

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



OPERATING SYSTEMS (CO2018)

Assignment

“Simple Operating System”

Students: Pham Duc Trung - 2153928
Nguyen Tien Thanh - 2053437
Le Duong Khanh Huy - 2153380

HO CHI MINH CITY, DECEMBER 2023



Contents

| | | |
|----------|--|-----------|
| 1 | Member list & Workload | 2 |
| 2 | Topic content | 3 |
| 3 | Scheduling | 3 |
| 3.1 | Multi level queue | 3 |
| 3.2 | Scheduling Implementation | 3 |
| 3.2.1 | Enqueue() and Dequeue() | 3 |
| 3.2.1.a | Enqueue() | 3 |
| 3.2.1.b | Dequeue() | 3 |
| 3.2.2 | Get_mlq_proc() | 4 |
| 3.2.3 | Implementation Result | 4 |
| 3.2.3.a | Starvation in MLQ | 6 |
| 3.3 | Question Answering | 8 |
| 4 | Memory management | 8 |
| 4.1 | Memory Management System using Paging Mechanism | 8 |
| 4.2 | Least Recently Used (LRU) Page Replacement Algorithm | 10 |
| 4.2.1 | Definition | 10 |
| 4.2.2 | Example | 11 |
| 4.3 | Result | 11 |
| 4.3.1 | Input | 11 |
| 4.3.2 | Output | 12 |
| 4.4 | Question Answering | 17 |
| 5 | Synchronization | 20 |
| 5.1 | Implementation | 20 |
| 5.2 | Question Answering | 20 |
| 6 | Conclusion | 22 |



1 Member list & Workload

| <i>No.</i> | Fullname | Student ID | Percentage of work |
|------------|--------------------|-------------------|---------------------------|
| 1 | Pham Duc Trung | 2153928 | 33% |
| 2 | Nguyen Tien Thanh | 2053437 | 33% |
| 3 | Le Duong Khanh Huy | 2153380 | 33% |

2 Topic content

- **Goal:** The objective of this assignment is the simulation of major components in a simple operating system, for example, scheduler, synchronization, related operations of physical memory and virtual memory.
- **Content:** In detail, student will practice with three major modules: scheduler, synchronization, mechanism of memory allocation from virtual-to-physical memory.
- **Result:** After this assignment, student can understand partly the principle of a simple OS. They can understand and draw the role of OS key modules.

3 Scheduling

3.1 Multi level queue

The Ready queue is organized into distinct queues based on specific criteria, such as the characteristics and response times of foreground (interactive) and background processes. Each algorithm supports a different process, but in a general system, some processes require scheduling using a priority algorithm.

The number of ready queue algorithms between queues and within queues is declared in the input. For the assignment's system, the system contains MAX PRIO priority levels. Although the real system, i.e. Linux kernel, may group these level into subset, we keep the design where each priority is held by one ready queue for simplicity. We simplify the add_queue and put_proc as putting the proc to appropriated ready queue by priority matching.

3.2 Scheduling Implementation

3.2.1 Enqueue() and Dequeue()

3.2.1.a Enqueue()

- If the queue is already full ($\text{size} > \text{MAX_SIZE}$) or the queue is NULL, then throw an error (queue full/max capacity).
- If the queue has no elements yet ($\text{size} == 0$), then simply insert at the first position in the array.
- Otherwise, traverse the array using an insertion mechanism (meaning if a position is found where the process has a higher priority than the current process, then stop).
- Next, shift the subsequent elements back one index and then insert the process into that position. Thus, the processes are also sorted by priority. (The process with the highest priority is at the beginning).

3.2.1.b Dequeue()

- Since the enqueue mechanism has already sorted the processes by priority, here, to remove the highest priority process from the queue, we just need to take the first element of the array.
- Next, we shift the elements behind the first element forward by one index and decrease the size variable by one unit.”

3.2.2 Get_mlq_proc()

With the function `get_mlq_proc()`, we proceed to traverse through the queues in the array `mlq_ready_queue[MAX_PRIO]`. If we find for the first time a non-empty queue (one that has processes), then we carry out the execution of the processes in this queue according to the Round Robin mechanism with a time slice as configured in the testcase.

Additionally, once a ready_queue is selected, we must create a static global variable to track the amount of time that queue has already used the CPU. If this time exceeds the max slot = `MAX_PRIO - prio`, then we must immediately yield the CPU to another queue for execution.

3.2.3 Implementation Result

To illustrate the code execution results, here we consider the following input:

```

1 2 3
0 p1s 1
1 p2s 2
2 p3s 0

```

Figure 1: Input scheduling MLQ

We have:

- **Time slice:** 1
- **Num of CPUs:** 2
- **Num of processes:** 3
 1. Process 1 (P1): Burst time = 10
 2. Process 2 (P2): Burst time = 12
 3. Process 3 (P3): Burst time = 11
- **Arrival time and live priority:** Shown on picture
We can see that the queues currently have the following processes:
- **mlq_ready_queue[1]:** P1 - max slot: $\text{MAX_PRIO} - \text{prio} = 140 - 1 = 139$
- **mlq_ready_queue[2]:** P2 - max slot: $\text{MAX_PRIO} - \text{prio} = 140 - 2 = 138$
- **mlq_ready_queue[0]:** P3 - max slot: $\text{MAX_PRIO} - \text{prio} = 140 - 0 = 140$

Gantt Chart:

| TimeSlot | T0 | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 | T9 | T10 | T11 |
|----------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| CPU0 | | | P2 | | | | | P1 | | | | P2 |
| CPU1 | | | P1 | | | | | P3 | | | | |
| TimeSlot | T12 | T13 | T14 | T15 | T16 | T17 | T18 | T19 | T20 | T21 | | |
| CPU0 | | | | | | P2 | | | | | | |
| CPU1 | | P3 | | | | | | | | | | |

