTE *data, struct pcb_t *caller)
```

Parameters:

- `mm`: A pointer to the `mm_struct` structure of the process, containing information about the process's memory.
- `addr`: The address in memory that needs to be read. This address has already been converted to a page number.
- `data`: A pointer to the byte that will be stored after reading from the memory.
- `caller`: A pointer to the PCB (Process Control Block) of the process executing the command.

Steps to realization:

1. Check if the required page is in RAM or not. If not, return.
2. Calculate the physical address by byte to be written into memory.
3. Read data into the byte in memory.
4. Perform system calls.
5. Update data and open the mutex lock after the operation.

2.2.5 File libmem.c (pg_setval)

Description: This function writes data to memory at a specific address.

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

Parameters:

- `mm`: A pointer to the `mm_struct` structure of the process, containing information about the process's memory.
- `addr`: The address in memory to which the value needs to be written. This address will be converted to a page number.
- `value`: The byte value to be written at the address. This value is stored in memory after the command has finished executing.
- `caller`: A pointer to the PCB structure of the process being executed.

Steps to relization:

1. Check if the memory required has been stored on the RAM. If not, return.
2. Calculate the physical byte address to which the byte is to be written into memory.
3. Write the value to memory at the calculated physical address.
4. Perform the system call.
5. Update data and release the mutex lock after the operation.

2.2.6 File libmem.c (int find_victim_page)

Description: This function is used to find a "victim" page to remove from RAM when memory is full.

```
int find_victim_page(struct mm_struct *mm, int *retpgn)
```

Parameters:

- `mm`: A pointer to the `mm_struct` structure of the process, which contains information about the memory of that process.

- `retpgn`: A pointer to an integer variable which will contain the page index of the selected "victim" page for swapping (victim page). This is the page that will be swapped if memory is not sufficient.

Steps to relization:

1. Check the FIFO list. If a free page is available, return -1.
2. Take the first page in FIFO, the page is chosen as the "victim" to swap.
3. Update FIFO list: Update the pointer to point to the next page.
4. Free the memory of the chosen page: Free memory by taking it out of the page.
5. Return the victim page: Returns the number of pages for swapping.

2.2.7 File libmem.c (int get_free_vmrg_area)

Description: This function has the ability to find a memory region with a suitable and corresponding size.

```
int get_free_vmrg_area(struct pcb_t *caller, int vmaid, int size, struct
    vm_rg_struct *newrg)
```

Parameters:

- `caller`: Process which requires allocating pages.
- `vmaid`: ID of pages (virtual memory area) which the process uses.
- `size`: Size of memory block requested.
- `newrg`: Information on pages which will be allocated, if found.

Steps to relization:

1. Get information on VMA in the process table via ID (vmaid).
2. Search for the available pages using (vm_freerg_list) to find a page with sufficient size.
3. If a sufficient size is found.
4. Update starting address (rg_start) and ending address (rg_end) for the allocated page.

2.2.8 File mm-memphy.c (int MEMPHY_dump)

Description: Function for memory dumping

```
int MEMPHY_dump(struct memphy_struct *mp)
```

Steps to relization:

1. This function iterates through each word in memory, printing the address and value of each word (since each word takes up 4 bytes, the iteration is performed with i * 4).

2. In addition, if the memory is not enough, the function will return -1 to indicate an error. Upon successful completion, the function will return 0.

2.2.9 File mm.c (int VMAP_page_range)

Description: The function assigns frames to virtual address, maps the frames that have been allocated to all the PTEs corresponding to pages based on FIFO. Simply stated, the function iterates through the list of physical frames. For each frame, it will map the frame to the page and to the pages based on the FIFO rule.

```
int VMAP_page_range(struct pcb_t *caller,
                    int addr,
                    int pgnumn,
                    struct framephy_struct *frames,
                    struct vm_rg_struct *ret_rg)
```

Parameters:

1. `caller`: The process control block of the process calling the function.

2. `addr`: The starting address, which is aligned to the page size (`pagesz`).

3. `frames`: A list of the mapped frames.

4. `ret_rg`: The return mapped region.

Steps to relization:

1. Initialize the parameters: `pgit`: Variable used to iterate through the number of pages (`pgnum`). `pgn`: Variable to store page index (page number) of the starting address (`addr`).

2. Initialize the memory area: Initialize the memory area that starts and ends in `addr`.

3. Map memory blocks by frames:
    - Loop through the list of physical memory (frames).
    - For each memory block (frame), set the frame to the page with the current index.
    - Update information for the memory block for each page table (`pte_set_fpn`).
    - At the same time, each frame will be added to the FIFO list for the next use.
4. After the mapping of all pages completes, the function will update the ending address of the memory area to the mapped address.

2.2.10 File mm.c (int alloc_pages_range)

Description: Function to request various page frames from. `memphy_struct` (RAM), returns the frame list that can be allocated. In simple terms, the goal is to traverse through page frames in the process memory. If a page frame is found to be empty, the page frame will be saved. And the empty page frames will be returned, after going through the list to see if there are enough page frames according to the request

