Ho Chi Minh City University of Technology

Faculty of Computer Science and Engineering

OPERATING SYSTEMS

Simple Operating System

Authors:	*Student's ID:*
Nguyen Thanh Thao Nhi	2152840
Nguyen Phuc Anh	2153166
Pham Tran Thanh Trung	2153929

Instructors:

Le Thanh Van

Completion date: 15th December, 2023

Contents

1 Introduction

1.1 Contributions

ID	Name	Contributions
2152840	Nguyen Thanh Thao Nhi	Questions + Memory Management + Report
2153166	Nguyen Phuc Anh	Memory Management + Report
2153929	Pham Tran Thanh Trung	Scheduler + Report

1.2 Overview

This report encapsulates an exploration into the core components of a simulated basic operating system. The focus revolves around three fundamental modules: scheduler, synchronization, and the mechanism of memory allocation from virtual to physical memory.

2 Scheduler

2.1 Questions and Answers

Question 1

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

Answer:

1. Priority queue and its advantages:
    - A priority queue arranges elements based on their priority values, where higher priority elements are processed or removed before lower priority ones.
    - Priority queues are used to select the next process to run, ensuring high-priority tasks run before low-priority ones. This ensures that critical tasks are completed first, which can be very important in real-time systems.
    - The advantage of a priority queue in scheduling lies in its adaptability, efficiency in managing varying priorities, and its ability to ensure that high-priority tasks are handled promptly, optimizing resource utilization and system responsiveness.
2. During theory lectures, there are 6 basic scheduling algorithms, including:
    - **First-Come First-Served (FCFS):** A non-preemptive scheduling algorithm where processes are executed in the order they arrive in the ready queue.
    - **Shortest Job Next (SJN) or Shortest Remaining Time First (SRTF):** SJN (or SRTF for preemption) schedules the shortest job or process next to

optimize for minimal remaining processing time.

- **Round Robin (RR):** A preemptive scheduling algorithm where each process is assigned a fixed time slice or quantum to execute before being moved to the back of the queue.
- **Priority Scheduling:** Priority scheduling assigns priorities to processes and executes higher-priority processes first.
- **Multi-level Queue (MLQ):** MLQ segregates processes into different queues based on certain criteria (e.g., priority, process type).
- **Multi-level Feedback Queue (MLFQ):** MLFQ is a dynamic scheduling algorithm with multiple queues, where processes move between queues based on their behaviour.

3. Advantages of Priority Queue over Other Scheduling Algorithms:
   - **Compared to FCFS:** Unlike First-Come, First-Served, suitable for systems with low process arrival rates and short execution times where tasks are processed based on their arrival order, a priority queue allows for dynamic assignment and adjustment of priorities. It's **more adaptive to varying task importance or urgency** and can result in **shorter waiting times** for high-priority processes.
   - **Compared to SJN/SRTF:** While Shortest Job Next or Shortest Remaining Time First, which improves upon FCFS in terms of average waiting time, prioritizes shorter tasks, a priority queue allows tasks with critical importance to be handled ahead of shorter ones if they are deemed more urgent, leading to **optimization** and **more efficient resource utilization**.
   - **Compared to Round Robin:** Round Robin ensures fairness (ensure a large number of tasks to be executed and each task gets a fair share of CPU time), but doesn't prioritize based on task importance. Priority queues enable the execution of critical tasks ahead of less critical ones, ensuring the timely processing of high-priority tasks, resulting in **preventing delays** and **flexibility in task execution**.
   - **Compared to Priority Scheduling:** Priority queues share similarities with Priority Scheduling but often offer **more granularity and flexibility in managing priorities**. They are adaptable to systems where tasks have multiple priority levels, leading to **better service quality**.
   - **Compared to MLQ and MLFQ:** While Multi-level Queue and Multi-level Feedback Queue manage tasks in different queues with criteria like priority or behaviour, priority queues offer **more seamless management of tasks with varying priorities within a single queue structure**.

4. Conclusion:
    - The advantage of a priority queue in scheduling is its ability to **efficiently manage and process tasks** based on their assigned priorities. Unlike some traditional scheduling algorithms, a priority queue's **adaptability** to varying priorities ensures that critical or high-priority tasks receive prompt attention, leading to **optimized resource utilization** and **responsiveness** in systems where task priorities play a crucial role.

2.2 Implementation

For the scheduler, we need to implement 3 function:

- enqueue()
- dequeue()
- get_mlq_proc()

2.2.1 enqueue()

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

Comment: This function will add a new process to the queue with a maximum size of **MAX_QUEUE_SIZE**. It first checks if there is an available slot in the queue. If so, add the process to the end of the queue. If the slot is full, the function will print "**FULL SLOT**".

2.2.2 dequeue()

```
struct pcb_t * dequeue(struct queue_t * q) {
/* TODO: return a pcb whose priority is the highest
* in the queue [q] and remember to remove it from q
* */
    if (q->size == 0) return NULL;
    struct pcb_t* rm_proc = q->proc[0];
    for (int i = 0; i < q->size; i++) {
        if (i + 1 == MAX_QUEUE_SIZE) {
            q->proc[i] = NULL;
```

```
        }
        else q->proc[i] = q->proc[i + 1];
    }
    q->size--;
    return rm_proc;
}
```

Comment: This function is used to retrieve and remove a process from the queue. It specifically operates on a queue containing processes with the same priority. In this case, we remove and return the queue's first element, represented by **q** $\rightarrow$ **proc[0]**. Additionally, all the elements in the queue are shifted one position to the left to maintain the order. The queue **size** is also reduced by 1.

2.2.3 get_mlq_proc()

```
struct pcb_t * get_mlq_proc(void) {
   struct pcb_t * proc = NULL;
/*TODO: get a process from PRIORITY [ready_queue].
* Remember to use lock to protect the queue.
* */
    static unsigned long slot[MAX_PRIO]={0};
    unsigned long prio=0;
    pthread_mutex_lock(&queue_lock);
    while(proc==NULL){
      if(!empty(&mlq_ready_queue[prio]) && slot[prio]<MAX_PRIO-prio){
         proc=dequeue(&mlq_ready_queue[prio]);
         slot[prio]++;
      }
      else{
         prio++;
         if(prio==MAX_PRIO){
            prio=0;
            for(int i=0;i<MAX_PRIO;i++){
               slot[i]=0;
            }
            break;
         }
      }
    }
    pthread_mutex_unlock(&queue_lock);

    return proc;
}
```

Comment: This function is implemented to dequeue a process at the front of the highest priority non-empty queue. It also reduces the **slot**, which limits the resource the queue is currently holding.

- The function iterates through the priority levels of the multi-level queue. At each level, it checks if the current queue is not empty and if the number of processes selected from that level (**slot[prio]**) is less than **MAX_PRIO - prio**. If both conditions are true, it selects a process from that queue and increments the **slot[prio]** counter.
- If the number of selected processes from the current level reaches **MAX_PRIO - prio**, it means that the maximum number of processes allowed for that level has been reached. In that case, the function moves to the next lower priority level.
- When the function reaches the lowest priority level (level **MAX_PRIO**), it resets all the slot counters for each priority level to 0. This ensures that the function starts selecting processes from the highest priority level again in a round-robin manner.

2.2.4 Dual priority mechanism

In this mechanism, the default value can be overwritten by the live priority during process execution calling. To tackle the conflict, when it has priority in process loading (the input file), it will overwrite and replace the default priority in the process description file.

To implement this mechanism:

1. When reading the input file, use the ”fget()” method to read the entire line.
2. Check the number of elements. If the ”sscanf” function successfully reads three values, concatenate ”proc” to ”ld_processes.path[i]” using the ”strcat” function.
3. If the ”sscanf” function does not successfully read three values, check if it successfully reads two values.
4. If the ”sscanf” function successfully reads two values, open the file using ”ld_processes.path[i]” and assign the default priority to ”ld_processes.prio[i]”. Then close the file.

Therefore, if the input files do not include a third parameter (live priority) value, the mechanism will be activated by reading the default priority. Otherwise, the default priority is overwritten.