Figure 2: Gantt Chart for MLQ

Output from the program:



```
Time slot 0
ld_routine
  Loaded a process at input/proc/pls, PID: 1 PRI0: 1
Time slot 1
  CPU 1: Dispatched process 1
  Loaded a process at input/proc/p2s, PID: 2 PRI0: 2
Time slot 2
  CPU 0: Dispatched process 2
  CPU 1: Put process 1 to run queue
  CPU 1: Dispatched process 1
  Loaded a process at input/proc/p3s, PID: 3 PRI0: 0
Time slot 3
  CPU 1: Put process 1 to run queue
  CPU 1: Dispatched process 3
  CPU 0: Put process 2 to run queue
  CPU 0: Dispatched process 1
Time slot 4
  CPU 0: Put process 1 to run queue
  CPU 0: Dispatched process 1
  CPU 1: Put process 3 to run queue
  CPU 1: Dispatched process 3
Time slot 5
  CPU 0: Put process 1 to run queue
  CPU 0: Dispatched process 1
  CPU 1: Put process 3 to run queue
  CPU 1: Dispatched process 3
Time slot 6
  CPU 0: Put process 1 to run queue
  CPU 0: Dispatched process 1
  CPU 1: Put process 3 to run queue
  CPU 1: Dispatched process 3
Time slot 7
  CPU 0: Put process 1 to run queue
  CPU 0: Dispatched process 1
  CPU 1: Put process 3 to run queue
  CPU 1: Dispatched process 3
Time slot 8
  CPU 0: Put process 1 to run queue
  CPU 0: Dispatched process 1
  CPU 1: Put process 3 to run queue
  CPU 1: Dispatched process 3
Time slot 9
  CPU 1: Dispatched process 3
  CPU 1: Put process 3 to run queue
  CPU 1: Dispatched process 3
  CPU 0: Put process 1 to run queue
  CPU 0: Dispatched process 1
Time slot 10
  CPU 0: Put process 1 to run queue
  CPU 0: Dispatched process 1
  CPU 1: Put process 3 to run queue
  CPU 1: Dispatched process 3
Time slot 11
  CPU 0: Processed 1 has finished
  CPU 0: Dispatched process 2
  CPU 1: Put process 3 to run queue
  CPU 1: Dispatched process 3
Time slot 12
  CPU 0: Put process 2 to run queue
  CPU 0: Dispatched process 2
  CPU 1: Put process 3 to run queue
  CPU 1: Dispatched process 3
Time slot 13
  CPU 1: Put process 3 to run queue
  CPU 1: Dispatched process 3
  CPU 0: Put process 2 to run queue
  CPU 0: Dispatched process 2
Time slot 14
  CPU 0: Put process 2 to run queue
  CPU 0: Dispatched process 2
  CPU 1: Processed 3 has finished
  CPU 1 stopped
Time slot 15
  CPU 0: Put process 2 to run queue
  CPU 0: Dispatched process 2
Time slot 16
  CPU 0: Put process 2 to run queue
  CPU 0: Dispatched process 2
Time slot 17
  CPU 0: Put process 2 to run queue
  CPU 0: Dispatched process 2
```

```
Time slot 18
CPU 0: Put process 2 to run queue
CPU 0: Dispatched process 2
Time slot 19
CPU 0: Put process 2 to run queue
CPU 0: Dispatched process 2
Time slot 20
CPU 0: Put process 2 to run queue
CPU 0: Dispatched process 2
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
```

3.2.3.a Starvation in MLQ

Starvation can occur in the Multi-Level Queue (MLQ) scheduling algorithm implemented in the provided code. Starvation happens when a process or a group of processes is unable to gain the required resources (CPU time in this context) to make progress.

In the MLQ scheduling algorithm, starvation may occur if processes with lower priority levels (higher values of prio in this case) are continuously enqueued in the `mlq_ready_queue` and do not get a chance to execute because higher-priority processes keep arriving. This can be problematic, especially if the time quantum for lower-priority processes is set too high.

To mitigate starvation, you can consider implementing aging or dynamic priority adjustment. Aging involves gradually increasing the priority of processes that have been waiting in the queue for an extended period without getting a chance to execute. This ensures that lower-priority processes eventually get an opportunity to run.

Explanation of Starvation Avoidance in MLQ Scheduling

1. Multi-Level Queue (MLQ) Scheduling: The MLQ (Multilevel Queue) scheduling algorithm organizes processes into multiple priority levels. Each priority level has its own queue, denoted as `mlq_ready_queue`, and a certain number of time slices, referred to as `timeslot`, are assigned to processes within each queue.

2. Priority Levels and Time Quantum:

- The `MAX_PRIO` constant represents the maximum number of priority levels. Processes at higher priority levels have shorter time slices.
- The `timeslot` variable is crucial; it represents the time quantum or time slice allocated to a process at a specific priority level.

3. Time Quantum and Process Selection:

- In the `get_mlq_proc` function, the code iterates through each priority level.
- If a non-empty queue is found at a priority level, and the `timeslot` for that level is not exhausted, a process is dequeued from the queue, and its `timeslot` is decremented.

```
1  for (int i = 0; i < MAX_PRIO; i++)
2  {
3      if (!empty(&mlq_ready_queue[i]))
4      {
5          if (mlq_ready_queue[i].timeslot != 0)
6          {
7              proc = dequeue(&mlq_ready_queue[i]);
8              mlq_ready_queue[i].timeslot--;
9              break;
10         }
11     }
```

```
12 }  
13
```

- In the `get_mlq_proc` function, the code iterates through each priority level.
- If a non-empty queue is found at a priority level, and the timeslot for that level is not exhausted, a process is dequeued from the queue, and its timeslot is decremented.