```
int alloc_pages_range(struct pcb_t *caller, int req_pgnum, struct framephy_struct
    **frm_lst)
```

Parameters:

- `caller`: Pointer to the process calling the function.
- `req_pgnum`: Number of pages which the process requires.
- `frm_lst`: Pointer to `framephy_struct`, this is the list of frame pages that has been received and returned after successful completion.

Steps to relization:

1. Initialize:
    - `pgit`: Variable to loop and iterate through the page blocks needed to be allocated.
    - `fpn`: Variable for the frame number (frame number) of a memory block that has been allocated.
    - `newfp_str = NULL`: Creates a new `newfp_str` pointer, to save the information for blocks which have not been allocated.

- `tmp = NULL`: Creates a `tmp` pointer to maintain the list of memory blocks that have been allocated.

2. Loop for each time a new frame is discovered. If a new empty frame is found, save `fpn` and add the frame to `vaotmp`.

3. If there are not enough memory blocks based on the request, return to memory.

2.2.11 File mm.c (int init_mm)

Description: Function to create a new memory region for a process: create the page table (`pgd`), set the start of VMA0 and create the first empty memory region (vm_rg_struct)

```
int init_mm(struct mm_struct *mm, struct pcb_t *caller)
```

Parameters:

- `mm`: Representation for memory space of process
- `caller`: Process that is calling the functions

Steps to relization:

1. Creates a page table
2. Creates a new area for memory vma0
3. Creates the first empty region
4. Links different regions (updates `mmap`)

2.2.12 File mm-vm.c (int get_vm_area_node_at_brk)

Description: Function to create a new memory block for a process at the current location of `sbrk` (is heap, and allows to extend).

```
struct vm_rg_struct *get_vm_area_node_at_brk(struct pcb_t *caller, int vmaid, int
    size, int alignedsz)
```

Parameters:

- `caller`: The process requesting memory allocation
- `vmaid`: ID of the memory that needs to be allocated.

- `size`: The requested size
- `alignedsz`: Size aligned based on system requirements.

Steps to relization

1. Create the new memory region.
2. Get information on VMA (Virtual Memory Area).
3. Update the information for the new memory region (starting and ending positions).
4. Return the new memory region.

2.2.13 File mm-vm.c (int inc_vma_limit)

Description: This function is responsible for increasing the limit of the virtual memory area (VMA) for a process.

```
1 int inc_vma_limit(struct pcb_t *caller, int vmaid, int inc_sz)
```

Parameters: Parameters:

- `caller`: The process requesting memory expansion (pointer to the process).
- `vmaid`: The ID of the memory area (VMA) to be expanded.
- `inc_sz`: The size (in bytes) to be added to the memory area.

Steps to relization:

1. Allocate memory for the new memory region.
2. Calculate the status of the allocation.
3. Retrieve information about the new memory region to be allocated.
4. Retrieve information about the current VMA.
5. Save the value of `vm_end` before modification.
6. Check for overlap between the new memory region and existing regions.
7. Update the value of `vm_end` in the VMA.
8. Map the memory into RAM.

2.2.14 File mm-vm.c (int validate_overlap_vm_area)

Description: This function is used to check whether the newly allocated memory region (`vmastart`, `vmaend`) overlaps with any existing free regions in the virtual memory area (VMA).

```
int validate_overlap_vm_area(struct pcb_t *caller, int vmaid, int
    vmastart, int vmaend)
```

Parameters:

- `caller`: The process performing the check.
- `vmaid`: The ID of the virtual memory area to check.
- `vmastart`: The start address of the new memory region.
- `vmaend`: The end address of the new memory region.

Note: The function simply iterates through all memory regions of the process, and for each region, checks whether the new region lies within or overlaps with any existing region.

2.3 System Call

2.3.1 File sys_killall.c (int _sys_killall)

This function is used to terminate all processes with the same name as the process name passed into memory.

