ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC BÁCH KHOA TP. HỒ CHÍ MINH

KHOA KHOA HỌC – KĨ THUẬT MÁY TÍNH

----ଓ📖ଌ----

Assignment- Simple Operating System

MÔN: HỆ ĐIỀU HÀNH

Họ & Tên	MSSV	Đóng góp
Nguyễn Võ Thiên Bảo	2310250	100%
Nguyễn Ngọc Trúc Quỳnh	2312912	100%
Trần Anh Tài	2252726	100%
Trần Nguyễn Phú Nghĩa	2312284	100%

GVHD: *Nguyễn Minh Tâm*

TP. HCM, ngày 26 April 2025

1. Scheduling

1.1. Mô tả thuật toán

Hệ điều hành mô phỏng trong bài tập này sử dụng thuật toán lập lịch Đa hàng đợi (Multi-Level Queue - MLQ). Thuật toán này được thiết kế để hoạt động trên hệ thống đa xử lý (multiple processors). Cơ chế hoạt động chính dựa trên việc phân loại các tiến trình (processes) vào các hàng đợi (ready queues) khác nhau dựa trên độ ưu tiên (priority) của chúng. Mỗi hàng đợi tương ứng với một mức ưu tiên cố định, với giá trị `prio` thấp hơn thể hiện độ ưu tiên cao hơn. Hệ thống có tổng cộng `MAX_PRIO` (ví dụ: 140) mức ưu tiên.

Khi một chương trình mới được nạp (load), bộ nạp (loader) sẽ tạo một tiến trình mới, cấp phát một Process Control Block (PCB), sao chép mã lệnh vào text segment của tiến trình, và đưa PCB vào hàng đợi sẵn sàng tương ứng với độ ưu tiên `prio` của nó.

Các CPU chọn tiến trình từ các hàng đợi sẵn sàng để thực thi theo chính sách MLQ. Chính sách này cấp cho mỗi hàng đợi ưu tiên `prio` một số lượng time slot nhất định, được tính bằng `slot = MAX_PRIO - prio`. Một hàng đợi chỉ được sử dụng CPU nếu nó còn `slot`. Khi một hàng đợi đã sử dụng hết `slot` của mình, hệ thống sẽ chuyển sang cấp phát CPU cho hàng đợi có độ ưu tiên thấp hơn tiếp theo, ngay cả khi hàng đợi ưu tiên cao hơn vẫn còn tiến trình đang chờ. Điều này đảm bảo các tiến trình có độ ưu tiên thấp hơn vẫn có cơ hội được thực thi, tránh tình trạng "starvation". Khi một tiến trình đã chạy hết `time_slice` được cấp, nó sẽ bị buộc đưa trở lại hàng đợi ưu tiên tương ứng của nó thông qua hàm `put_proc()`. CPU sau đó sẽ chọn một tiến trình khác từ hệ thống hàng đợi thông qua `get_proc()` để tiếp tục thực thi. Nếu trong một lượt duyệt qua tất cả các hàng đợi mà không tìm thấy tiến trình nào phù hợp (do hàng đợi rỗng hoặc hết slot), hệ thống sẽ reset lại số `slot` cho tất cả các hàng đợi và bắt đầu lại chu trình lựa chọn.

Hệ thống lập lịch được minh họa trong Hình 2 của tài liệu đề bài. Lưu ý rằng `run_queue` được đề cập trong sơ đồ là một thành phần lỗi thời và không còn được sử dụng trong logic MLQ hiện tại.

1.2. Trả lời câu hỏi

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

Trả lời: Thuật toán MLQ được triển khai trong bài tập này có nhiều ưu điểm so với các thuật toán lập lịch khác:

- **Hỗ trợ ưu tiên đa cấp hiệu quả:** MLQ cho phép phân chia tiến trình thành nhiều mức ưu tiên khác nhau, đảm bảo rằng các tiến trình quan trọng (có `prio` thấp) được ưu tiên thực thi trước. Điều này rất hữu ích cho các hệ thống đòi hỏi thời gian phản hồi nhanh.
- **Chống Starvation:** Khác với thuật toán Priority Scheduling thuần túy có thể khiến các tiến trình ưu tiên thấp không bao giờ được chạy, MLQ sử dụng cơ chế `slot`. Mỗi mức ưu tiên được cấp một số lượng `slot` nhất định (`MAX_PRIO - prio`), đảm bảo rằng ngay cả các tiến trình có độ ưu tiên thấp nhất (ví dụ `prio = 139` vẫn có 1 slot) cũng có cơ hội được cấp CPU.
- **Công bằng hơn Round-Robin (RR) trong nhiều trường hợp:** RR đối xử công bằng tuyệt đối với mọi tiến trình nhưng không phân biệt được tầm quan trọng. MLQ kết hợp cả yếu tố ưu tiên và tính công bằng tương đối thông qua `slot`, phù hợp hơn cho các hệ thống có sự khác biệt về độ quan trọng giữa các tiến trình.
- **Đơn giản hơn Shortest Job First (SJF):** SJF yêu cầu biết trước thời gian thực thi của tiến trình, điều này thường không khả thi trong thực tế. MLQ chỉ dựa vào độ ưu tiên `prio` được gán sẵn, làm cho việc triển khai đơn giản hơn.
- **Phù hợp với môi trường đa CPU:** Thiết kế MLQ cho phép nhiều CPU hoạt động đồng thời, mỗi CPU có thể lấy tiến trình từ hệ thống hàng đợi chung, giúp tối ưu hóa việc sử dụng tài nguyên phần cứng.
- **Thiết kế đơn giản:** Việc giữ mỗi mức ưu tiên trong một hàng đợi riêng biệt giúp đơn giản hóa thiết kế và triển khai so với các biến thể MLQ phức tạp hơn (ví dụ như có cơ chế feedback).

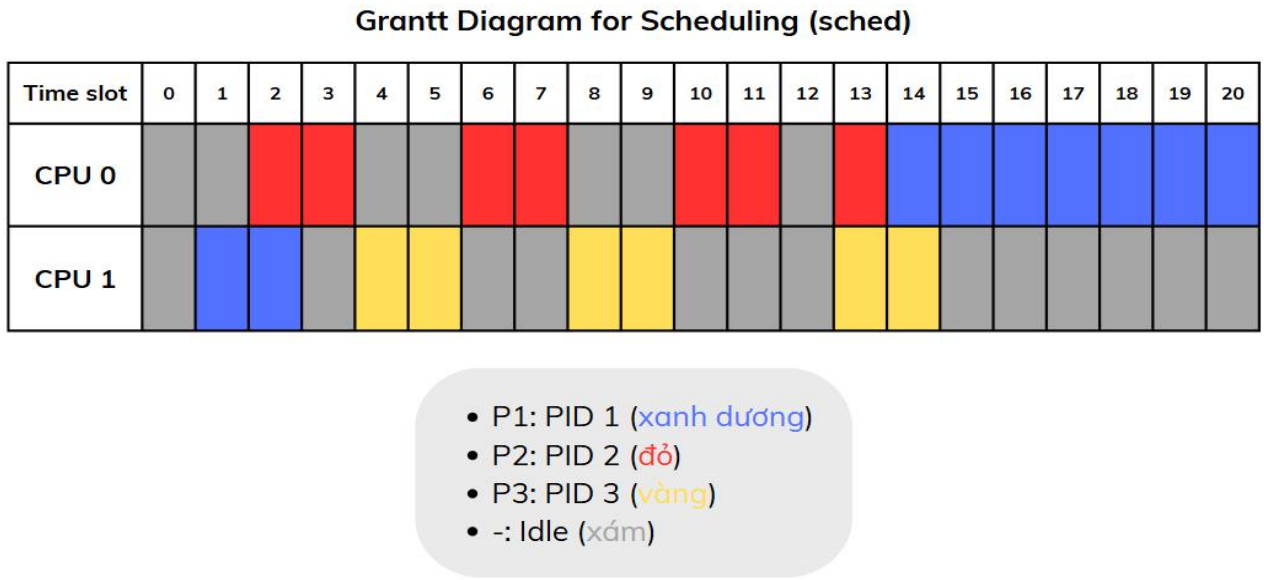
Tuy nhiên, MLQ cũng có nhược điểm:

- **Phức tạp hơn RR:** Việc quản lý nhiều hàng đợi (`MAX_PRIO` hàng đợi) phức tạp hơn so với chỉ một hàng đợi của RR.
- **Thiếu linh hoạt (không có Feedback):** Phiên bản MLQ này không có cơ chế feedback (như trong Linux thực tế) để điều chỉnh độ ưu tiên của tiến trình trong quá trình chạy. Độ ưu tiên của tiến trình là cố định sau khi được nạp (trừ khi bị ghi đè bởi giá trị trong file cấu hình).