4. Time Quantum Exhaustion and Preemption:

- If a process's timeslot reaches zero, it is pre-empted, ensuring fair CPU allocation across different priority levels.
- The code then resets the timeslot for each priority level, with higher-priority levels receiving shorter time slices.

```
1  for (int i = 0; i < MAX_PRIO; i++)  
2  {  
3      mlq_ready_queue[i].timeslot = MAX_PRIO - i;  
4  }
```

5. Second Attempt After Time Slot Reset:

- If no process is selected during the initial iteration (indicating time quantum exhaustion), the code resets the timeslot for each priority level and runs the algorithm again.
- This mechanism prevents higher-priority processes from continuously preempting lower-priority processes, ensuring that processes at lower priority levels eventually get a chance to execute.

```
1  for (int i = 0; i < MAX_PRIO; i++)  
2  {  
3      if (!empty(&mlq_ready_queue[i]))  
4      {  
5          if (mlq_ready_queue[i].timeslot != 0)  
6          {  
7              proc = dequeue(&mlq_ready_queue[i]);  
8              mlq_ready_queue[i].timeslot--;  
9              break;  
10         }  
11     }  
12 }
```

- The timeslot mechanism prevents starvation by ensuring that processes at all priority levels get a fair share of CPU time.
- Lower-priority processes receive longer time slices, allowing them to execute even when higher-priority processes are present.
- The reset of timeslot and the second attempt mechanism prevent higher-priority processes from monopolizing the CPU, promoting fairness and preventing starvation.

3.3 Question Answering

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

Answer: A priority queue is a data structure used to store elements or tasks in a way that allows efficient retrieval of the highest priority elements or tasks. The advantage of using a priority queue over other scheduling algorithms is that it allows tasks with higher priorities to be processed before those with lower priorities, which is very useful in many situations.

- Firstly, faster processing speed: Priority queues allow for quicker processing as they prioritize tasks with higher priorities first. This means that important tasks are completed more quickly, which is very crucial in applications that require time accuracy.”
- Secondly, efficient memory use: Priority queues use memory efficiently as they only store the highest priority tasks. This means that the size of the data structure is minimized, which is very important in resource-constrained environments.
- Thirdly, better resource utilization: Priority queues allow for better resource utilization as they can be used to allocate resources to high-priority tasks first. This can prevent bottlenecks and improve the overall system performance.
- Fourthly, easy to implement realize: Priority queues are relatively easy to implement and can be used in a variety of applications. This makes them a popular choice for scheduling algorithms.
- Finally, customizable to needs: Priority queues can be customized to suit specific applications. For example, different priority levels can be assigned to different types of tasks, allowing for more detailed control over the scheduling process.

Overall, the use of priority queues in scheduling algorithms can provide many benefits, including faster processing time, efficient memory usage, better resource optimization, ease of deployment, and customization.

4 Memory management

4.1 Memory Management System using Paging Mechanism

1. **Virtual memory mapping in each process:** The virtual memory space is organized in the form of memory mappings for each process PCB. From the process’s perspective, the virtual memory includes continuous vm areas. In reality, each vm area corresponds to regions like code, stack, or heap. Therefore, the process maintains in its PCB a pointer to these adjacent memory regions.

Some structures to note in this section include Memory Area, Memory Region, Memory Mapping, CPU Address. These structures, along with a few others that support Virtual Memory, are located in the module mm-vm.c.

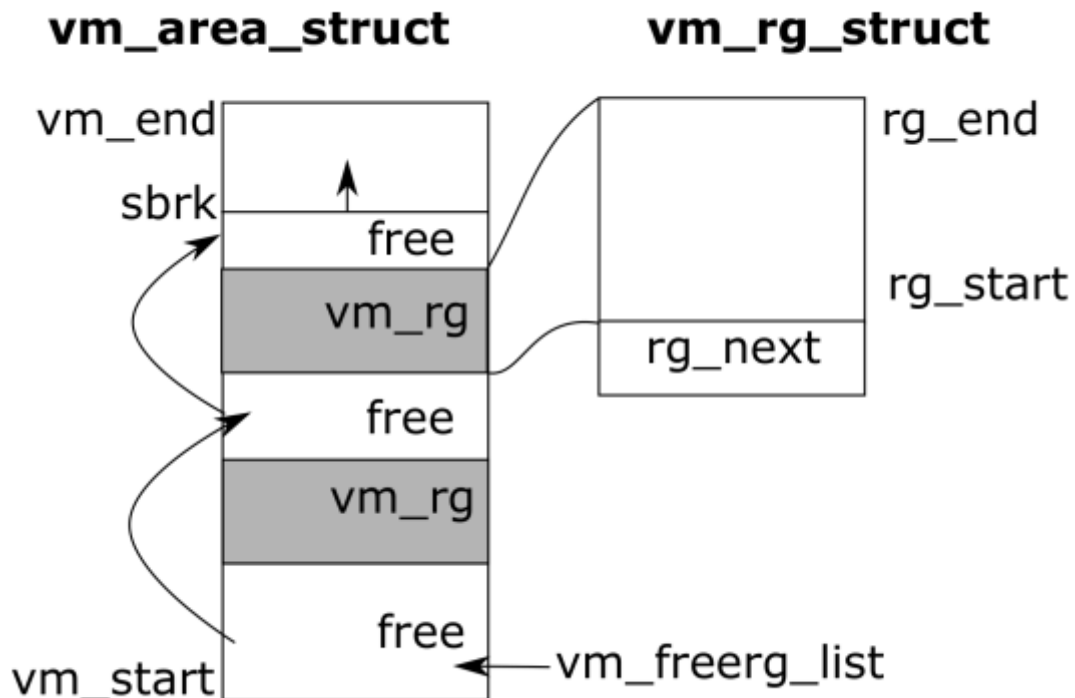


Figure 3: Structure of vm area and vm region

2. **Physical Memory System:** All processes own their separate memory mappings, but all these mappings point to the same single physical device. There are two types of devices used in this system: RAM and SWAP. The RAM device, part of the main memory system, can be accessed directly from the CPU's address bus, meaning it can be read/written with CPU commands. Meanwhile, SWAP is just an auxiliary memory device, and all operations of its stored data must be performed by moving them into the main memory. Since there's no direct access from the CPU, the system is often equipped with large SWAP at a low cost and may even have more than one device. Specifically, in this system, we support hardware installed with one RAM device and up to 4 SWAP devices.