2.3 Output results

2.3.1 sched

Input:

```
4 2 3
0 p1s 1
```

```
1 p2s
2 p3s 7
```

Result:

```
================================================================
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: 20
   CPU 1: Dispatched process 2
================================================================
Time slot  2
   Loaded a process at input/proc/p3s, PID: 3 PRIO: 7
================================================================
Time slot  3
================================================================
Time slot  4
   CPU 0: Put process 1 to run queue
   CPU 0: Dispatched process 1
================================================================
Time slot  5
   CPU 1: Put process 2 to run queue
   CPU 1: Dispatched process 3
================================================================
Time slot  6
================================================================
Time slot  7
================================================================
Time slot  8
   CPU 0: Put process 1 to run queue
   CPU 0: Dispatched process 1
================================================================
Time slot  9
   CPU 1: Put process 3 to run queue
   CPU 1: Dispatched process 3
================================================================
Time slot 10
   CPU 0: Processed 1 has finished
   CPU 0: Dispatched process 2
================================================================
Time slot 11
```

```
============================================================
Time slot 12
============================================================
Time slot 13
   CPU 1: Put process 3 to run queue
   CPU 1: Dispatched process 3
============================================================
Time slot 14
   CPU 0: Put process 2 to run queue
   CPU 0: Dispatched process 2
============================================================
Time slot 15
============================================================
Time slot 16
============================================================
Time slot 17
   CPU 1: Put process 3 to run queue
   CPU 1: Dispatched process 3
============================================================
Time slot 18
   CPU 0: Processed 2 has finished
   CPU 0 stopped
============================================================
Time slot 19
============================================================
Time slot 20
============================================================
Time slot 21
   CPU 1: Put process 3 to run queue
   CPU 1: Dispatched process 3
============================================================
Time slot 22
   CPU 1: Processed 3 has finished
   CPU 1 stopped
```

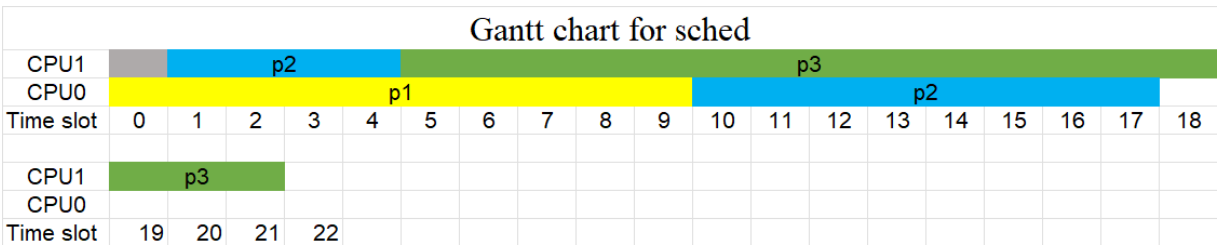

Figure 1: sched

Comment: As you can see, in sched output, we you two different formatted input:

- Process with live priority: 0 p1s 1, 2 p3s 7
- Process with default priority: 1 p2s

Which can show how dual priority mechanism of our program work.

2.3.2 sched_0

Input:

```
2 1 2
0 s0 12
4 s1 20
```

Result:

```
============================================================
Time slot  0
ld_routine
   Loaded a process at input/proc/s0, PID: 1 PRIO: 12
   CPU 0: Dispatched process 1
============================================================
Time slot  1
============================================================
Time slot  2
   CPU 0: Put process 1 to run queue
   CPU 0: Dispatched process 1
============================================================
Time slot  3
============================================================
Time slot  4
   Loaded a process at input/proc/s1, PID: 2 PRIO: 20
   CPU 0: Put process 1 to run queue
   CPU 0: Dispatched process 1
============================================================
Time slot  5
============================================================
Time slot  6
   CPU 0: Put process 1 to run queue
   CPU 0: Dispatched process 1
============================================================
Time slot  7
============================================================
Time slot  8
   CPU 0: Put process 1 to run queue
```

```
   CPU 0: Dispatched process 1
============================================================
Time slot  9
============================================================
Time slot 10
   CPU 0: Put process 1 to run queue
   CPU 0: Dispatched process 1
============================================================
Time slot 11
============================================================
Time slot 12
   CPU 0: Put process 1 to run queue
   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
   CPU 0: Processed 1 has finished
   CPU 0: Dispatched process 2
============================================================
Time slot 16
============================================================
Time slot 17
   CPU 0: Put process 2 to run queue
   CPU 0: Dispatched process 2
============================================================
Time slot 18
============================================================
Time slot 19
   CPU 0: Put process 2 to run queue
   CPU 0: Dispatched process 2
============================================================
Time slot 20
============================================================
Time slot 21
   CPU 0: Put process 2 to run queue
   CPU 0: Dispatched process 2
============================================================
Time slot 22
   CPU 0: Processed 2 has finished
   CPU 0 stopped
```

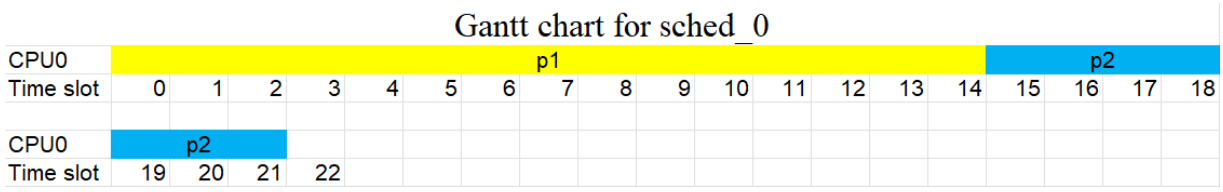

Figure 2: sched_0

2.3.3 sched_1

Input:

```
2 1 4
0 s0 12
4 s1 20
6 s2 20
7 s3 7
```

Result:

```
============================================================
Time slot  0
ld_routine
   Loaded a process at input/proc/s0, PID: 1 PRIO: 12
   CPU 0: Dispatched process 1
============================================================
Time slot  1
============================================================
Time slot  2
   CPU 0: Put process 1 to run queue
   CPU 0: Dispatched process 1
============================================================
Time slot  3
   Loaded a process at input/proc/s1, PID: 2 PRIO: 20
============================================================
Time slot  4
   CPU 0: Put process 1 to run queue
   CPU 0: Dispatched process 1
============================================================
Time slot  5
   Loaded a process at input/proc/s2, PID: 3 PRIO: 20
============================================================
Time slot  6
```

```
   Loaded a process at input/proc/s3, PID: 4 PRIO: 7
   CPU 0: Put process 1 to run queue
   CPU 0: Dispatched process 4
============================================================
Time slot  7
============================================================
Time slot  8
   CPU 0: Put process 4 to run queue
   CPU 0: Dispatched process 4
============================================================
Time slot  9
============================================================
Time slot 10
   CPU 0: Put process 4 to run queue
   CPU 0: Dispatched process 4
============================================================
Time slot 11
============================================================
Time slot 12
   CPU 0: Put process 4 to run queue
   CPU 0: Dispatched process 4
============================================================
Time slot 13
============================================================
Time slot 14
   CPU 0: Put process 4 to run queue
   CPU 0: Dispatched process 4
============================================================
Time slot 15
============================================================
Time slot 16
   CPU 0: Put process 4 to run queue
   CPU 0: Dispatched process 4
============================================================
Time slot 17
============================================================
Time slot 18
   CPU 0: Put process 4 to run queue
   CPU 0: Dispatched process 4
============================================================
Time slot 19
============================================================
Time slot 20
   CPU 0: Put process 4 to run queue
   CPU 0: Dispatched process 4
```

```
================================================================
Time slot 21
================================================================
Time slot 22
   CPU 0: Put process 4 to run queue
   CPU 0: Dispatched process 4
================================================================
Time slot 23
   CPU 0: Processed 4 has finished
   CPU 0: Dispatched process 1
================================================================
Time slot 24
================================================================
Time slot 25
   CPU 0: Put process 1 to run queue
   CPU 0: Dispatched process 1
================================================================
Time slot 26
================================================================
Time slot 27
   CPU 0: Put process 1 to run queue
   CPU 0: Dispatched process 1
================================================================
Time slot 28
================================================================
Time slot 29
   CPU 0: Put process 1 to run queue
   CPU 0: Dispatched process 1
================================================================
Time slot 30
================================================================
Time slot 31
   CPU 0: Put process 1 to run queue
   CPU 0: Dispatched process 1
================================================================
Time slot 32
   CPU 0: Processed 1 has finished
   CPU 0: Dispatched process 2
================================================================
Time slot 33
================================================================
Time slot 34
   CPU 0: Put process 2 to run queue
   CPU 0: Dispatched process 3
================================================================
```

```
Time slot 35
=============================================================
Time slot 36
   CPU 0: Put process 3 to run queue
   CPU 0: Dispatched process 2
=============================================================
Time slot 37
=============================================================
Time slot 38
   CPU 0: Put process 2 to run queue
   CPU 0: Dispatched process 3
=============================================================
Time slot 39
=============================================================
Time slot 40
   CPU 0: Put process 3 to run queue
   CPU 0: Dispatched process 2
=============================================================
Time slot 41
=============================================================
Time slot 42
   CPU 0: Put process 2 to run queue
   CPU 0: Dispatched process 3
=============================================================
Time slot 43
=============================================================
Time slot 44
   CPU 0: Put process 3 to run queue
   CPU 0: Dispatched process 2
=============================================================
Time slot 45
   CPU 0: Processed 2 has finished
   CPU 0: Dispatched process 3
=============================================================
Time slot 46
=============================================================
Time slot 47
   CPU 0: Put process 3 to run queue
   CPU 0: Dispatched process 3
=============================================================
Time slot 48
=============================================================
Time slot 49
   CPU 0: Put process 3 to run queue
   CPU 0: Dispatched process 3
```

```
============================================================
Time slot 50
============================================================
Time slot 51
   CPU 0: Put process 3 to run queue
   CPU 0: Dispatched process 3
============================================================
Time slot 52
   CPU 0: Processed 3 has finished
   CPU 0 stopped
```