```
int __sys_killall(struct pcb_t *caller, struct sc_regs* regs)
```

Note: The function simply iterates through all running and queued processes. If a process has the same name as the one passed into memory, it will be terminated (freed from memory).

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/p1s, PID: 1 PRIO: 1
    CPU 0: Dispatched process  1

Time slot   1
    Loaded a process at input/proc/p2s, PID: 2 PRIO: 0

Time slot   2
    CPU 1: Dispatched process  2
    Loaded a process at input/proc/p3s, PID: 3 PRIO: 0

Time slot   3
Time slot   4
    CPU 0: Put process  1 to run queue
    CPU 0: Dispatched process  3

Time slot   5
Time slot   6
    CPU 1: Put process  2 to run queue
    CPU 1: Dispatched process  2

Time slot   7
Time slot   8
    CPU 0: Put process  3 to run queue
    CPU 0: Dispatched process  3

Time slot   9
Time slot  10
    CPU 1: Put process  2 to run queue
    CPU 1: Dispatched process  2

Time slot  11
Time slot  12
    CPU 0: Put process  3 to run queue
    CPU 0: Dispatched process  3

Time slot  13
Time slot  14
    CPU 1: Processed  2 has finished
    CPU 1: Dispatched process  1

Time slot  15
    CPU 0: Processed  3 has finished
    CPU 0 stopped

Time slot  16
Time slot  17
```

```
Time slot   18
    CPU 1: Put process  1 to run queue
    CPU 1: Dispatched process  1

Time slot   19
Time slot   20
    CPU 1: Processed  1 has finished
    CPU 1 stopped
```