1.3. Phân tích kết quả thực thi

1.3.1. Giản đồ Gantt

Để minh họa trực quan quá trình lập lịch, chúng ta có thể vẽ giản đồ Gantt dựa trên output sched. Giản đồ này cho thấy rõ ràng CPU chuyển đổi giữa các tiến trình như thế nào, bao gồm cả việc ngắt xen dựa trên độ ưu tiên:



Hình 1: Giản đồ Gantt cho sched

1.3.2. Phân tích chi tiết

Phân tích các output mới cho thấy scheduler MLQ hoạt động theo cơ chế **preemptive (ngắt xen)** dựa trên độ ưu tiên:

- **Phân tích sched_0**
  - PID 1 (prio 4) được nạp và chạy từ slot 1.
  - PID 2 (prio 0) được nạp tại slot 4.
  - Tại slot 5, PID 1 (đang chạy, prio 4) bị đưa ra khỏi CPU ("Put process 1 to run queue"). Ngay lập tức, **PID 2 (prio 0, ưu tiên cao hơn)** được cấp phát CPU ("Dispatched process 2"). Điều này thể hiện rõ tính **preemptive**.
  - PID 2 chạy theo kiểu Round-Robin (vì chỉ có mình nó ở mức ưu tiên cao nhất) cho đến khi hoàn thành tại slot 12.

- Sau khi PID 2 (ưu tiên cao) hoàn thành, PID 1 (ưu tiên thấp) mới được chạy lại và hoàn thành tại slot 23.

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

```
Time slot  10
Time slot  11
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot  12
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  1
```

```
Time slot  23
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

```
Time slot  16
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  17
Time slot  18
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  19
Time slot  20
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  21
Time slot  22
```

- **Phân tích sched_1**
  - PID 1 (prio 4) chạy từ slot 1.
  - PID 2 (prio 0) được nạp tại slot 4. Tại slot 5, PID 1 bị ngắt và **PID 2 (ưu tiên cao hơn)** được chạy.
  - PID 3 (prio 0) được nạp tại slot 6. Tại slot 7, PID 2 bị ngắt và **PID 3 (cùng ưu tiên 0)** được chạy.
  - PID 4 (prio 0) được nạp tại slot 7.
  - Từ slot 7 trở đi, các tiến trình có cùng độ ưu tiên cao nhất (PID 2, 3, 4 - prio 0) chạy **luân phiên theo kiểu Round-Robin**. Ví dụ: slot 7-8 là PID 3, slot 9-10 là PID 2, slot 11-12 là PID 4, slot 13-14 là PID 3,...
  - Các tiến trình prio 0 lần lượt hoàn thành (PID 2 tại slot 22, PID 3 tại slot 33, PID 4 tại slot 34).
  - Chỉ sau khi **tất cả** các tiến trình ưu tiên cao (prio 0) đã hoàn thành, tiến trình ưu tiên thấp **PID 1 (prio 4)** mới được cấp phát CPU trở lại tại slot 35 và chạy cho đến khi hoàn thành tại slot 45.