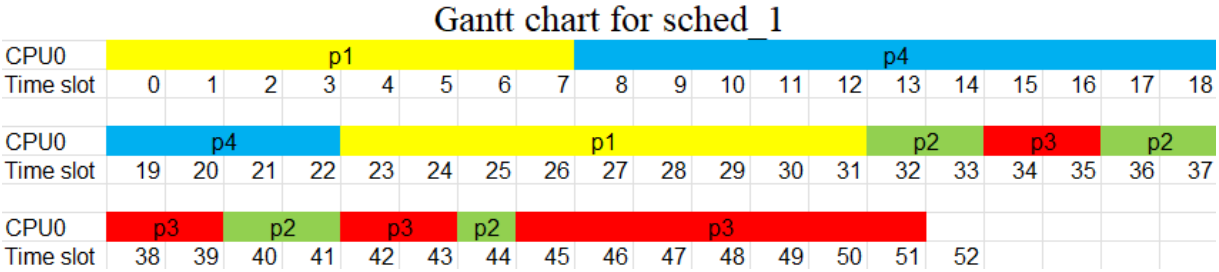

Figure 3: sched_1

2.4 Output explanations

- Our scheduler follows a **Multi-Level Queue (MLQ)** system where each priority level has its own queue. Each queue has a maximum capacity of 10 processes, and the number of slots available for each priority level is determined based on the priority value.
- When a queue has exhausted all its slots or has completed all the processes within it, the scheduler moves on to the next priority level, giving higher priority processes precedence over lower priority ones.
- The scheduler also considers a time slice for each process in a priority queue. This time slice determines the maximum duration a process can run before it is dequeued from the ready queue and enqueued again. This ensures fairness by allowing each process to have a fair share of CPU time.
- When a queue's slots are fully utilized, regardless of whether the time slice has been fully utilized or not, the system switches the resource to the next queue and leaves the remaining work for future slots. This ensures that processes at lower priority levels are given the opportunity to execute, even if a process at a higher priority level has not finished within its time slice.
- To make it clear about how scheduler work, let consider sched_1 again:

Input:

```
2 1 4
0 s0 12
4 s1 20
6 s2 20
7 s3 7
```

Comment: Meaning of input file:

- Time slice: 2
- Number of CPU: 1
- Number of Process: 4
- And others can be see in the table below

Process ID (PID)	Priority (PRIO)	Arrival Time
1 (P1)	12	0
2 (P2)	20	4
3 (P3)	20	6
4 (P4)	7	7

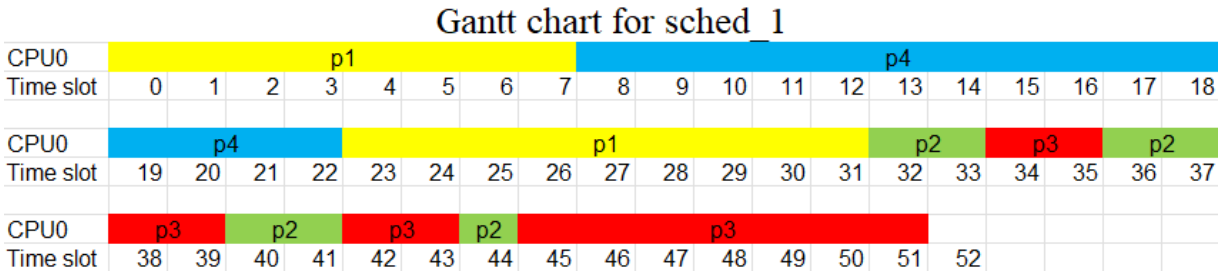

Figure 4: sched_1

Explaination:

1. Process 1 load to the queue and start to do work at time slot 0. Every 2 time units, it will dequeue the process from ready_queue and enqueue it finish. Because there aren't any process have same priority so it continue to run
2. Process 2 load to the queue at time slot 4. Since the process have lower priority than process 1, process 1 continue to run.
3. Process 3 load to the queue at time slot 6. Since the process have lower priority than process 1, process 1 continue to run.
4. Process 4 load to the queue at time slot 7. Since the time slice used up and process 4 have higher priority them it start to run by the CPU while process 1 dequeue and enqueue to the same level queue.

5. Since there are no processes have higher priority, process 1 run until finish at time slot 22.
6. After that, process 1 continue to run because it have highest priority.
7. At time slot 32, process 1 finish the work. Since process 2 and process 3 have the same priority, they will run alternate (which mean when time slice used up process 2 dequeue then enqueue to the same level queue again while process 3 start to run, vice versa).
8. At time slot 44, process 2 done the work.
9. Process 3 continue to run until finish its work at time slot 52.

3 Memory Management

3.1 Questions and Answers

Question 2

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:

1. Multiple memory segments or memory areas:
   - A memory segment is **a contiguous block of memory** in a computer system that has been allocated for a specific purpose. In a typical program, there may be separate memory segments allocated for code (executable instructions), data (initialized and uninitialized variables), and a stack (used for function calls and local variable storage). Each segment may have its own size, location in memory, and access permissions.
   - Multiple memory segments or memory areas refer to **a memory management technique** where the memory space is divided into distinct and separate segments, each allocated for specific purposes or types of data within an operating system. This approach aids in efficient memory management, system stability, security, and flexibility in handling memory resources within an operating system.
2. There are several advantages of multiple memory segments or memory areas:
   - **Efficient Resource Allocation:** Segregating memory into distinct segments allows for precise allocation based on specific needs. Each segment can be tailored for different types of data or program components, optimizing resource allocation.

- **Enhanced Memory Management:** Dividing memory into segments facilitates better control and management. It enables efficient tracking, allocation, and deallocation of memory, minimizing wastage and fragmentation.
- **Improved Security:** Segments can be assigned varying access permissions, enhancing security by restricting unauthorized access to critical memory areas.
- **Improved System Stability:** Separate memory segments prevent conflicts between different parts of the memory, reducing the risk of interference or corruption. This isolation contributes to enhanced system stability, ensuring smoother operations.
- **Reduced Internal Fragmentation:** Allocating memory into separate segments minimizes internal fragmentation, resulting in more contiguous and usable memory blocks. This aids in optimizing memory allocation and reducing inefficiencies.
- **Flexibility and Adaptability:** The segmented design allows for flexibility in adjusting memory layout and configurations as per evolving system requirements. It supports easy modifications or additions of segments to accommodate changes.
- **Organized Memory Structure:** Dividing memory into distinct segments allows for clearer organization and tracking of memory usage within the operating system. Each segment serves a specific purpose, facilitating better management and maintenance of memory.
- **Simplified Maintenance:** With clearly defined segments, it becomes easier to monitor and maintain different memory areas. This structured approach streamlines troubleshooting and debugging processes.

3. Conclusion:
    - Overall, employing multiple memory segments in the OS enhances resource utilization, strengthens security measures, enables flexibility in memory handling, and streamlines maintenance processes, contributing to a more robust and efficient operating system.

Question 3

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

Answer:

1. What is Memory Management?

- Memory management is a method in the operating system to manage operations between main memory and disk during process execution, involves managing the allocation and deallocation of memory space for programs and applications that require it. It ensures efficient use of main memory, allowing multiple processes to run simultaneously.
- Memory management techniques in operating systems include contiguous allocation, paging, segmentation, segmentation with paging, and virtual memory. Each technique offers unique benefits but also brings specific complexities and limitations, impacting system efficiency based on diverse needs and trade-offs.

2. What is Paging Memory Management?
   - Paging is a memory management scheme used by operating systems to handle memory allocation in a non-contiguous manner.
   - In this system, the physical memory is divided into fixed-size blocks called page frames, and the logical memory of processes is also divided into fixed-size blocks called pages. When a process requires memory, the operating system allocates page frames to it and maps the process's logical pages to these physical page frames using a data structure called a page table.
3. What is Muti-level Paging Memory Management?
   - Paging can be implemented using a single-level page table or a multi-level page table. Single-level page tables directly map logical to physical addresses in a single table, while multi-level page tables break down the page table into a hierarchy of tables and uses multiple levels of indirection to map logical addresses to physical addresses, offering scalability with slightly more complexity.
   - For instance, a two-level page table might have an outer page table (or page directory for the first level) that points to a set of inner page tables (or page tables for the second-level tables). The page tables then map specific page numbers to physical page frames.
4. When the address is divided into more than two levels in a paging memory management system:
   - **Basic Mechanism:** Fundamental concepts like the mechanism of paging, the principles of virtual memory, memory protection methods, the role of page tables in address translation, and the consistent page size remain unaffected by the addition of more levels in the page table hierarchy. These foundational elements persist regardless of changes in the hierarchy's structure, ensuring the core principles of memory management **remain constant**.
   - **Page Table and Address Structure:** The page table structure would become more complex as more levels are added. Each level would need its own

set of page tables, increasing the overall number of page tables. The address structure will also have additional fields to represent each level in the page table hierarchy. Therefore, the structure and organization of the page tables **change**.