Some structures to note in this section include Framephy Struct, Memphy Struct. These structures, along with a few others that support Physical Memory, are located within a module. `mm-memphy.c`

3. **Address Translation Mechanism:** In this version, the team developed a single-level paging system utilizing one RAM and one SWAP device. The team focuses only on using the first segment, which is the only segment of the vm area (with `vmaid = 0`). The specific translation mechanism is as follows:

Paging Table: Allows a process to determine the physical frame that each virtual page maps to. It contains a 32-bit value for each virtual page. For each entry, the page number (`pgn`) may correspond to a frame in MEMRAM or MEMSWP, or it may be null. The function of each data bit of the Page Table Entry is described as shown below. In the

chosen configuration, there are a total of 16,000 PTEs, each consuming 64 KB of storage capacity.

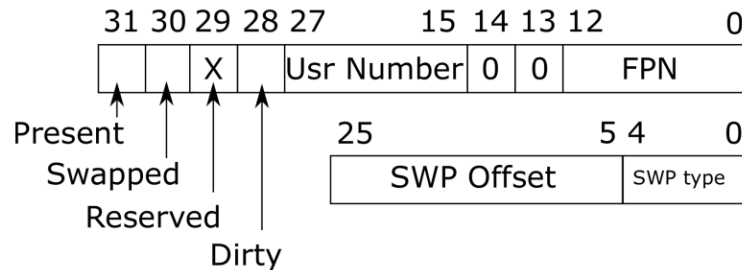
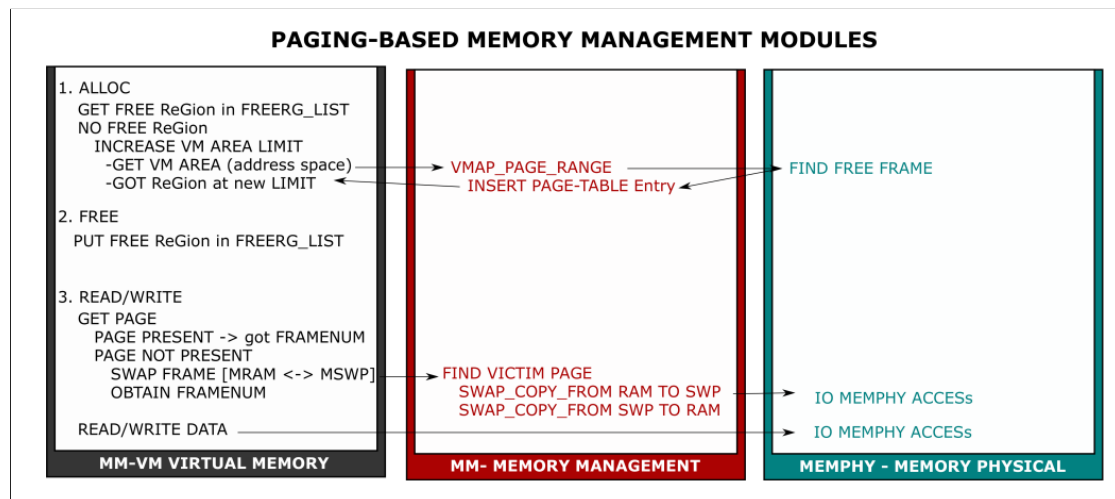


Figure 4: Paging Table Entry Format

Memory Swapping: It can be observed that a memory region may not fully utilize its allocated capacity, meaning that there are areas that are not mapped to MEMRAM (RAM). The swapping process involves exchanging the contents of physical frames between MEMRAM and MEMSWAP. In certain specific contexts, this swapping operation helps free up frames in RAM to accommodate the reading and storing of new data.

Basic Executions in the System: The memory management system based on the Paging mechanism developed by the team includes executions such as ALLOC, FREE, READ, WRITE. Details about these executions and related structures are located in the module mm.c, and they collaborate to manage memory as illustrated below.



4.2 Least Recently Used (LRU) Page Replacement Algorithm

4.2.1 Definition

This algorithm is basically dependent on the number of frames used. Then each frame takes up the certain page and tries to access it. When the frames are filled then the actual problem starts. The fixed number of frames is filled up with the help of first frames present. This concept is fulfilled with the help of Demand Paging

After filling up of the frames, the next page in the waiting queue tries to enter the frame. If the frame is present then, no problem is occurred. Because of the page which is to be searched is already present in the allocated frames.

If the page to be searched is found among the frames then, this process is known as Page Hit. And if the page to be searched is not found among the frames then, this process is known as Page Fault. When Page Fault occurs this problem arises, then the Least Recently Used (LRU) Page Replacement Algorithm comes into picture.

The Least Recently Used (LRU) Page Replacement Algorithms works on a certain principle: *"Replace the page with the page which is less dimension of time recently used page in the past."*

4.2.2 Example

Let say the page reference string 7 0 1 2 0 3 0 4 2 3 0 3 2 . Initially we have 4 page slots empty.

- Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots → **4 Page faults**.
- 0 is already there so → **0 Page fault**.
- when 3 came it will take the place of 7 because it is least recently used → **1 Page fault**
- 0 is already in memory so → **0 Page fault**.
- 4 will takes place of 1 → **1 Page fault**.
- Now for the further page reference string → **0 Page fault** because they are already available in the memory..