```
Time slot  22
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  4
Time slot  23
Time slot  24
        CPU 0: Put process  4 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  4
Time slot  27
Time slot  28
        CPU 0: Put process  4 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  4
Time slot  31
Time slot  32
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  3
Time slot  33
        CPU 0: Processed  3 has finished
        CPU 0: Dispatched process  4
```

```
Time slot   34
        CPU 0: Processed  4 has finished
        CPU 0: Dispatched process  1
Time slot   35
Time slot   36
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   37
Time slot   38
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   39
Time slot   40
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   41
Time slot   42
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   43
Time slot   44
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   45
        CPU 0: Processed  1 has finished
        CPU 0 stopped
Time slot   46
```

- **Phân tích sched**
    - Trong môi trường đa CPU (2 CPU), hành vi preemptive cũng được thể hiện. PID 1 (prio 1) đang chạy trên CPU 1 bị đưa ra tại slot 4 để nhường CPU cho PID 3 (prio 0) mới đến.
    - Các tiến trình prio 0 (PID 2, 3) chiếm dụng CPU 0 và CPU 1.
    - PID 1 (prio 1) chỉ được chạy lại trên CPU 0 tại slot 14 sau khi PID 2 (prio 0) đã hoàn thành, và tiếp tục chạy sau khi PID 3 (prio 0) cũng hoàn thành.

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

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

2. Memory Management

2.1. Mô tả cơ chế quản lý bộ nhớ

Hệ điều hành mô phỏng này triển khai một hệ thống quản lý bộ nhớ dựa trên sự kết hợp của phân đoạn (segmentation) và phân trang (paging). Mục tiêu là cung cấp cho mỗi tiến trình một không gian địa chỉ ảo (virtual address space) riêng biệt và ánh xạ nó vào bộ nhớ vật lý (physical memory) dùng chung.

2.1.1. Virtual Memory Mapping (Ánh xạ bộ nhớ ảo)

- **Không gian địa chỉ ảo:** Mỗi tiến trình có một không gian địa chỉ ảo được quản lý bởi cấu trúc `mm_struct`. Không gian này được chia thành các vùng nhớ (memory areas) liên tục, được biểu diễn bởi `vm_area_struct`. Mỗi `vm_area` có thể đóng vai trò như code segment, data segment, heap segment, hoặc stack segment. Danh sách các `vm_area` được liên kết với nhau qua con trỏ `vm_next` và được quản lý trong `mm_struct` thông qua con trỏ `mmap`.
- **Vùng nhớ (Memory Area -** `vm_area_struct`**):** Mỗi vùng nhớ được xác định bởi địa chỉ bắt đầu (`vm_start`) và kết thúc (`vm_end`). Bên trong một `vm_area`, không phải toàn bộ không gian đều được sử dụng ngay lập tức. Con trỏ `sbrk` đánh dấu giới hạn trên của vùng nhớ thực sự có thể sử dụng (ví dụ: cho heap).
- **Vùng cấp phát (Memory Region -** `vm_rg_struct`**):** Trong phần bộ nhớ có thể sử dụng (dưới `sbrk`), các khối nhớ đã được cấp phát được quản lý bởi cấu trúc `vm_rg_struct`. Các vùng nhớ chưa cấp phát (lỗ trống) trong `vm_area` được theo dõi qua danh sách liên kết `vm_freerg_list`. Các `vm_rg_struct` đại diện cho các "biến" hoặc các khối dữ liệu được cấp phát bởi tiến trình thông qua lệnh `ALLOC`. Để đơn giản, thay vì một bảng ký hiệu (symbol table) phức tạp, hệ thống sử dụng một mảng `symrgtbl` có kích thước cố định (`PAGING_MAX_SYMTBL_SZ`) trong `mm_struct` để lưu thông tin về các vùng đã cấp phát này.