- **Advantages:**
  - **Efficient Management:** In systems with very large address spaces, using multiple levels allows for more efficient and organized management of memory. It enables the system to handle vast amounts of data without overwhelming a two-level paging system.
  - **Reduced Memory Overhead:** More levels allow for smaller, more specialized page tables, which potentially reduces the memory required to store the page tables, particularly in systems with large address spaces. Smaller page tables mean less memory overhead.
  - **Enhanced Memory Utilization:** More levels enable finer granularity in addressing, allowing for more precise allocation of memory, leading to better memory utilization by minimizing internal fragmentation.
  - **Scalability:** With more levels, the system becomes more scalable as it can accommodate larger address spaces efficiently. It becomes easier to manage memory in systems with expansive memory requirements.
  - **Adaptability:** Multi-level paging provides flexibility in choosing different page sizes at various levels. This flexibility can help optimize memory usage based on the specific needs of different parts of the address space.
- **Disadvantages:**
  - **Increased Access Time:** Each additional level in the page table hierarchy introduces an extra memory lookup during address translation, resulting in increased latency for memory access since the system has to traverse multiple levels of page tables to map the virtual address to a physical one. Systems with excessive levels might suffer from notably slower performance.
  - **Complexity:** Adding more levels significantly increases the complexity of the page table structure and memory management. More levels mean more page tables, leading to increased overhead in managing and maintaining these tables. Debugging, system maintenance, and overall management become more challenging.
  - **Fragmentation Issues:** Multi-level paging systems might face fragmentation problems, especially if the page sizes at different levels don't align well with memory allocation patterns, resulting in wasted memory space

due to inefficient allocation.

- **Limited Benefits:** Beyond a certain point, the benefits of adding more levels might not justify the increased complexity and latency. In many cases, the overhead introduced by excessive levels might outweigh the advantages in memory utilization or scalability.
- **Balancing Act:** The decision to use more levels involves a trade-off between optimizing memory utilization and managing the increased complexity and potential performance degradation. Often, finding the right balance between these factors is crucial.

5. Conclusion:

- Dividing the address into more than two levels in the paging system offers benefits in terms of managenent, memory overhead, memory utilization, scalability and adaptability. However, these advantages come at the cost of increased access time due to additional memory lookups, increased system complexity, and potential fragmentation issues.
- In modern computer systems, two-level paging is the most common choice for memory management due to its balanced approach between efficiency and complexity. It's practical for various systems with moderate to large address spaces. In some cases of extremely large address spaces, three-level paging might be chosen to handle memory more efficiently.

Question 4

What is the advantage and disadvantage of segmentation with paging?

Answer:

1. What is segmentation with paging?

- Segmentation and paging are two memory management techniques used by operating systems to manage memory allocation. Segmentation divides the memory space into variable-sized segments, while paging divides memory into fixed-sized blocks called pages. Segmentation with paging is a hybrid of these two techniques, where the logical address space is divided into segments, and each segment is further divided into fixed-size pages

2. Advantages:

- **Flexible Memory Management:** Segmentation with paging allows the logical address space of a process to be divided into segments, and each segment can further be divided into pages. By breaking the memory into segments and pages, this approach caters to diverse memory requirements efficiently.

- **Reduced Memory Overhead:** The segment table in segmentation with paging contains fewer entries compared to pure segmentation, meaning less memory consumption. By linking segments directly to their respective page tables, it reduces the amount of information stored in the segment table.
- **External Fragmentation Mitigation:** Fixed-size pages manage memory allocation instead of variable-sized segments, reducing the risk of creating unnecessary gaps in memory. Using fixed-size pages helps avoid memory fragmentation, ensuring better use of available memory.
- **Demand Paging Capability:** Demand paging loads pages into memory only when needed, optimizing memory usage by loading essential pages rather than the entire program. Loading only necessary pages conserves memory resources, as it fetches data on-demand, reducing initial load times and improving overall memory efficiency.
- **Effective Protection Mechanism:** Segment-level protection allows different segments to have distinct access permissions, preventing unauthorized access. Page-level protection safeguards individual pages, preventing accidental or malicious modifications, and ensuring data integrity. This provides both segment-level and page-level protection, enhancing system security.
- **Memory Sharing Efficiency:** Segments can be shared among different processes, allowing for efficient utilization of memory resources by avoiding unnecessary duplication of data or code.

3. Disadvantages:

   - **Fragmentation:** Fixed-size pages within segments can lead to internal fragmentation, where memory within a page might remain unused. Fixed-size pages ensure uniform memory allocation but might not perfectly match the memory needs of all data, resulting in inefficient usage and wastage of memory space. Also, improper sizing of segments may lead to fragmentation issues, causing inefficient use of memory space and hindering memory allocation.
   - **Increased Complexity:** Managing the interaction between segments and pages, along with maintaining segment and page tables, adds complexity to system design and operation, demanding more hardware support and intricate handling. Additional hardware components elevate system complexity, potentially affecting system efficiency and demanding more resources.

4. Conclusion:

   - Segmentation with paging offers a versatile approach to memory management, delivering benefits such as flexible memory allocation, reduced external fragmentation, enhanced protection mechanisms, and efficient memory sharing.

However, challenges like fragmentation issues and system complexity need consideration.

3.2 Implementation

In this memory section, we first move on **mm-memphy.c** file, which implements the physical memory part in paging-based memory management. In this file, we implement the function that is used to show the content of physical memory:

- **MEMPHY_dump**

Next, we move on to the **mm-vm.c** file, as the same as the previous file, but in this case, we deal with the virtual memory part. In this file, we need to implement 4 functions:

- **__alloc**, **__free**
- **pg_getpage**
- **find_victim_page**
- **validate_overlap_vm_area**

Finally, the **mm.c** file, this is the file that handles the memory management. In this file contains two functions that we need to implement:

- **vmap_page_range**
- **alloc_pages_range**

3.2.1 MEMPHY_dump

In this assignment, we want to see the RAM content and just print the addresses whose content doesn't equal 0 because of large-size memory.

```
int MEMPHY_dump(struct memphy_struct *mp)
{
   /*TODO dump memphy content mp->storage
    *     for tracing the memory content
    */
   for(long int it; it < mp->maxsz; it++) {
      if(mp->storage[it] != 0) {
         printf("0x%08lx: %08x\n", it, mp->storage[it]);
      }
   }
   printf("\n");
   return 0;
}
```

Comment: This function iterates through the storage array within the **memphy_struct** and prints out non-zero values along with their respective memory addresses.

3.2.2 __alloc

The syntax for memory allocation is as: `alloc [size] [reg]`, where `[size]` is the size of expeccted allocating region and `[reg]` is the register stored the starting address of the newly allocated region, which is the index of the **Symbol table**. The **__alloc** function is implemented to allocate a memory region.

```
int __alloc(struct pcb_t *caller, int vmaid, int rgid, int size, int
    *alloc_addr)
{
  /*Allocate at the toproof */
    struct vm_rg_struct rgnode; // Only use this when exist free region

    // Exist free vm_rg in the listl
    if (get_free_vmrg_area(caller, vmaid, size, &rgnode) == 0)
    {
        caller->mm->symrgtbl[rgid].rg_start = rgnode.rg_start;
        caller->mm->symrgtbl[rgid].rg_end = rgnode.rg_end;

        *alloc_addr = rgnode.rg_start;

        return 0;
    }
    // Don't exist then increase the size of vm_area to create a new rg
    /* TODO get_free_vmrg_area FAILED handle the region management (Fig.6)*/

    /*Attempt to increate limit to get space */
    struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);
    // int inc_sz = PAGING_PAGE_ALIGNSZ(size);
    // int inc_limit_ret
    int old_sbrk;

    old_sbrk = cur_vma->sbrk;

    /* TODO INCREASE THE LIMIT
    * inc_vma_limit(caller, vmaid, inc_sz)
    */
    if (inc_vma_limit(caller, vmaid, size) == 0)
    {
        /*Successful increase limit */
        caller->mm->symrgtbl[rgid].rg_start = old_sbrk;
        caller->mm->symrgtbl[rgid].rg_end = old_sbrk + size;
```

```
        *alloc_addr = old_sbrk;
        caller->mm->mmap->sbrk = old_sbrk + size;

        return 0;
    }
    else
        return -1;
}
```

Comment:

First, check whether a free virtual memory region is available using **get_free_area** function. If exists, allocate a new region from that free list and update the symbol table accordingly. If there is currently no freed region, increase the virtual memory limit to obtain more free space by calling the **inc_vma_limit** function. If the limit is successfully increased, it assigns a new memory region within this expanded space to the requested rgid and updates the symbol table. If increasing the virtual memory limit fails or encounters an error, it returns **-1**, indicating a failure to allocate memory.

3.2.3 __free

This function is used to remove a memory region. That is, just to add the determined region to the freed region list.

```
int __free(struct pcb_t *caller, int vmaid, int rgid)
{

  if (rgid < 0 || rgid > PAGING_MAX_SYMTBL_SZ)
    return -1;

  /* TODO: Manage the collect freed region to freerg_list */
  struct vm_rg_struct *regRG = get_symrg_byid(caller->mm, rgid);
  struct vm_rg_struct *headFreeList = caller->mm->mmap->vm_freerg_list;
  while (headFreeList)
  {
    if (headFreeList->rg_start == regRG->rg_end)
    {
      headFreeList->rg_start = regRG->rg_start;
      regRG->rg_start = regRG->rg_end = 0;
      return 0;
    }
    else if (headFreeList->rg_end == regRG->rg_start) {
      headFreeList->rg_end = regRG->rg_end;
      regRG->rg_start = regRG->rg_end = 0;
```

```
      return 0;
    }
    headFreeList = headFreeList->rg_next;
  }
  struct vm_rg_struct *rgnode = (struct vm_rg_struct*)malloc(sizeof(struct
      vm_rg_struct));
  rgnode->rg_start = regRG->rg_start;
  rgnode->rg_end = regRG->rg_end;
  rgnode->rg_next = NULL;
  // Reset that symbol table
  regRG->rg_start = regRG->rg_end = 0;
  /*enlist the obsoleted memory region */
  enlist_vm_freerg_list(caller->mm, rgnode);

  return 0;
}
```

Comment: The function first checks whether the **rgid** is in a certain valid range. If **rgid** is not in this range, the function returns **-1**, indicating an invalid value. It then iterates through the existing list of free memory regions (**vm_freerg_list**) in the caller's memory management to find a suitable place to insert the newly freed region. After the freed region is enlisted, the function resets the start and end positions of the original memory region to 0 to mark it as free.