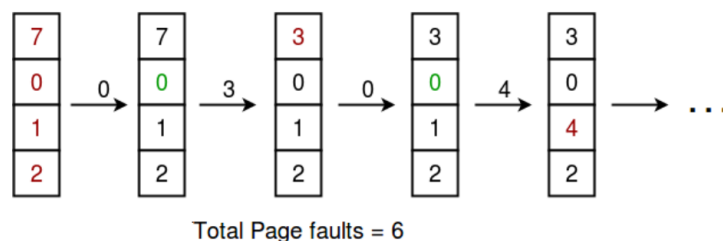


Figure 5: Given memory capacity (as number of pages it can hold) and a string representing pages to be referred, write a function to find number of page faults.

4.3 Result

4.3.1 Input

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

The input named `os_0_m1q_paging`. There are some important keys of this input:

- Total available memory: 1048576 bytes (1 MB)
- Frame size: 16777216 bytes (16 MB)
- There are no shared regions (0) and no regions with special memory allocation requirements (0).
- Process 0 (p0s): Priority (PRIO) is 0.
- Process 1 (p1s): Priority (PRIO) is 15.

4.3.2 Output

```
Time slot 0
ld_routine
    Loaded a process at input/proc/p0s, PID: 1 PRIO: 0
    CPU 1: Dispatched process 1
Time slot 1
Process 1 ALLOC CALL | SIZE = 300

>>>>Alloc Case>>>>No free region. #RGID: 0
[Page mapping] PID #1: Frame:0 PTE:80000000 PGN:0
[Page mapping] PID #1: Frame:1 PTE:80000001 PGN:1
>>>>Done>>>> #RGID: 0
Region id 0 : start = 0, end = 300
=====
Process 1 Free Region list
Start = 300, end = 512
=====
VMA id 1 : start = 0, end = 512, sbrk = 512
----- RAM mapping status -----
Number of mapped frames:      2
Number of remaining frames:  4094
-----
LRU LIST
FPN:
[0] -> [1]
=====
```

- Process 1 is then loaded from the file "input/proc/p0s" with PID 1 and priority 0
- The process makes an ALLOC call for a region of size 300. The system doesn't find a free region in the existing ones, so 2 new regions (page 0 and page 1) are allocated because the size of each page is 256 B. The allocated region is from 0 to 300, and the free region (from 300 to 512) list is updated accordingly.
- Page Mapping:
 - PID #1 (Process 1) is mapped to Frame 0 with Page Table Entry (PTE) 80000000 for Page Number (PGN) 0.
 - PID #1 (Process 1) is mapped to Frame 1 with PTE 80000001 for PGN 1.
- VMA (Virtual Memory Area) ID 1:
 - Start address: 0, End address: 512.
 - sbrk (break address): 512.

- Frame Page Numbers (FPN):
 $0 \rightarrow [1]$.

```
Time slot 2
Process 1 ALLOC CALL | SIZE = 300

>>>>Alloc Case>>>>No free region. #RGID: 4
[Page mapping] PID #1: Frame:2 PTE:80000002 PGN:2
[Page mapping] PID #1: Frame:3 PTE:80000003 PGN:3
>>>>Done>>>> #RGID: 4
Region id 0 : start = 0, end = 300
Region id 4 : start = 512, end = 812
=====
Loaded a process at input/proc/p1s, PID: 2 PRIO: 15
Process 1 Free Region list
Start = 300, end = 512
Start = 812, end = 1024
=====
VMA id 1 : start = 0, end = 1024, sbrk = 1024
----- RAM mapping status -----
Number of mapped frames: 4
Number of remaining frames: 4092
-----
LRU LIST
FPN:
[0] -> [1] -> [2] -> [3]
=====
```

- The process 1 continues to make an ALLOC call for a region of size 300. The system doesn't find a free region in the existing ones, so 2 new regions (page 2 and page 3) are allocated. The allocated region is from 0 to 300 and 512 to 812, and the free region(not used) (300 to 512 and 812 to 1024) lists are updated accordingly.
- A new process is loaded from "input/proc/p1s" with PID 2 and priority 15.
- Page Mapping:
 - PID #1 (Process 1) is mapped to Frame 2 with Page Table Entry (PTE) 80000002 for Page Number (PGN) 2.
 - PID #1 (Process 1) is mapped to Frame 3 with PTE 80000003 for PGN 3.
- VMA(Virtual Memory Area) ID 1:
 - Start address: 0, End address: 1024.
 - sbrk (break address): 1024.
 - Frame Page Numbers (FPN):
 $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$.

```
Time slot 3
CPU 0: Dispatched process 2
Process 1 FREE CALL | Region id 0 after free: [0,0]
Region id 4 : start = 512, end = 812
Process 1 Free Region list
Start = 300, end = 512
Start = 812, end = 1024
Start = 0, end = 300
LRU LIST
FPN:
[0] -> [1] -> [2] -> [3]
=====
```

```
Time slot 4
Process 1 ALLOC CALL | SIZE = 100

Alloc Case: GET FREE RG in FREERG LIST. #RGID: 1
current_pgn = 1
print head: 80000000
Looking for: 80000001
Found: 80000001

>>>>DONE>>>> #RGID: 1
FOUND A FREE region to alloc.
=====
```

- Process 1 has freed Region id 0, and after freeing, the region is now empty ([0,0]). Only region id 4 is used so the free region list is updated from 0 to 512 and from 812 to 1024.

```
>>>>DONE>>>> #RGID: 1
FOUND A FREE region to alloc.
=====
Region id 1 : start = 300, end = 400
Region id 4 : start = 512, end = 812
VMA id 1 : start = 0, end = 1024, sbrk = 1024
Process 1 Free Region list
Start = 400, end = 512
Start = 812, end = 1024
Start = 0, end = 300
----- RAM mapping status -----
Number of mapped frames: 4
Number of remaining frames: 4092
-----
LRU LIST
FPN:
[0] -> [2] -> [3] -> [1]
=====
```