- **Page Table Directory (PGD):** Trường `pgd` trong `mm_struct` là thành phần quan trọng nhất cho việc phân trang, nó trỏ đến cấu trúc bảng trang (page table) của tiến trình, thực hiện ánh xạ từ trang ảo (virtual page) sang khung vật lý (physical frame).

2.1.2. Physical Memory

- **Thiết bị:** Hệ thống quản lý hai loại bộ nhớ vật lý chính: RAM (bộ nhớ chính) và SWAP (bộ nhớ phụ, thường dùng để lưu trữ tạm các trang không hoạt động). Cả hai đều có thể được mô phỏng bởi cấu trúc `memphy_struct`. Hệ thống hỗ trợ một thiết bị RAM và tối đa 4 thiết bị SWAP.

- **Cấu trúc (`memphy_struct`):** Lưu trữ con trỏ đến vùng nhớ thực sự (`storage`), kích thước tối đa (`maxsz`), cờ cho biết truy cập ngẫu nhiên hay tuần tự (`rdmflg`), và con trỏ quản lý danh sách các khung trống (`free_fp_list`) và đã sử dụng (`used_fp_list`).
- **Khung (Frame):** Bộ nhớ vật lý được chia thành các khối có kích thước cố định gọi là khung (frame), ví dụ 256B hoặc 512B. Cấu trúc `framephy_struct` dùng để biểu diễn và quản lý các khung này (ví dụ: lưu số khung `fpn`).

2.1.3. Paging-based Address Translation (Dịch địa chỉ dựa trên Phân trang)

- **Địa chỉ CPU:** Địa chỉ ảo do CPU tạo ra (ví dụ: 22-bit) được chia thành hai phần: số hiệu trang (Page Number - p) và độ lệch trong trang (Page Offset - d). Ví dụ, với địa chỉ 22-bit và kích thước trang 256B (2^8), 8 bit cuối là offset, 14 bit đầu là page number.
- **Bảng trang (Page Table):** Mỗi tiến trình có bảng trang riêng. Bảng trang này chứa các mục từ bảng trang (Page Table Entry - PTE), mỗi PTE ánh xạ một trang ảo sang một khung vật lý hoặc chỉ ra rằng trang đó đang ở SWAP hoặc chưa được cấp phát. Với cấu hình 22-bit CPU và trang 256B, bảng trang có khoảng 2^14 = 16384 mục từ.
- **Định dạng PTE (Page Table Entry):** Mỗi PTE là một giá trị 32-bit chứa các thông tin:
    - **Present Bit (Bit 31):** Cho biết trang có đang ở trong RAM không.
    - **Swapped Bit (Bit 30):** Cho biết trang có đang ở SWAP không.
    - **Dirty Bit (Bit 28):** Cho biết trang đã bị sửa đổi chưa (dùng cho swapping out).
    - **Page Frame Number (FPN) (Bits 0-12):** Số hiệu khung vật lý nếu trang có trong RAM (Present=1).
    - **Swap Type (Bits 0-4) & Swap Offset (Bits 5-25):** Thông tin vị trí trên SWAP nếu trang bị swap ra ngoài (Swapped=1, Present=0).
    - Các bit còn lại: Reserved hoặc User-defined.
- **Quá trình dịch địa chỉ:** Khi CPU truy cập địa chỉ ảo, MMU (Memory Management Unit) sử dụng Page Number để tra cứu PTE trong bảng trang của tiến trình hiện hành.
    - Nếu Present=1, FPN được lấy ra, kết hợp với Page Offset để tạo thành địa chỉ vật lý.
    - Nếu Present=0 và Swapped=1, xảy ra Page Fault. Hệ điều hành phải tìm một khung trống trong RAM, swap trang cần thiết từ SWAP vào khung đó, cập nhật PTE (đặt Present=1, Swapped=0, ghi FPN mới), rồi mới thực hiện lại truy cập. Nếu không có khung trống, một trang khác trong RAM phải bị swap out (dựa trên thuật toán thay thế trang, ví dụ FIFO được đề cập với `fifo_pgn`) để giải phóng khung.

    - Nếu Present=0 và Swapped=0, đây là truy cập vào vùng nhớ ảo chưa được cấp phát, gây lỗi Segmentation Fault.
- **Hoạt động cơ bản (ALLOC, FREE, READ, WRITE):**
    - **ALLOC:** Tìm một vùng trống trong `vm_freerg_list` hoặc mở rộng vùng nhớ (`sbrk`). Nếu mở rộng vào vùng ảo chưa có trang vật lý, cần gọi syscall để yêu cầu MMU cấp phát các khung vật lý (từ `free_fp_list` của RAM) và cập nhật các PTE tương ứng trong bảng trang.