3.2.4 pg_getpage

This function is used to manage the retrieval of a specific page from RAM, ensuring that the required pages are present in RAM and handling swaps between RAM and the swap space when necessary to optimize memory usage.

```
int pg_getpage(struct mm_struct *mm, int pgn, int *fpn, struct pcb_t
    *caller)
{
  uint32_t pte = mm->pgd[pgn];
  int tgtfpn = PAGING_SWP(pte); // the target frame storing our variable
  // pte: change from SWAP mode to MEM mode

  if (!PAGING_PAGE_PRESENT(pte)) // The page we want is in the backing
      store
  {                             /* Page is not online, make it actively
      living */
    // Two case, exist free frame and not exist free frame.
    // There is free frame in MEM
    int fpn;
    if (MEMPHY_get_freefp(caller->mram, &fpn) == 0)
```

```
    {
      pte_set_fpn(&mm->pgd[pgn], fpn);
      __swap_cp_page(caller->active_mswp, tgtfpn, caller->mram, fpn);
    }
    else
    {
      int vicpgn, swpfpn;
      // vicpgn: pgn will change from MEM mode to SWAP mode
      // swpfpn : fpn in SWAP that vicpgn will put into
      // int vicfpn;
      // uint32_t vicpte;

      /* TODO: Play with your paging theory here */

      /* Find victim page */
      if(find_victim_page(caller->mm, &vicpgn) != 0){
        return -1;
      }

      /* Get free frame in MEMSWP */
      MEMPHY_get_freefp(caller->active_mswp, &swpfpn);

      /*Get frame from victim page*/
      int vicfpn = PAGING_FPN(vicpgn);

      /* Do swap frame from MEMRAM to MEMSWP and vice versa*/
      /* Copy victim frame to swap */
      __swap_cp_page(caller->mram, vicfpn, caller->active_mswp, swpfpn);
      // Update the victim page
      /* Copy target frame from swap to mem */
      __swap_cp_page(caller->active_mswp, tgtfpn, caller->mram, vicfpn);

      /* Update page table */
      // Change vicpgn
      pte_set_swap(&caller->mm->pgd[vicpgn], 0, swpfpn);

      /* Update its online status of the target page */
      // update the pte
      pte_set_fpn(&caller->mm->pgd[pgn], vicfpn);
    }

    enlist_pgn_node(&caller->mm->fifo_pgn, pgn);
  }

  *fpn = PAGING_FPN(pte);
```

```
    return 0;
  }
```

Comment:

First, check whether there exists a page table entry (PTE) for the requested page number. If PTE equals 0, the page has not yet been allocated or mapped. If the page exits, there are two primary scenarios: the page already present in RAM and the page not present in RAM.

- If the page is not online or present in memory (**PAGING_PAGE_PRESENT(pte)** evaluates to false), find a free frame available in the physical memory using **MEMPHY_get_freefp**. It then swaps the required page into the RAM and updates the page table accordingly.
- If there is no free frame available in the physical memory, it needs to swap pages between the RAM and the SWAP to make room for the requested page. It identifies a victim page by calling **find_victim_page** and then retrieves a free frame in the swap space using **MEMPHY_get_freefp**. To perform the necessary swapping, call **__swap_cp_page**. After the swapping is done, it updates the page table to reflect the changes.

3.2.5 find_victim_page

This function will find the victim page to be swapped. In this assignment, our group uses the **Least Recently Used (LRU)** page replacement algorithm, which will choose the page which has not been referenced for the longest time to be the victim page, ensuring that frequently used pages remain in memory to minimize the cost for page swaps.

```
int find_victim_page(struct mm_struct *mm, int *retpgn)
{
    struct pgn_t *pg = mm->fifo_pgn;

    /* TODO: Implement the theoretical mechanism to find the victim page */
    struct LRU_node *last_node = LRU_last_node(mm);
    *retpgn = last_node->LRU_pgn;

    free(pg);

    return 0;
 }
```

Comment:

It uses **LRU_last_node** to obtain the least recently used page from the stack to be the

victim page.

LRU_last_node: This function finds and returns the least recently used page from the LRU stack. It traverses the LRU stack to find the last node, which represents the least recently used page. Once the last node is found, it removes it from the stack by setting its next pointer to NULL. The function returns this last node, which is to be used as the victim page for page replacement.

```
struct LRU_node *LRU_last_node(struct mm_struct *mm)
{
  struct LRU_node **stackLRU = &(mm->LRU_stack_head);
  while (*stackLRU)
  {
    if (!(*stackLRU)->next_pgn)
      break;
    stackLRU = &(*stackLRU)->next_pgn;
  }
  struct LRU_node *last_node = *stackLRU;
  *stackLRU = NULL;
  return last_node;
}
```

3.2.6 validate_overlap_vm_area

In this assignment, we just use one virtual memory area, so there's no need to implement it. However, the idea of this function is to check whether two or more area is overlapping when creating a new region.

3.2.7 vmap_page_range

This function maps a range of pages to an aligned address for a given process. It utilizes physical frames to map pages and handles tracking for potential page replacement activities using an **LRU** stack.

```
  int vmap_page_range(
      struct pcb_t *caller,        // process call
      int addr,                    // start address which is aligned to
          pagesz
      int pgnum,                   // num of mapping page
      struct framephy_struct *frames, // list of the mapped frames
      struct vm_rg_struct *ret_rg) // return mapped region, the real mapped
          fp
  {// no guarantee all given pages are mapped
    int pgn = PAGING_PGN(addr); // Get the page number
    int pgit;
```

```
    ret_rg->rg_end = ret_rg->rg_start = addr; // at least the very first
        space is usable

    /* TODO map range of frame to address space
     *      [addr to addr + pgnum*PAGING_PAGESZ
     *      in page table caller->mm->pgd[]
     */
    for (int i = 0; i < pgnum; ++i)
    {
      uint32_t pte = 0;
      pte_set_fpn(&pte, frames->fpn);
      caller->mm->pgd[i + pgn] = pte;
      frames = frames->fp_next;
      update_LRU(caller->mm, i); // Put to LRU queue for page replacement alg
      pgit = i;
    }

    // Update ret_rg->rg_end
    ret_rg->rg_end = addr + pgnum * PAGING_PAGESZ;

    /* Tracking for later page replacement activities (if needed)
     * Enqueue new usage page */
    enlist_pgn_node(&caller->mm->fifo_pgn, pgn + pgit);

    return 0;
  }
```

Comment:

The function begins by initializing a counter **pgit** to 0 and computing the page number **pgn** corresponding to the given address **addr**. It then initializes the **rg_start** and **rg_end** members of the **ret_rg** structure to **addr**, since the first page is always mapped to that address. The function then enters a loop that maps each page in the range to a physical memory frame. Within the loop, it first obtains a pointer to the page table entry for the current page in the caller process's page table, using the page number **pgn** + counter **pgit**. It then calls the **pte_set_fpn** function, passing in the page table entry, a value of 1 indicating that the page is present, the physical frame number from the frames array, and some additional flags. The function then advances the pointer **fpit** to the next frame in the frames array. After the loop completes, the function enters another loop that enqueues each page in the range to the process's **fifo_pgn** page usage queue. Finally, the function updates the **rg_end** member of the **ret_rg** structure to the end address of the mapped page range, which is computed by adding **pgnum** x **PAGING_PAGESZ** to the starting address.

3.2.8 alloc_pages_range

This function is responsible for allocating a range of pages for a given process.