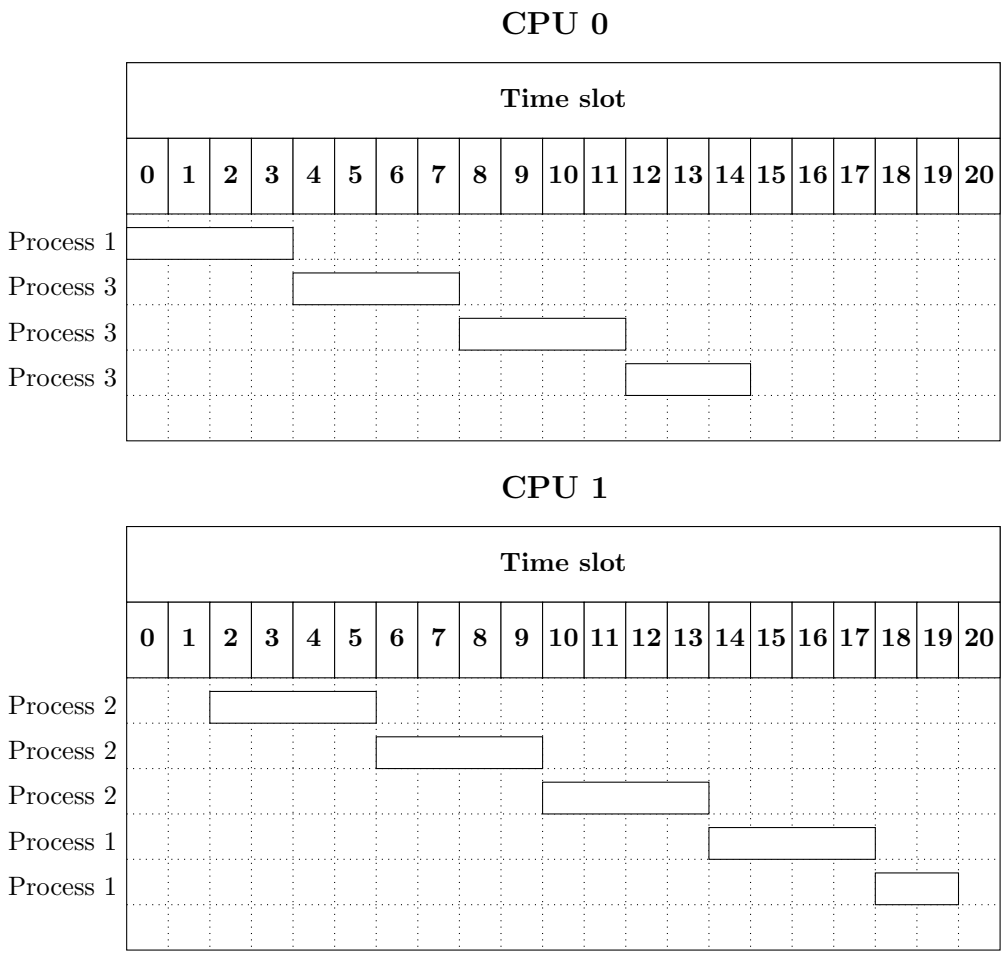
Explanation of Output

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 - 2:**
  - **Loaded Processes:** At time slot 0, process 1 is loaded with priority 1. At time slot 1, process 2 is loaded with priority 0, and at time slot 2, process 3 is also loaded with priority 0. Each process is loaded from the input directory (e.g., `input/proc/p1s`).
  - **Process Dispatching:** In time slot 0, process 1 is dispatched to CPU 0, and in time slot 2, process 2 is dispatched to CPU 1.
- **Time Slot 3 - 4:**
  - **Put Process in Run Queue:** At time slot 4, process 1 is placed in the run queue and process 3 is dispatched to CPU 0 for execution.
- **Time Slot 5 - 6:**
  - **CPU 1 Activity:** In time slot 6, process 2 is placed in the run queue and dispatched to CPU 1 for further execution.
- **Time Slot 7 - 10:**
  - **Continuing Process Dispatch:** During these time slots, the system continues dispatching processes based on priority. In time slot 8, process 3 is placed back in the run queue, while process 3 is dispatched to CPU 0. CPU 1 is dispatching process 2 as needed.
- **Time Slot 11 - 14:**
  - **Process Completion:** In time slot 14, process 2 finishes execution, and CPU 1 dispatches process 1. Meanwhile, CPU 0 completes processing of process 3, which is marked as finished.

- **Time Slot 15 - 20:**
  - **Final Dispatching and Completion:** At time slot 15, CPU 0 stops since process 3 has finished. From time slot 16 to 20, process 1 is dispatched and executed. Finally, at time slot 20, process 1 completes its execution and CPU 1 stops.

3.1.2 Scheduling: Gantt chart



3.2 Memory

3.2.1 The result input file of Memory (os_0_mlq_paging)

```
Time slot   0
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
==== PHYSICAL MEMORY AFTER ALLOCATION ====
PID=1 - Region=0 - Address=00000000 - Size=300 byte
print_pgtbl: 0 - 512
00000000: 80000001
00000004: 80000000
Page number: 0 -> Frame number: 1
Page number: 1 -> Frame number: 0
================================================================
  CPU 0: Dispatched process  2
Time slot   2
Time slot   3
==== PHYSICAL MEMORY AFTER ALLOCATION ====
PID=1 - Region=4 - Address=00000200 - Size=300 byte
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page number: 0 -> Frame number: 1
Page number: 1 -> Frame number: 0
Page number: 2 -> Frame number: 3
Page number: 3 -> Frame number: 2
================================================================
  Loaded a process at input/proc/p1s, PID: 3 PRIO: 0
Time slot   4
==== PHYSICAL MEMORY AFTER DEALLOCATION ====
PID=1 - Region=0
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page number: 0 -> Frame number: 1
Page number: 1 -> Frame number: 0
Page number: 2 -> Frame number: 3
Page number: 3 -> Frame number: 2
================================================================
Time slot   5
==== PHYSICAL MEMORY AFTER ALLOCATION ====
PID=1 - Region=1 - Address=0000032c - Size=100 byte
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page number: 0 -> Frame number: 1
Page number: 1 -> Frame number: 0
Page number: 2 -> Frame number: 3
```

```
Page number: 3 -> Frame number: 2
================================================================
  Loaded a process at input/proc/p1s, PID: 4 PRIO: 0
Time slot   6
==== PHYSICAL MEMORY AFTER WRITING ========
write region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page number: 0 -> Frame number: 1
Page number: 1 -> Frame number: 0
Page number: 2 -> Frame number: 3
Page number: 3 -> Frame number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000240: 100
===== PHYSICAL MEMORY END-DUMP =====
================================================================

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
Time slot  10
Time slot  11
Time slot  12
Time slot  13
  CPU 1: Put process  3 to run queue
  CPU 1: Dispatched process  1
==== PHYSICAL MEMORY AFTER READING ====
read region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page number: 0 -> Frame number: 1
Page number: 1 -> Frame number: 0
Page number: 2 -> Frame number: 3
Page number: 3 -> Frame number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000240: 100
===== PHYSICAL MEMORY END-DUMP =====
```

```
================================================================
Time slot  14
==== PHYSICAL MEMORY AFTER WRITING ========
write region=2 offset=20 value=102
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page number: 0 -> Frame number: 1
Page number: 1 -> Frame number: 0
Page number: 2 -> Frame number: 3
Page number: 3 -> Frame number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000114: 102
BYTE 00000240: 100
  CPU 0: Put process  4 to run queue
  CPU 0: Dispatched process  3
===== PHYSICAL MEMORY END-DUMP =====
================================================================
Time slot  15
==== PHYSICAL MEMORY AFTER READING ====
read region=2 offset=20 value=102
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page number: 0 -> Frame number: 1
Page number: 1 -> Frame number: 0
Page number: 2 -> Frame number: 3
Page number: 3 -> Frame number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000114: 102
BYTE 00000240: 100
===== PHYSICAL MEMORY END-DUMP =====
================================================================
Time slot  16
==== PHYSICAL MEMORY AFTER WRITING ========
write region=3 offset=20 value=103
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page number: 0 -> Frame number: 1
Page number: 1 -> Frame number: 0
```

```
Page number: 2 -> Frame number: 3
Page number: 3 -> Frame number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000114: 103
BYTE 00000240: 100
===== PHYSICAL MEMORY END-DUMP =====
================================================================
Time slot   17
==== PHYSICAL MEMORY AFTER READING ====
read region=3 offset=20 value=103
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page number: 0 -> Frame number: 1
Page number: 1 -> Frame number: 0
Page number: 2 -> Frame number: 3
Page number: 3 -> Frame number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000114: 103
BYTE 00000240: 100
===== PHYSICAL MEMORY END-DUMP =====
================================================================
Time slot   18
  CPU 0: Processed  3 has finished
  CPU 0: Dispatched process  4
  CPU 1: Put process  1 to run queue
  CPU 1: Dispatched process  1
==== PHYSICAL MEMORY AFTER DEALLOCATION ====
PID=1 - Region=4
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page number: 0 -> Frame number: 1
Page number: 1 -> Frame number: 0
Page number: 2 -> Frame number: 3
Page number: 3 -> Frame number: 2
================================================================
Time slot   19
Time slot   20
Time slot   21
  CPU 1: Processed  1 has finished
  CPU 1: Dispatched process  2
Time slot   22
```

```
  CPU 0: Processed  4 has finished
  CPU 0 stopped
Time slot  23
Time slot  24
Time slot  25
  CPU 1: Processed  2 has finished
  CPU 1 stopped
```