- Process 1 continues to make an allocation call for a region of size 100. The system is looking for a free region in the free region list, and it found one with Region ID 1 (page 1).
- The used regions are re-updated from 0 to 400 and 512 to 812. The free regions are updated accordingly from 400 to 512 and 812 to 1024.
- The LRU list also changes from [0] → [1] → [2] → [3]. to [0] → [2] → [3] → [1] because the last used or updated page (page 1) will be moved to the tail of the list.



```
Time slot 5
Process 1 write region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000000
00000004: 80000001
00000008: 80000002
00000012: 80000003

Print content of RAM (only print nonzero value)
-----
Address 0x00000140: 100
-----
LRU LIST
FPN:
[0] -> [2] -> [3] -> [1]
=====
```

```
Time slot 6
CPU 1: Put process 1 to ready queue
CPU 1: Dispatched process 1
+++++
Process 1 read region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000000
00000004: 80000001
00000008: 80000002
00000012: 80000003

Print content of RAM (only print nonzero value)
-----
Address 0x00000140: 100
-----
LRU LIST
FPN:
[0] -> [2] -> [3] -> [1]
=====
```

- The process 1 attempts to executes "Write" and "Read" value 100 at offset 20 in Region ID 1(allocated previously in time slot 4).
- These operation succeeded, so the content of RAM at address 0x00000140 is 100.
- The virtual page at 4 addresses 0x00000000, 0x00000004, 0x00000008, 0x00000012 are mapped to to a physical frame with the identifier(PTE Value) 80000000, 80000001, 80000002, 80000003


```
Time slot 7
Process 1 write region=2 offset=20 value=102
Process 1 write error: Region not found (freed or uninitialized)
print_pgtbl: 0 - 1024
00000000: 80000000
00000004: 80000001
00000008: 80000002
00000012: 80000003

Print content of RAM (only print nonzero value)
-----
Address 0x00000140: 100
-----
LRU LIST
FPN:
[0] -> [2] -> [3] -> [1]
=====
```

- Process 1 writes to Region 2 at Offset 20 with Value 102 but occur an error "Region not found". This is because Process 1 try to access a region of memory that does not currently exist in physical memory (RAM). So, this operation can not be processed.
- The virtual page at 4 addresses 0x00000000, 0x00000004, 0x00000008, 0x00000012 are mapped to to a physical frame with the identifier(PTE Value) 80000000, 80000001, 80000002, 80000003. The content of RAM at address 0x00000140 still 100. Frame Page Numbers remains unchanged (FPN): [0] → [2] → [3] → [1].

```
Time slot 8
Process 1 read error: Region not found (freed or uninitialized)
Process 1 error when read region=2 offset=20
print_pgtbl: 0 - 1024
00000000: 80000000
00000004: 80000001
00000008: 80000002
00000012: 80000003

Print content of RAM (only print nonzero value)
-----
Address 0x00000140: 100
-----
LRU LIST
FPN:
[0] -> [2] -> [3] -> [1]
=====
```

- Process 1 read data to Region 2 at Offset 20 but occur an error "Region not found". This situation is quite similar to "Time slot 7" when Process 1 try to access a region of memory that does not currently exist in physical memory (RAM).
- The virtual page remain unchanged at 4 addresses 0x00000000, 0x00000004, 0x00000008, 0x00000012 are mapped to to a physical frame with the identifier(PTE Value) 80000000, 80000001, 80000002, 80000003.
- The content of RAM at address 0x00000140 still 100. Frame Page Numbers remains unchanged (FPN): [0] → [2] → [3] → [1].

```
Time slot 9
Process 1 write region=3 offset=20 value=103
Process 1 write error: Region not found (freed or uninitialized)
print_pgtbl: 0 - 1024
00000000: 80000000
00000004: 80000001
00000008: 80000002
00000012: 80000003

Print content of RAM (only print nonzero value)
-----
Address 0x00000140: 100
CPU 0: Put process 2 to ready queue
CPU 0: Dispatched process 2
-----
LRU LIST
FPN:
[0] -> [2] -> [3] -> [1]
=====

Time slot 10
CPU 1: Processed 1 has finished
CPU 1 stopped
Time slot 11
Time slot 12
Time slot 13
CPU 0: Processed 2 has finished
CPU 0 stopped
ubuntu@ubuntu:~/Desktop$
```

- Process 1 writes to Region 2 at Offset 20 with Value 103 but occur an error "Region not found". This situation is similar to time slot 7 and 8 because Process 1 try to access a region of memory that does not currently exist in physical memory (RAM).
- The virtual page table at 4 addresses 0x00000000, 0x00000004, 0x00000008, 0x00000012 are mapped to to a physical frame with the identifier(PTE Value) 80000000, 80000001, 80000002, 80000003.
- The content of RAM at address 0x00000140 is still 100.
- Frame Page Numbers remains unchanged (FPN): [0] → [2] → [3] → [1].
- CPU 0 processes and finishes executing Process 1.
- CPU 1 processes and finishes executing Process 2.
- Finally, Both CPUs stop.

4.4 Question Answering

Question 1: 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: In operating systems, programs and data are stored in memory. However, not all memory can be used for each program. Instead, memory is divided into different segments to serve different purposes.

For example, an operating system might divide memory into the following segments:

- Code segment: Stores the machine code of the program.
- Data segment: Stores data used by the program.
- Heap segment: Stores dynamically generated data while the program is running.
- Stack segment: Stores local variables and function call values within the program.

With such segmentation of memory, programs can be managed better and the risks of memory overflow or memory conflicts are minimized.

Moreover, dividing memory into segments allows the operating system to provide different access rights for each segment. For example, a program is not allowed to overwrite its own machine code in the Code segment, but it can write to the Data segment.