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

  for (pgit = 0; pgit < req_pgnum; pgit++)
  {
    // TODO: add usable frame to frm_lst
    if (MEMPHY_get_freefp(caller->mram, &fpn) == 0)
    {
      struct framephy_struct *newFrame = malloc(sizeof(struct
          framephy_struct));
      newFrame->fpn = fpn;
      newFrame->fp_next = (*frm_lst);
      (*frm_lst) = newFrame;
    }
    else
    { // ERROR CODE of obtaining somes but not enough frames
      // Doing the page replacement
      int vicpgn, vicfpn;
      find_victim_page(caller->mm, &vicpgn);
      vicfpn = PAGING_FPN(caller->mm->pgd[vicpgn]);
      // Find free frame in SWAP
      int swpfpn;
      MEMPHY_get_freefp(caller->active_mswp, &swpfpn);
      // Swap data from frame to swap
      __swap_cp_page(caller->mram, vicfpn, caller->active_mswp, swpfpn);
      // Update pgd info
      pte_set_swap(&caller->mm->pgd[vicpgn], 0, swpfpn);

      // use the frame taken from other page
      struct framephy_struct *newFrame = malloc(sizeof(struct
          framephy_struct));
      newFrame->fpn = vicfpn;
      newFrame->fp_next = (*frm_lst);
      (*frm_lst) = newFrame;
    }
  }
  return 0;
}
```

Comment:

It uses a loop to get several free frames based on how many new pages are created. It calls the function **MEMPHY_get_freefp** from the mram memory object to get a free frame from RAM. If there is a free frame in RAM, it creates a new **framephy_struct** object to hold the frame number and inserts it at the head of the linked list of frames allocated so far. If there are no free frames in RAM, it will do the page replacement mechanism, finding the victim page that will be put to **MEMSWAP**, using the frame that holds the victim page value as a free frame. If it succeeds, it creates a new **framephy_struct** object to hold the frame number and inserts it at the head of the linked list of frames allocated so far, marking the frame as owned by the process's memory map **caller→mm**.

3.3 Output results

We take input **os_0_mlq_paging** as an example:

Input:

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

Result:

```
============================================================
Time slot  0
ld_routine
   Loaded a process at input/proc/p0s, PID: 1 PRIO: 0
   CPU 0: Dispatched process 1
============================================================
Time slot  1
============================================================
Time slot  2
   Loaded a process at input/proc/p1s, PID: 2 PRIO: 15
   CPU 1: Dispatched process 2
============================================================
Time slot  3
   Loaded a process at input/proc/, PID: 3 PRIO: 0
============================================================
Time slot  4
   Loaded a process at input/proc/, PID: 4 PRIO: 0
============================================================
Time slot  5
write region=1 offset=20 value=100
--------------------PAGE TABLE CONTENT--------------------
```

```
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
------------------------------------------------------------
============================================================
Time slot  6
   CPU 0: Put process 1 to run queue
   CPU 0: Dispatched process 3
============================================================
Time slot  7
   CPU 0: Processed 3 has finished
   CPU 0: Dispatched process 4
============================================================
Time slot  8
   CPU 0: Processed 4 has finished
   CPU 0: Dispatched process 1
read region=1 offset=20 value=100
---------------------PAGE TABLE CONTENT---------------------
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
------------------------------------------------------------
   CPU 1: Put process 2 to run queue
   CPU 1: Dispatched process 2
============================================================
Time slot  9
============================================================
Time slot 10
============================================================
Time slot 11
============================================================
Time slot 12
   CPU 0: Processed 1 has finished
   CPU 0 stopped
   CPU 1: Processed 2 has finished
   CPU 1 stopped
```

As we can see, the output follow the flow we have described above. Let use **MY_CHECK** variable as a debug mechanism to understand clearly about the output:

Listing 1: Result with debugger

```
Time quantum: 6, number of CPUS: 2, number of process: 4
Size of MEM: 1048576
Size of MEMSWAP 0: 16777216
Size of MEMSWAP 1: 0
Size of MEMSWAP 2: 0
Size of MEMSWAP 3: 0
Testing:
input/proc/p0s 0   0
Testing:
input/proc/p1s 2   15
Testing:
input/proc/  0   0
Testing:
input/proc/  0   0
============================================================
Time slot  0
ld_routine
   Loaded a process at input/proc/p0s, PID: 1 PRIO: 0
   CPU 1: Dispatched process 1
============================================================
Time slot  1
   Loaded a process at input/proc/p1s, PID: 2 PRIO: 15
   CPU 0: Dispatched process 2
============================================================
Time slot  2
============================================================
Time slot  3
   Loaded a process at input/proc/, PID: 3 PRIO: 0
============================================================
Time slot  4
   Loaded a process at input/proc/, PID: 4 PRIO: 0
write region=1 offset=20 value=100
---------------------PAGE TABLE CONTENT---------------------
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
------------------------------------------------------------
-----------------------RAM CONTENT-------------------------
0x00000014: 00000064
============================================================
Time slot  5
```

```
------------------------------------------------------------
============================================================
Time slot  6
   CPU 1: Put process 1 to run queue
   CPU 1: Dispatched process 3
============================================================
Time slot  7
   CPU 1: Processed 3 has finished
   CPU 1: Dispatched process 4
   CPU 0: Put process 2 to run queue
   CPU 0: Dispatched process 1
read region=1 offset=20 value=100
---------------------PAGE TABLE CONTENT---------------------
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
------------------------------------------------------------
------------------------RAM CONTENT-------------------------
0x00000014: 00000064
   CPU 1: Processed 4 has finished
   CPU 1: Dispatched process 2
------------------------------------------------------------
============================================================
Time slot  8
writing error: not allocated or out of region range
============================================================
Time slot  9
reading error: not allocated or out of region range
============================================================
Time slot 10
writing error: not allocated or out of region range
============================================================
Time slot 11
reading error: not allocated or out of region range
   CPU 0: Processed 1 has finished
   CPU 0 stopped
============================================================
Time slot 12
   CPU 1: Processed 2 has finished
   CPU 1 stopped
```

Combining with the list of instructions **p0s**:

```
1 10
calc
alloc 300 0
alloc 300 4
free 0
alloc 100 1
write 100 1 20
read 1 20 20
write 102 2 20
read 2 20 20
write 103 3 20
read 3 20 20
calc
free 4
calc
```

Now, we are about to explain instructions related to memory such as **alloc**, **free**, **read**, **write**. Since we have 4 processes; however, we just have 1 among them including those instructions mentioned above, which is **p0s**

- **alloc 300 0:** Allocate the memory having the size of 300 and save this region to the **symrgtbl** at the index 0. We also map the page 0, 1 to the frame number 1, 0 respectively.
- **alloc 300 4:** similarly, we do the same thing but now we save this region to **symrgtbl[4]** and map the page 2, 3 to the frame number 3, 2 respectively. To this point, we have allocate memory for total 4 pages, each page have the size of 256, so totally we allocate 1024 unit of memory.
- **free 0:** Free the memory region at symrgtbl[0], so the region starting from 0 to 300 now becomes a free region.
- Due to the fact that we have just freed the region from 0 to 300, this will allocate the 100-size region from 0 to 100 and save it to **symrgtbl[1]**.
- **write 100 1 20:** Write the value 100 to the address of region 1 with offset 20, as we can see from the ouput.
- **read 1 20 20:** Read the value of memory at the address of region 1, offset 20. Because of the previous command, we gain the value 100.
- **write 102 2 20:** Because we haven't allocated memory for region 2 yet, so we fail to execute this command.
- **read 2 20 20:** With the same reason, we fail to execute this command.
- **write 103 3 20:** Because we haven't allocated memory for region 2 yet, so we fail

to execute this command.

- **read 3 20 20:**With the same reason, we fail to execute this command.
- **free 4:** free the memory region at **symrgtbl**[4], so the region starting from 512 to 812 currently becomes a free region.

4 Synchronization

4.1 Questions and Answers

Question 5

What will happen if the synchronization is not handled in your simple OS? Illustrate the problem of your simple OS by example if you have any.

Answer:

1. The need for synchronization:
   - Synchronization refers to the coordination and control of multiple threads or processes accessing shared resources to ensure their proper and coordinated execution.
   - The primary purpose of synchronization is to manage access to shared resources among concurrent threads or processes, preventing several critical issues like race conditions, deadlocks, and inconsistencies that can arise when multiple entities attempt to access shared data simultaneously.
2. Issues when the synchronization is not handled:
   - **Race Conditions**: A race condition manifests when multiple threads or processes access a shared resource in an unexpected sequence, leading to unpredictable behaviour. Race conditions are complex to detect as they rely on timing, and they can result in **data inconsistency** or **data corruption**, system crashes, and security vulnerabilities.

     For instance, consider two threads, **T1** and **T2**, both accessing a shared variable, **x**. If **T1** reads **x**, and before it can write a new value, **T2** also reads and writes to **x**, the final result depends on that of **T1**. If **T1**'s write operation happens after **T2**'s write, the value **T1** intended to set might get overwritten by **T2**'s write operation. This unexpected sequence of operations can cause **x** to end up with an unexpected value, deviating from the anticipated outcome due to the concurrent and non-deterministic nature of thread execution.
   - **Deadlocks**: Deadlocks emerge when multiple processes hold resources and simultaneously wait for other processes to release resources, creating a circular

dependency where none can proceed. Detecting and resolving deadlocks can be intricate as they may occur intermittently.

For instance, consider two processes, **A** and **B**, each holding a resource required by the other, such as **A** holds resource **X** and waits for resource **Y**. **B** holds resource **Y** and waits for resource **X**. In this case, neither process can proceed because they are waiting for a resource that the other process possesses. This circular dependency effectively halts both processes' progress, creating a standstill or deadlock situation.

- **Starvation**: Starvation occurs when a thread or process is prevented from accessing a shared resource indefinitely, even though the resource is available. This can happen when a high-priority thread monopolizes a shared resource or when a low-priority thread is constantly preempted by higher-priority threads. Starvation doesn't involve a deadlock scenario but a rather unfair resource distribution, hindering efficient system operation.

3. What will happen if the synchronization is not handled in our simple OS?

   The resources that need to be protected are:

   - **mlq_ready_queue**: Multiple processors or CPUs will try to access this queue to get the next process to execute, which can lead to issues if they accidentally access one queue and dequeue at a time. That is why we use **pthread_mutex_lock**, implemented in **scheduler.c** to ensure that only one processor can access and modify the queue at a time.

     Without a lock mechanism like **pthread_mutex_lock** in place to handle concurrent access, several issues could arise: **race conditions**.

     – **When two or more CPUs dequeue a process at the same time**, they might end up removing the same process from the queue. As a result, the process is removed twice or accessed when it shouldn't be, causing data loss or corruption. Besides, when dequeue, we take the process at index 0 and shift all the remaining processes to the left. Therefore, multiple CPU dequeues simultaneously dequeue can lead to an unpredictable outcome. Additionally, multiple CPUs simultaneously shift elements, they might interfere with each other's operations, leading to incorrect shifting, lost processes, or corrupt data structures.

     – **Two or more CPUs enqueue a process at the same time** can result in race conditions. CPUs might contend for the same location within the queue to add their processes. This can lead to conflicts where the processes might overwrite each other leading to incorrect placement within the queue and causing loss of process and inconsistent updates.

     – **Concurrent modifications during enqueueing or dequeuing oper-**

ations can result in invalid pointers within the queue structure, occurring when multiple CPUs are accessing and modifying pointers simultaneously. This inconsistency might disrupt the logical flow of the queue, causing elements to be out of sequence, and making the queue inefficient for scheduling processes based on their priorities or arrival times.

Overall, without synchronization, the lack of coordination between processors accessing the queue can result in **data corruption**, **inconsistency**, and **unpredictable behaviour**, compromising the **correctness and reliability of the scheduler** and the **entire operating system behaviour**.

To illustrate, we create a simple test case and compare the difference between before and after adding a mutex lock by adjusting **MTX** variable:

Input