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

Data Segment: This segment is used to store global and static variables of a program. Memory in this segment is allocated to processes throughout their execution time.

Heap Segment: Heap is the dynamic memory segment, allocated when needed (e.g., through commands like malloc in C). Processes can request memory from the heap during execution, and when no longer needed, it is freed.

Explaining the status of the memory allocation in heap and data segments

Time Slot 0:

- **Process loaded into memory:** Process PID=1 is loaded into memory at Region 0, occupying 300 bytes, starting at address 00000000.
- **Page Table Update**:
  - Page number 0 -> Frame number 1
  - Page number 1 -> Frame number 0
  - This shows how pages of Process PID=1 are mapped into physical memory.

Time Slot 1:

- Process PID=2 is loaded into memory at address 00000200, occupying 300 bytes.
- PID=1's memory remains unchanged, indicating that memory allocation for PID=2 does not affect PID=1.

Time Slot 3:

- Process PID=1 (Region 4) is reallocated, and its memory pages are mapped again:
  - Page number 0 -> Frame number 1
  - Page number 1 -> Frame number 0

- Page number 2 -> Frame number 3
- Page number 3 -> Frame number 2

- This is an example of memory reallocation after performing actions within a process.

Time Slot 4-5:

- **Memory of PID=1 is deallocated:** When PID=1 completes, the memory it used is freed, and those memory frames will be reused by subsequent processes.

Time Slot 6-17:

- **Write and Read operations on memory:** Subsequent processes perform write and read operations on memory at specific addresses, such as 00000240 and 00000114. Each operation updates the value at those memory locations, showing the state change of memory as processes run.

Time Slot 18-25:

- **Memory deallocation:** When processes complete, their memory is deallocated. The memory frames used by the processes are then available for reuse by other processes. This demonstrates efficient memory management, where memory resources are freed when no longer needed.