A real-life example is when you are using a web browser to access different websites, each website is loaded and displayed in a separate tab. Each tab is created by generating a separate process in the operating system, and each process will have separate memory segments for Code segment, Data segment, Heap segment, and Stack segment. Dividing memory into different segments allows tabs to operate independently of each other and minimizes the impact of memory conflict errors between tabs.

Another advantage of using multiple memory segments is that it allows for more efficient memory management. Different segments can be allocated and freed independently, enabling more flexible memory management.

For example, in a program that requires both stack and heap, the stack can be allocated in one segment and the heap in another. This allows for separate management of these two memory areas, potentially improving overall performance and reducing the risk of memory-related errors. Additionally, multiple memory segments also offer improvements in security. By isolating different segments, unauthorized access or modification to critical system components can be prevented.

For instance, in an operating system, the kernel and user processes can be placed in separate memory segments. This allows for better control over access to system resources and can prevent malware from accessing or modifying important system components.

Overall, the use of multiple memory segments can offer several benefits, including more efficient memory management, improved performance, reduced risk of memory-related errors, and enhanced security. However, it also requires more complex memory management algorithms and may increase the costs associated with memory allocation and deallocation.

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

Answer: In a paged memory management system, we can divide the logical address into several levels to manage memory more effectively. Typically, paged memory management systems are designed with 2 or 3 levels of addressing, but if we divide the address into more than 2 levels, it can have some impact on system performance.

When dividing the address into more than 2 levels, the number of bits used for each level decreases. This can lead to internal fragmentation in memory, meaning that some parts of the frame will not be used. For example, if we divide the address into 4 levels, with each level using only 4 bits, then each frame would only use 16 bytes instead of 4KB as in 2 levels, leading to memory wastage.

Another issue that can arise with using multiple levels of addressing is a decrease in system processing speed due to the need to access multiple page tables. When accessing a page table, the system must perform calculations to determine the actual address of the frame corresponding to the logical address. Accessing multiple page tables can slow down the memory retrieval process. However, dividing the address into multiple levels also has some benefits. When using multiple

levels of addressing, the system can manage memory better and use memory resources more efficiently. For example, in a paged memory management system with 3 levels of addressing, the logical address is divided into a page number, page offset, and segment number. The segment number helps us determine the physical address of the page table corresponding to that segment, speeding up the memory retrieval process.

However, dividing the address into multiple levels also has some benefits. When using multiple levels of addressing, the system can manage memory better and use memory resources more efficiently. For example, in a paged memory management system with 3 levels of addressing, the logical address is divided into a page number, page offset, and segment number. The segment number helps us determine the physical address of the page table corresponding to that segment, speeding up the memory retrieval process.

In conclusion, dividing the address into multiple levels has its own advantages and limitations. The needs and objectives of the implementation should be considered to choose the appropriate number of levels for dividing the memory space.

Question 3: What is the advantage and disadvantage of segmentation with paging?

Answer: Segmentation combined with paging is a method of memory management in operating systems. This method combines the advantages of both segmentation and paging. However, it also has its own strengths and weaknesses.

Advantages:

- Flexible memory management: Segmentation allows for the management of larger memory blocks, while paging enables the management of smaller blocks. Combining both leads to more flexible memory management, helping to reduce memory wastage and increase data retrieval speed.
- Increased accuracy: Segmentation allows for dividing memory into different segments depending on the program's purpose, which helps increase the accuracy in memory allocation and reduces memory wastage.
- Enhanced security: When combining segmentation and paging, each segment and page are assigned different access rights. This enhances the system's security by preventing unauthorized access to critical memory areas.

Disadvantages:

- Complex and resource-intensive: Combining segmentation and paging increases the complexity of memory management, requiring more resources for operation. It can decrease system performance if not implemented correctly.
- Fragmentation: The combination of segmentation and paging can lead to memory fragmentation. When there are no longer sufficient contiguous memory blocks to allocate for a segment or a page, the system will have to search for free memory blocks in different locations, leading to memory fragmentation and decreased system performance.
For example, when a program requires a segment larger than the page size, this leads to a situation where the segment is spread over multiple pages. When this segment is no longer in use, the pages are freed but are not continuous enough to form a new segment, causing fragmentation in both segmentation and paging.

The combination of segmentation and paging helps to overcome the drawbacks of each technique individually. However, this combination also inherits the disadvantages of both methods. It creates complexity in memory management, causes delays, and leads to memory fragmentation. Nonetheless, in many cases, the combination of segmentation and paging is an effective solution for memory management.

5 Synchronization

5.1 Implementation

Since there are multiple threads running concurrently and using some shared memory, we need to use a synchronization mechanism to protect the data and ensure the correct result.

In this simple operating system, the threads mentioned above are the loader and the CPU(s). All of these threads can access the ready queues, so we use a mutex queue lock and put it in the functions `get_mlq_proc`, `put_mlq_proc`, `add_mlq_proc` (in `MLQ_SCHED` mode), and `get_proc`, `put_proc`, `add_proc` (in normal mode). The CPUs also use the same physical memory device, hence we put a mutex `memphy_lock` to protect it in the functions `MEMPHY_read`, `MEMPHY_write`, `MEMPHY_get_freep`, `MEMPHY_put_freep`.

5.2 Question Answering

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

Answer: Synchronization is the coordination of the execution of multiple processes in a multi-process system to ensure that they access shared resources in a controlled and predictable way. It aims to address issues related to race conditions and other synchronization problems in a concurrent system.

In operating systems, synchronization is very important as it ensures that multiple processes can access shared resources without interfering with each other. If synchronization is not handled properly in an operating system, it can lead to various issues such as race conditions, deadlocks, and starvation.

1. Race conditions: In this scenario, multiple processes access and update a shared resource without any synchronization mechanism, leading to the final value of the resource being undefined or incorrect. As a result, the operation outcome of the system becomes unpredictable and can cause issues.
2. Deadlocks: In this case, multiple processes request access to resources that are locked and are waiting for each other to release the resources. This can lead to a situation of congestion, where all processes are locked and unable to perform any operation.
3. Starvation: In this situation, a process is prevented from accessing the crucial resources it needs to perform its operations. This can lead to a state of resource starvation, where a process is unable to continue its operations because the important resources are being held by other processes.