```
6 2 2
1048576 16777216 0 0 0
1 p1s 0
2 p1s 1
```

Output

Listing 2: Before adding mutex lock

```
============================================================
Time slot 0
ld_routine
============================================================
Time slot 1
    Loaded a process at input/proc/p1s, PID: 1 PRIO: 0
============================================================
Time slot 2
    CPU 0: Dispatched process 1
    CPU 1: Dispatched process 1
    Loaded a process at input/proc/p1s, PID: 2 PRIO: 0
============================================================
Time slot 3
============================================================
Time slot 4
============================================================
Time slot 5
============================================================
Time slot 6
============================================================
Time slot 7
    CPU 0: Processed 1 has finished
```

```
   CPU 0: Dispatched process 2
============================================================
Time slot 8
   Segmentation fault
```

Listing 3: After adding mutex lock

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

```
CPU 1 stopped
```

As we can see, without mutex lock, one process will be assigned to both of the two CPUs for execution, allowing the process to be processed simultaneously in two separate locations with identical instructions. At **time slot 8**, **CPU1** failed to reinsert **process 1** into the system, but the execution of **process 1** was already completed. Consequently, **CPU1** encounters an error when attempting to add a non-existent (a freed) process to the multi-level queue.

- **free_fp_list**: Multiple processes may try to access this list to find fee frames either on MEMRAM or MEMSWP (secondary storage) to allocate new memory regions or perform page replacement. That is the reason why we implement **pthread_mutex_t access_lock** in the structure **memphy_struct** and used by **MEMPHY_get_freefp** and **MEMPHY_put_freefp**.

  Without synchronization mechanisms like the mutex locks implemented, concurrent access to shared resources from multiple threads can lead to various issues.

  – **When multiple CPUs allocate frames simultaneously without synchronization**, they might retrieve the same frame numbers from the free frame list. Consequently, multiple processes might end up using the same physical frame, leading to data conflicts or potential data overwrite.

  – **One thread could be removing a frame from the free frame list (MEMPHY_get_freefp) while another thread is adding a frame (MEMPHY_put_freefp)**. This scenario, without synchronization, might lead to inconsistencies, like the removal of a frame that's concurrently being added or adding a frame that's concurrently being removed. Such actions could corrupt the list structure or lead to missing or duplicated frames in the list.

  Aslo for the **memory storage access**: Because all operations on the memory utilize the shared same **RAM** and share **secondary storage**, there will have conflicts arise from asynchrony of **write**, **read**, or **alloc** commands. Therefore, we also implement mutex lock in **MEMPHY_read** and **MEMPHY_write**.

  The illustration is shown in section 4.3.

- **Virtual Memory**: Because a virtual memory only belongs to a process and a process in our assignment is single-threaded, there is no possible race condition.

4.2 Implementation

To add a mutex lock, we define a static variable **process_mtx**, initialize and call it before and after executing a command of process. Particularly, it is implemented as:

```
/*
* OTHER IMPLEMENTATIONS HERE
*/
static pthread_mutex_t process_mtx ;
/*
* OTHER IMPLEMENTATIONS HERE
*/
static void * cpu_routine ( void * args ) {
    /*
    * OTHER IMPLEMENTATIONS HERE
    */
    while (1) {
    /*
    * OTHER IMPLEMENTATIONS HERE
    */
    /* Run current process */
    pthread_mutex_lock (&process_mtx);
    run(proc);
    pthread_mutex_unlock(&process_mtx);
    /*
    * OTHER IMPLEMENTATIONS HERE
    */
    }
    /*
    * OTHER IMPLEMENTATIONS HERE
    */
}
/*
* OTHER IMPLEMENTATIONS HERE
*/
int main (int argc, char * argv []) {
    /*
    * OTHER IMPLEMENTATIONS HERE
    */
    /* Init process mutex */
    pthread_mutex_init (& process_mtx, NULL);
    /*
    * OTHER IMPLEMENTATIONS HERE
    */
}
```

Moreover, we also define a mutex lock in **memphy_struct** for locking when physical memory access free frame:

```
struct memphy_struct {
    /*
    * OTHER IMPLEMENTATIONS HERE
    */
    /* pthread for locking when get free frame*/
    pthread_mutex_t access_lock;
    /*
    * OTHER IMPLEMENTATIONS HERE
    */
}
```

Then, we implement it in **mm-memphy.c** file:

```
/*
* OTHER IMPLEMENTATIONS HERE
*/
int MEMPHY_read(struct memphy_struct *mp, int addr, BYTE *value)
{
    /*
    * OTHER IMPLEMENTATIONS HERE
    */
   pthread_mutex_lock(&mp->access_lock);
   if (mp->rdmflg)
      *value = mp->storage[addr];
   else /* Sequential access device */
      return MEMPHY_seq_read(mp, addr, value);
    pthread_mutex_unlock(&mp->access_lock);
    /*
    * OTHER IMPLEMENTATIONS HERE
    */
}
/*
* OTHER IMPLEMENTATIONS HERE
*/
int MEMPHY_write(struct memphy_struct *mp, int addr, BYTE data)
{
    /*
    * OTHER IMPLEMENTATIONS HERE
    */
   pthread_mutex_lock(&mp->access_lock);
    if (mp->rdmflg)
      mp->storage[addr] = data;
   else /* Sequential access device */
      return MEMPHY_seq_write(mp, addr, data);
   pthread_mutex_unlock(&mp->access_lock);
```

```
   /*
    * OTHER IMPLEMENTATIONS HERE
    */
}
/*
* OTHER IMPLEMENTATIONS HERE
*/
int MEMPHY_get_freefp(struct memphy_struct *mp, int *retfpn)
{
   pthread_mutex_lock(&mp->access_lock);
       /*
    * OTHER IMPLEMENTATIONS HERE
    */
   if (fp == NULL) {
    pthread_mutex_unlock(&mp->access_lock);
    return -1;
   }
   *retfpn = fp->fpn;
   mp->free_fp_list = fp->fp_next;
   /* MEMPHY is iteratively used up until its exhausted
    * No garbage collector acting then it not been released
    */
   // Attach this frame into the used_fp_list
   fp->fp_next = mp->used_fp_list;
   mp->used_fp_list = fp->fp_next;
   pthread_mutex_unlock(&mp->access_lock);
    /*
    * OTHER IMPLEMENTATIONS HERE
    */
}
/*
* OTHER IMPLEMENTATIONS HERE
*/
int MEMPHY_put_freefp(struct memphy_struct *mp, int fpn)
{
    pthread_mutex_lock(&mp->access_lock);
    struct framephy_struct *fp = mp->free_fp_list;
    struct framephy_struct *newnode = malloc(sizeof(struct framephy_struct));
    /* Create new node with value fpn */
    newnode->fpn = fpn;
    newnode->fp_next = fp;
    mp->free_fp_list = newnode;
    pthread_mutex_unlock(&mp->access_lock);
    /*
    * OTHER IMPLEMENTATIONS HERE
```

```
    */
}
/*
* OTHER IMPLEMENTATIONS HERE
*/
```

Combining with the mutex lock **queue_lock** that we already implemented in **sched.c** file, we archieve an effective synchronization mechanism.

4.3 Put it all together

After implementing all things above, we have a simple Operating System.
Firstly, led compare the difference between our result and given output of sched_1:

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