3.3 SystemCall

3.3.1 The result input file of Memory (os_syscall

```
    Time slot   0
ld_routine
Time slot   1
Time slot   2
Time slot   3
Time slot   4
Time slot   5
Time slot   6
Time slot   7
Time slot   8
Time slot   9
  Loaded a process at input/proc/sc2, PID: 1 PRIO: 15
Time slot  10
  CPU 0: Dispatched process  1
==== PHYSICAL MEMORY AFTER ALLOCATION ====
```

```
PID=1 - Region=1 - Address=00000000 - Size=100 byte
print_pgtbl: 0 - 256
00000000: 80000000
Page number: 0 -> Frame number: 0
================================================================
Time slot  11
==== PHYSICAL MEMORY AFTER WRITING ========
write region=1 offset=0 value=80
print_pgtbl: 0 - 256
00000000: 80000000
Page number: 0 -> Frame number: 0
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 80
===== PHYSICAL MEMORY END-DUMP =====
================================================================
Time slot  12
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
==== PHYSICAL MEMORY AFTER WRITING ========
write region=1 offset=1 value=48
print_pgtbl: 0 - 256
00000000: 80000000
Page number: 0 -> Frame number: 0
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 12368
===== PHYSICAL MEMORY END-DUMP =====
================================================================
Time slot  13
==== PHYSICAL MEMORY AFTER WRITING ========
write region=1 offset=2 value=-1
print_pgtbl: 0 - 256
00000000: 80000000
Page number: 0 -> Frame number: 0
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 16724048
===== PHYSICAL MEMORY END-DUMP =====
================================================================
Time slot  14
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
==== PHYSICAL MEMORY AFTER READING ====
read region=1 offset=0 value=80
print_pgtbl: 0 - 256
00000000: 80000000
Page number: 0 -> Frame number: 0
================================================================
```

```
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 16724048
===== PHYSICAL MEMORY END-DUMP =====
================================================================
==== PHYSICAL MEMORY AFTER READING ====
read region=1 offset=1 value=48
print_pgtbl: 0 - 256
00000000: 80000000
Page number: 0 -> Frame number: 0
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 16724048
===== PHYSICAL MEMORY END-DUMP =====
================================================================
==== PHYSICAL MEMORY AFTER READING ====
read region=1 offset=2 value=-1
print_pgtbl: 0 - 256
00000000: 80000000
Page number: 0 -> Frame number: 0
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 16724048
===== PHYSICAL MEMORY END-DUMP =====
================================================================
The procname retrieved from memregionid 1 is "P0"
Time slot   15
  CPU 0: Processed  1 has finished
  CPU 0 stopped
```

3.3.2 Show the status of the memory allocation in heap and data segments.

Time Slot 9:

- **Process Loaded:** A process is loaded from input/proc/sc2 with PID=1 and priority 15. The memory allocation for this process occurs in Region 1 (address 00000000, size 100 bytes).
- **Page Table Mapping:** Page number 0 -> Frame number 0.
- **System Call:** The process has its memory allocated and is ready for further interaction with the memory system.

Time Slot 11:

- **Memory Write Operation:** A write system call is performed to store the value 80 at offset 0 in region 1.

- **Memory State After Write:** The byte at address 00000000 is updated to 80. The byte at address 00000000 is updated to 12368. Page number 0 -> Frame number 0.
- System Call: The write operation updates memory in Region 1 at the specified offset.

Time Slot 12:

- **Process Re-queued:** Process PID=1 is put back into the run queue and dispatched again.
- **Memory Write Operation:** A write system call writes the value 48 at offset 1
- **Memory State After Write:** The byte at address 00000000 is updated to 12368.. **Page Table Mapping remains the same**: Page number 0 -> Frame number 0.
- System Call: The memory is updated at the specified offset in Region 1.

Time Slot 13:

- **Memory Write Operation:** A write system call writes the value -1 at offset 2
- **Memory State After Write:** The byte at address 00000000 is updated to 16724048 **Page Table Mapping remains the same**: Page number 0 -> Frame number 0.
- System Call: The memory is further updated at the specified offset.

Time Slot 14:

- **Process Re-queued:** Process PID=1 is put back into the run queue again and dispatched for further execution.
- **Memory Read Operations:** Multiple read system calls occur for offsets 0, 1, and 2 in Region 1.
- **Memory State After Read:**
  - At offset 0, the read value is 80.
  - At offset 1, the read value is 48.
  - At offset 2, the read value is -1.
- System Call: The memory contents are retrieved from Region 1, and the system updates the memory table accordingly.

Time Slot 15:

- **Process Finished:** Process PID=1 completes execution and is marked as finished.
- **CPU Stopped:** The CPU halts as there are no more processes to execute.
- **System Call:** The completion of the process triggers the system to stop the CPU.

3.4 Overall

In this section, we provide a comprehensive analysis of the simulation results, focusing on the performance of the operating system simulation, process management, memory allocation, and system call functionality.