Here is my practical example show the importance of synchronized tool and how issues mentioned previously occur:

Here is our input before we run the program:

```
10 2 15
2 p1s 1
3 p2s 2
4 p3s 3
5 p4s 139
7 p6s 8
8 p7s 9
9 p8s 10
10 p9s 11
11 p10s 12
12 p11s 13
13 p12s 14
14 p13s 99
15 p14s 50
16 p15s 70
17 p16s 0
```

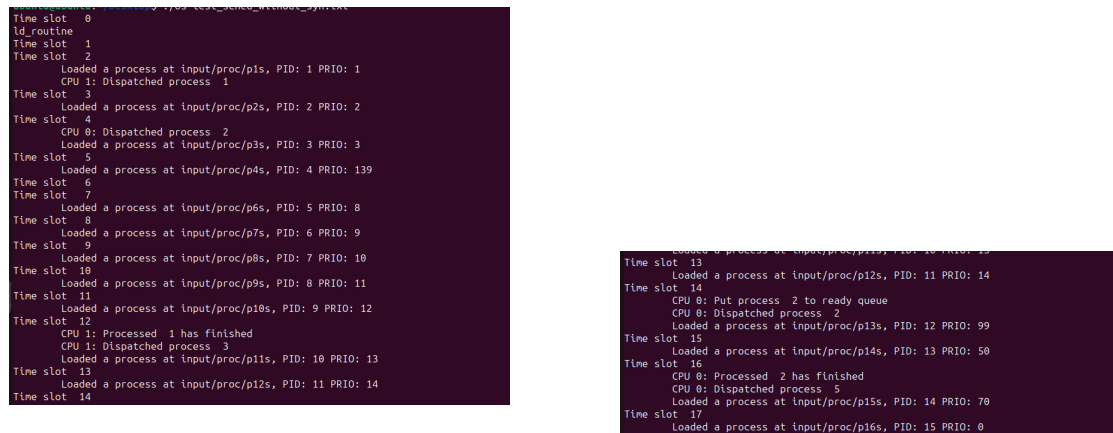
Figure 6: Input

And here is the output without synchronized tools (mutex-lock and mutex-unlock):

```
Time slot 0
ld_routine
Time slot 1
Time slot 2
    Loaded a process at input/proc/p1s, PID: 1 PRI0: 1
    CPU 0: Dispatched process 1
    CPU 1: Dispatched process 1
Time slot 3
    Loaded a process at input/proc/p2s, PID: 2 PRI0: 2
Time slot 4
    Loaded a process at input/proc/p3s, PID: 3 PRI0: 3
Time slot 5
    Loaded a process at input/proc/p4s, PID: 4 PRI0: 139
Time slot 6
Time slot 7
    CPU 1: Processed 1 has finished
    Loaded a process at input/proc/p6s, PID: 5 PRI0: 8
    CPU 1: Dispatched process 2
Segmentation fault (core dumped)
```

Figure 7: Output without synchronized tools

As we can see, without synchronized tool, the program encounters a problem: 1 process is dispatched to 2 CPU, that means 2 CPU can access and modify concurrently the same process. Access to one process simultaneously results many negative outcome: Data Corruption, Inconsistent State, Race Conditions, Unpredictable Results. In our situation, our program stops due to Segmentation fault.



```
Time slot 0
ld_routine
Time slot 1
Time slot 2
  Loaded a process at input/proc/p1s, PID: 1 PRI0: 1
  CPU 1: Dispatched process 1
Time slot 3
  Loaded a process at input/proc/p2s, PID: 2 PRI0: 2
Time slot 4
  CPU 0: Dispatched process 2
  Loaded a process at input/proc/p3s, PID: 3 PRI0: 3
Time slot 5
  Loaded a process at input/proc/p4s, PID: 4 PRI0: 139
Time slot 6
Time slot 7
  Loaded a process at input/proc/p6s, PID: 5 PRI0: 8
Time slot 8
  Loaded a process at input/proc/p7s, PID: 6 PRI0: 9
Time slot 9
  Loaded a process at input/proc/p8s, PID: 7 PRI0: 10
Time slot 10
  Loaded a process at input/proc/p9s, PID: 8 PRI0: 11
Time slot 11
  Loaded a process at input/proc/p10s, PID: 9 PRI0: 12
Time slot 12
  CPU 1: Processed 1 has finished
  CPU 1: Dispatched process 3
  Loaded a process at input/proc/p11s, PID: 10 PRI0: 13
Time slot 13
  Loaded a process at input/proc/p12s, PID: 11 PRI0: 14
Time slot 14
  Loaded a process at input/proc/p13s, PID: 12 PRI0: 99
Time slot 15
  Loaded a process at input/proc/p14s, PID: 13 PRI0: 50
Time slot 16
  CPU 0: Processed 2 has finished
  CPU 0: Dispatched process 5
  Loaded a process at input/proc/p15s, PID: 14 PRI0: 70
Time slot 17
  Loaded a process at input/proc/p16s, PID: 15 PRI0: 0
```

Figure 8: Output with synchronized tool

By using synchronized tool, everything become better, each process is dispatched and executed sequentially.

6 Conclusion

In this modified version, my team tried to implement a simple Operating System with scheduling, memory management, synchronization features. Besides, the program is tested carefully with many testcases.

There are some additional parts included in this project to match the requirements:

- We re-implemented the "sched.c" module with some starvation avoidance mechanisms (aging).
- Also, We re-ran the whole program without synchronized tool(mutex-lock and mutex-unlock) and gave some evaluation on the results.
- The output layout was also changed to be concise and readable.