```
   CPU 0: Dispatched process 4
Time slot 14
Time slot 15
   CPU 0: Put process 4 to run queue
   CPU 0: Dispatched process 4
Time slot 16
Time slot 17
   CPU 0: Put process 4 to run queue
   CPU 0: Dispatched process 4
Time slot 18
Time slot 19
   CPU 0: Put process 4 to run queue
   CPU 0: Dispatched process 4
Time slot 20
   CPU 0: Processed 4 has finished
   CPU 0: Dispatched process 3
Time slot 21
Time slot 22
   CPU 0: Put process 3 to run queue
   CPU 0: Dispatched process 3
Time slot 23
Time slot 24
   CPU 0: Put process 3 to run queue
   CPU 0: Dispatched process 3
Time slot 25
Time slot 26
   CPU 0: Put process 3 to run queue
   CPU 0: Dispatched process 3
Time slot 27
Time slot 28
   CPU 0: Put process 3 to run queue
   CPU 0: Dispatched process 3
Time slot 29
Time slot 30
   CPU 0: Put process 3 to run queue
   CPU 0: Dispatched process 3
Time slot 31
Time slot 32
   CPU 0: Put process 3 to run queue
   CPU 0: Dispatched process 3
Time slot 33
Time slot 34
   CPU 0: Put process 3 to run queue
   CPU 0: Dispatched process 3
Time slot 35
```

```
   CPU 0: Processed 3 has finished
   CPU 0: Dispatched process 2
Time slot 36
Time slot 37
   CPU 0: Put process 2 to run queue
   CPU 0: Dispatched process 1
Time slot 38
Time slot 39
   CPU 0: Put process 1 to run queue
   CPU 0: Dispatched process 2
Time slot 40
Time slot 41
   CPU 0: Put process 2 to run queue
   CPU 0: Dispatched process 1
Time slot 42
Time slot 43
   CPU 0: Put process 1 to run queue
   CPU 0: Dispatched process 2
Time slot 44
Time slot 45
   CPU 0: Put process 2 to run queue
   CPU 0: Dispatched process 1
Time slot 46
Time slot 47
   CPU 0: Put process 1 to run queue
   CPU 0: Dispatched process 2
Time slot 48
Time slot 49
   CPU 0: Put process 2 to run queue
   CPU 0: Dispatched process 1
Time slot 50
Time slot 51
   CPU 0: Put process 1 to run queue
   CPU 0: Dispatched process 2
Time slot 52
Time slot 53
   CPU 0: Put process 2 to run queue
   CPU 0: Dispatched process 1
Time slot 54
Time slot 55
   CPU 0: Put process 1 to run queue
   CPU 0: Dispatched process 2
Time slot 56
Time slot 57
   CPU 0: Put process 2 to run queue
```

```
   CPU 0: Dispatched process 1
Time slot 58
   CPU 0: Processed 1 has finished
   CPU 0: Dispatched process 2
Time slot 59
Time slot 60
   CPU 0: Put process 2 to run queue
   CPU 0: Dispatched process 2
Time slot 61
   CPU 0: Processed 2 has finished
   CPU 0 stopped
```

There are several differences:

- Load wrong process file: Instead of loading four different processes as intended, the program only loads input s0. Additionally, it incorrectly retrieves the priority (PRIO) for the process.
- Same priority processes: With processes having the same priority, the given output shows that each process in the queue runs until completion before moving on to the next process in the same queue. However, in our code, the process will change when the time slice is used up. This approach ensures fairness among all processes with the same priority. This problem appear mainly because different dequeue and enqueue method.
- Priority queue slot: In our instruction, each priority queue will have a fixed slot (=MAX PRIO-prio), and it should only change to a lower priority queue if the slot is used up. However, in the given output, at time slot 37, the slot of level 0 priority still remains, but your program changes to a lower priority queue, which conflicts with 2.1 method.

Secondly, let compare the difference between our result and given output of **os_0_mlq_paging** to have a clearer view about memory management:

Listing 4: Our result of **os_0_mlq_paging**

```
================================================================
Time slot  0
ld_routine
   Loaded a process at input/proc/p0s, PID: 1 PRIO: 0
   CPU 0: Dispatched process 1
================================================================
Time slot  1
   Loaded a process at input/proc/p1s, PID: 2 PRIO: 15
   CPU 1: Dispatched process 2
================================================================
```

```
Time slot  2
================================================================
Time slot  3
   Loaded a process at input/proc/, PID: 3 PRIO: 0
================================================================
Time slot  4
   Loaded a process at input/proc/, PID: 4 PRIO: 0
================================================================
Time slot  5
write region=1 offset=20 value=100
---------------------PAGE TABLE CONTENT---------------------
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
------------------------------------------------------------
================================================================
Time slot  6
   CPU 0: Put process 1 to run queue
   CPU 0: Dispatched process 3
================================================================
Time slot  7
   CPU 0: Processed 3 has finished
   CPU 0: Dispatched process 4
================================================================
Time slot  8
   CPU 1: Put process 2 to run queue
   CPU 1: Dispatched process 1
read region=1 offset=20 value=100
---------------------PAGE TABLE CONTENT---------------------
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
------------------------------------------------------------
   CPU 0: Processed 4 has finished
   CPU 0: Dispatched process 2
================================================================
Time slot  9
================================================================
Time slot 10
================================================================
Time slot 11
```

```
============================================================
Time slot 12
   CPU 1: Processed 1 has finished
   CPU 1 stopped
   CPU 0: Processed 2 has finished
   CPU 0 stopped
```

Listing 5: Given output of **os_0_mlq_paging**

```
Time slot  0
ld_routine
   Loaded a process at input/proc/p0s, PID: 1 PRIO: 0
   CPU 0: Dispatched process 1
Time slot  1
Time slot  2
   Loaded a process at input/proc/p1s, PID: 2 PRIO: 15
Time slot  3
   Loaded a process at input/proc/p1s, PID: 3 PRIO: 0
Time slot  4
   CPU 1: Dispatched process 2
   Loaded a process at input/proc/p1s, PID: 4 PRIO: 0
Time slot  5
write region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Time slot  6
   CPU 0: Put process 1 to run queue
   CPU 0: Dispatched process 1
read region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Time slot  7
write region=2 offset=20 value=102
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Time slot  8
```

```
read region=2 offset=20 value=102
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Time slot  9
write region=3 offset=20 value=103
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Time slot 10
   CPU 1: Put process 2 to run queue
   CPU 1: Dispatched process 4
   CPU 0: Processed 1 has finished
   CPU 0: Dispatched process 3
Time slot 11
Time slot 12
Time slot 13
Time slot 14
Time slot 15
Time slot 16
   CPU 1: Put process 4 to run queue
   CPU 1: Dispatched process 2
   CPU 0: Put process 3 to run queue
   CPU 0: Dispatched process 3
Time slot 17
Time slot 18
Time slot 19
Time slot 20
   CPU 0: Processed 3 has finished
   CPU 0: Dispatched process 4
   CPU 1: Processed 2 has finished
   CPU 1 stopped
Time slot 21
Time slot 22
Time slot 23
Time slot 24
   CPU 0: Processed 4 has finished
   CPU 0 stopped
```

As we can see, the given output also allocate 1024 unit of memory for 4 pages, and map

the page 0, 1, 2, 3 to frame number 1, 0, 3, 2. The command **write 100 1 20** and **read 1 20 20** is executed successfully because region 1 is allocated. Oppositely, we don't see the prompts present for the other read commands, so we assume that 4 commands **write 102 2 20**, **read 2 20 20**, **write 103 3 20**, **read 3 20 20** fail to execute because region 2, 3 is not allocated. However, the given output load 4 processes, although the input file just have 2 processes, **p0s** and **p1s**, so there are faults in loader's procedure.

Now, we use our Operating System to run the other given inputs:

- **os_1_mlq_paging**
- **os_1_mlq_paging_small_1K**
- **os_1_mlq_paging_small_4K**
- **os_1_singleCPU_mlq**
- **os_1_singleCPU_mlq_paging**

Because the outputs are too long, so we will put them in this link for better presentation and access: **Link to outputs**

NOTE: All input can configure the memory size, except for **os_1_singleCPU_mlq**, so we have to adjust reasonably the variable **MM_FIXED_MEMSZ** file to run the above inputs. Particularly, you have to define **MM_FIXED_MEMSZ** in **os-mm.h** and **common.h** file to run input **os_1_singleCPU_mlq** and undefine it to run the other ones. The illustrate for this mechanism can be seen in the output of this file in the link above.

5 Conclusion

In conclusion, the exploration of the scheduler, synchronization, and memory allocation mechanisms within the simulated operating system has provided a comprehensive glimpse into the foundational pillars of OS functionality.

6 References

[1] Abraham Silberschatz, Operating System Concepts, 9th Edition

[2] Remzi H Arpaci-Dusseau - Andrea C Arpaci-Dusseau, Operating Systems - Three Easy Pieces

[3] GeeksforGeeks, Multilevel Queue (MLQ) CPU Scheduling

[4] GeeksforGeeks, Multilevel Feedback Queue Scheduling (MLFQ) CPU Scheduling

[5] GeeksforGeeks, Introduction of Process Synchronization

[6] GeeksforGeeks, Memory Mana ent in Operating System

[7] GeeksforGeeks, Virtual Memory in Operating System

[8] GeeksforGeeks, Memory Hierarchy Design and its Characteristics

[9] GeeksforGeeks, Paged Segmentation and Segmented Paging