Process Management: The operating system simulation effectively managed process execution, ensuring that processes were dispatched and executed based on their scheduling. The processes were loaded, queued, and dispatched in the correct time slots, with the system successfully handling both the arrival of new processes and the termination of completed ones. The system's ability to efficiently queue and manage processes demonstrates its core functionality in process scheduling.

Memory Allocation: The memory management system worked effectively throughout the simulation. Memory allocation for each process occurred in designated regions, with physical memory properly mapped to virtual addresses. As processes were dispatched, the page tables were updated accordingly, and the system accurately performed memory writes and reads. The memory dumps show that data was correctly written to and read from specified memory addresses, confirming that memory management was functioning as expected. Additionally, the deallocation process worked well, with memory being freed when processes finished executing.

System Calls: System calls, such as memory read and write operations, were performed correctly, allowing processes to interact with the operating system as intended. The simulation demonstrated how system calls are used to manage memory, with the ability to modify memory regions through writing data and reading back values at specific offsets. These operations were successfully handled by the system, showing that system calls were well-implemented in the simulation.

Performance Evaluation: The performance of the operating system was generally satisfactory. Memory allocation was handled efficiently, and processes were managed with minimal overhead. However, there was room for optimization in areas such as memory management, where performance could be improved by fine-tuning the handling of memory pages and frames. Additionally, while the system's process scheduling worked well, it could benefit from further enhancements

to prioritize high-priority processes more effectively.

Issues and Potential Improvements: While the overall performance was acceptable, there were some areas where further improvements could be made. For instance, the system's ability to handle processes with varying priority levels could be optimized by implementing a more sophisticated scheduling algorithm. Additionally, memory management could be optimized to reduce fragmentation and improve the overall speed of memory allocation and deallocation.

Conclusion: Overall, the operating system simulation provided a solid demonstration of core OS concepts such as process management, memory allocation, and system calls. The simulation's results indicate that the system can efficiently manage processes and memory, though further optimizations in process prioritization and memory management could enhance its performance. These improvements would make the system more efficient and scalable for handling larger workloads in real-world applications.

4 Questionnaire

4.1 Question 1

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

Answer: The scheduling algorithm is a simplified MLQ, based on the Linux kernel's MLQ without feedback.

- **Simplicity:** The assignment simplifies the MLQ by assigning each priority level to a single ready queue, avoiding the complexity of grouping levels into subsets (as potentially done in the real Linux kernel). This makes it easier to implement and understand.
- **Priority-based:** MLQ inherently prioritizes processes. Processes with higher priority (lower numerical value in this assignment) get preferential treatment, reducing the average waiting time for important tasks.
- **Fixed Time Slots:** The system is designed around the idea of traversing the list of ready queues in fixed steps. The size of the step is a function of the process's priority, which ensures that no process is starved and resource allocation is fair.

MLQ offers better control than FCFS/SJF and allows varying time slices unlike Round Robin.

4.2 Question 2

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

Answer: Multiple memory segments provide:

- **Logical Organization:** Segmentation allows the virtual address space to be divided into logical units (code, data, stack, heap). This improves code organization and maintainability.
- **Protection:** Segmentation can facilitate memory protection. Different segments can have different access rights (read-only, execute-only, etc.), preventing processes from accidentally overwriting code or data in other segments. While the assignment doesn't elaborate on explicit protection mechanisms, the separation lays the groundwork.
- **Simplified Memory Management:** By managing memory in segments, allocation and deallocation can be more efficient compared to a single large memory block.
- **Virtual Memory Support:** Segmentation is a key component of virtual memory systems, allowing processes to have a larger virtual address space than the available physical memory.
- **Dynamic Allocation:** Segmentation supports dynamic allocation.

4.3 Question 3

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

Answer: Dividing the address into more than two levels in a paging system (e.g., a multi-level page table) offers the following:

- **Reduced Page Table Size:** Multi-level page tables reduce the memory footprint of page tables, particularly for processes with sparse address spaces. By having multiple levels, you only need to allocate page table entries for the portions of the address space that are actually in use.
- **Improved Memory Efficiency:** Because not all page table entries need to be resident in memory at all times, multi-level paging can improve overall memory efficiency. Only the necessary levels are loaded into memory.

- **Increased Complexity:** The trade-off is increased complexity in address translation. Each level of the page table requires a memory access, which can slow down the translation process.
- **Increased Overhead:** The larger number of required steps introduces overhead to access data.

4.4 Question 4

Question: What are the advantages and disadvantages of segmentation with paging?
Answer: Segmentation with paging combines the benefits of both techniques:
Advantages:

- **Logical Organization and Protection (Segmentation):** Segmentation provides a logical view of memory, allowing for protection and sharing at the segment level.
- **Efficient Memory Utilization (Paging):** Paging eliminates external fragmentation and simplifies memory allocation by using fixed-size pages.
- **Virtual Memory:** Both segmentation and paging support virtual memory, allowing processes to have address spaces larger than physical memory.

Disadvantages:

- **Complexity:** Combining segmentation and paging increases the complexity of the memory management system. The system must manage both segment tables and page tables.
- **Overhead:** Address translation becomes more complex, potentially increasing overhead.
- **Internal Fragmentation (Paging):** Paging can lead to internal fragmentation, where a page is not fully utilized by a segment.

4.5 Question 5

Question: What is the mechanism to pass a complex argument to a system call using the limited registers?
Answer: When the number of registers is insufficient to pass all arguments to a system call, the following mechanisms are typically used:

- **Passing a Pointer:** The most common approach is to pass a pointer to a data structure in memory that contains the complex arguments. The system call can then access the data structure through the pointer. Only the pointer (a memory address) needs to be passed in a register.
- **Using a Stack:** Arguments can be placed on the stack, and a pointer to the stack frame can be passed to the system call.
- **Message Passing:** In some systems, arguments can be passed using message passing mechanisms. The arguments are packaged into a message, and the message is sent to the system call handler.

4.6 Question 6

Question: What happens if the syscall job implementation takes too long execution time?
Answer: If a system call takes too long to execute, several issues can arise:

- **System Unresponsiveness:** The system may become unresponsive, as other processes are blocked waiting for the system call to complete.
- **Starvation:** Processes waiting for the system call to finish may be starved of CPU time.
- **Priority Inversion:** A low-priority process making a long-running system call can block higher-priority processes that need to access the same resources.
- **Deadlock:** If the system call involves acquiring multiple resources, it can lead to a deadlock if another process is waiting for one of those resources.

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: If synchronization is not handled, you will encounter race conditions and data corruption.

- **Race Conditions:** Multiple processes or threads may access and modify shared data concurrently, leading to unpredictable and incorrect results.

- **Data Corruption:** Shared data structures can become corrupted if multiple processes modify them simultaneously without proper synchronization.
- **Inconsistent State:** The system's state can become inconsistent, leading to errors and crashes.

Example: Suppose two processes are incrementing a shared counter variable without any synchronization mechanisms like mutexes or semaphores.

- Process A reads the counter (e.g., counter = 5).
- Process B reads the counter (e.g., counter = 5).
- Process A increments the counter in its register (e.g., 5 + 1 = 6).
- Process B increments the counter in its register (e.g., 5 + 1 = 6).
- Process A writes the value back to memory (counter = 6).
- Process B writes the value back to memory (counter = 6).

Although the counter was incremented twice, the final value is 6 instead of 7. This is a race condition that arises due to the lack of synchronization. The document indicates the need to design and implement synchronization primitives in your Simple OS to handle similar concurrency issues.

5 Conclusion

In this assignment, we have successfully implemented a simplified version of the Multi-Level Queue (MLQ) scheduling algorithm, inspired by the Linux kernel's MLQ, but without feedback mechanisms. This design allows us to efficiently manage CPU resources by categorizing processes into multiple priority queues, ensuring that higher-priority tasks are given preferential treatment, thus reducing the average waiting time.

The scheduler works by selecting processes from different queues based on their priority levels, and by using a fixed time slice for each process. This ensures fairness in resource allocation, preventing process starvation. The simplicity of this design makes it easier to understand and implement, compared to more complex scheduling algorithms like feedback-based MLQ.

The memory management system, based on segmentation, offers significant benefits such as logical organization of memory, protection against unwanted access, and simplified memory allocation. By dividing memory into logical units, it ensures better utilization and scalability, especially when dealing with large processes.

Additionally, we discussed the implementation of multi-level paging and its effects on memory efficiency, trade-offs related to performance, and complexity. The use of multiple levels of page tables can reduce memory usage but may introduce additional overhead during address translation.

Another important aspect covered in this assignment is thread synchronization. We demonstrated the potential issues that arise when synchronization is not handled, such as race conditions and data corruption, and highlighted the importance of using synchronization mechanisms like mutex locks to ensure the safe access to shared resources.

In conclusion, this assignment has provided a hands-on understanding of key operating system components such as scheduling, memory management, and synchronization. The knowledge gained from this implementation serves as a foundation for understanding more advanced operating system features and provides insight into the challenges faced when building efficient systems for managing resources in a multi-tasking environment.