    - **FREE:** Chỉ đơn giản là đánh dấu vùng nhớ (`vm_rg_struct`) là trống và thêm nó vào `vm_freerg_list` trong không gian ảo. Không thu hồi khung vật lý ngay lập tức để tránh phân mảnh.
    - **READ/WRITE:** Yêu cầu trang ảo tương ứng phải có mặt trong RAM (Present=1). Nếu không, xảy ra page fault và cơ chế swapping được kích hoạt như mô tả ở trên.

2.2. Trả lời câu hỏi

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

Trả lời: Việc thiết kế nhiều vùng nhớ (memory areas/segments) như `vm_area_struct` trong không gian địa chỉ ảo của tiến trình mang lại các lợi ích sau:

1. **Phân tách hợp lý (Logical Separation):** Cho phép tổ chức bộ nhớ của tiến trình thành các phần riêng biệt với chức năng rõ ràng, ví dụ: code segment, data segment, heap, stack. Điều này giúp quản lý bộ nhớ dễ dàng hơn và theo dõi lỗi hiệu quả hơn.
2. **Bảo vệ bộ nhớ (Memory Protection):** Mỗi segment/area có thể được gán các quyền truy cập khác nhau (đọc, ghi, thực thi). Ví dụ, code segment có thể được đặt là chỉ đọc (read-only) để ngăn chặn việc vô tình hoặc cố ý ghi đè lên mã lệnh. Việc phân tách cũng giúp ngăn lỗi tràn bộ đệm (buffer overflow) từ vùng này ảnh hưởng sang vùng khác (ví dụ: heap overflow ghi đè stack).
3. **Quản lý bộ nhớ linh hoạt:** Cho phép các vùng nhớ khác nhau phát triển độc lập. Ví dụ, heap và stack có thể mở rộng mà không xung đột trực tiếp với nhau hoặc với code/data segment cố định. Con trỏ `sbrk` giúp quản lý việc mở rộng linh hoạt của một vùng (thường là heap).
4. **Hỗ trợ chia sẻ (Sharing):** Thiết kế này tạo nền tảng cho việc chia sẻ các segment giữa các tiến trình.

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

Trả lời: Khi chia địa chỉ thành nhiều hơn 2 cấp (ví dụ: 3 hoặc 4 cấp), chúng ta sẽ sử dụng hệ thống bảng trang đa cấp (multi-level page table) thay vì bảng trang một cấp như được mô tả chính trong bài tập này. Điều này có các ảnh hưởng sau:

- **Ưu điểm:**
    - **Tiết kiệm bộ nhớ vật lý:** Thay vì cấp phát toàn bộ một bảng trang lớn (có thể rất thưa thớt) cho mỗi tiến trình, bảng trang đa cấp chỉ cấp phát các bảng trang cấp thấp hơn khi các mục từ ở cấp cao hơn thực sự được sử dụng. Điều này đặc biệt hiệu quả với không gian địa chỉ ảo lớn và thưa thớt (sparse address space).
    - **Khả năng mở rộng:** Phù hợp hơn cho các kiến trúc có không gian địa chỉ lớn (như 64-bit), nơi bảng trang một cấp sẽ trở nên quá lớn để quản lý hiệu quả.
    - **Cấu trúc phân cấp:** Tạo ra cấu trúc quản lý bộ nhớ có tổ chức hơn.
- **Nhược điểm:**
    - **Tăng thời gian truy cập bộ nhớ:** Mỗi lần dịch địa chỉ ảo sang vật lý cần nhiều lượt truy cập bộ nhớ hơn (một lượt cho mỗi cấp bảng trang) so với bảng trang một cấp. Làm tăng độ trễ truy cập bộ nhớ tổng thể.
    - **Tăng độ phức tạp:** Hệ điều hành cần quản lý cấu trúc bảng trang phức tạp hơn.
    - **Ảnh hưởng của TLB Miss:** Khi xảy ra TLB (Translation Lookaside Buffer) miss, chi phí để nạp lại thông tin dịch địa chỉ sẽ cao hơn đáng kể vì cần phải duyệt qua nhiều cấp bảng trang trong bộ nhớ chính.

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

Trả lời: Việc kết hợp cả phân đoạn (segmentation) và phân trang (paging) - mặc dù bài tập này chủ yếu tập trung vào phân trang trong các `vm_area` (có thể coi là các segment logic) - mang lại sự kết hợp ưu nhược điểm của cả hai phương pháp:

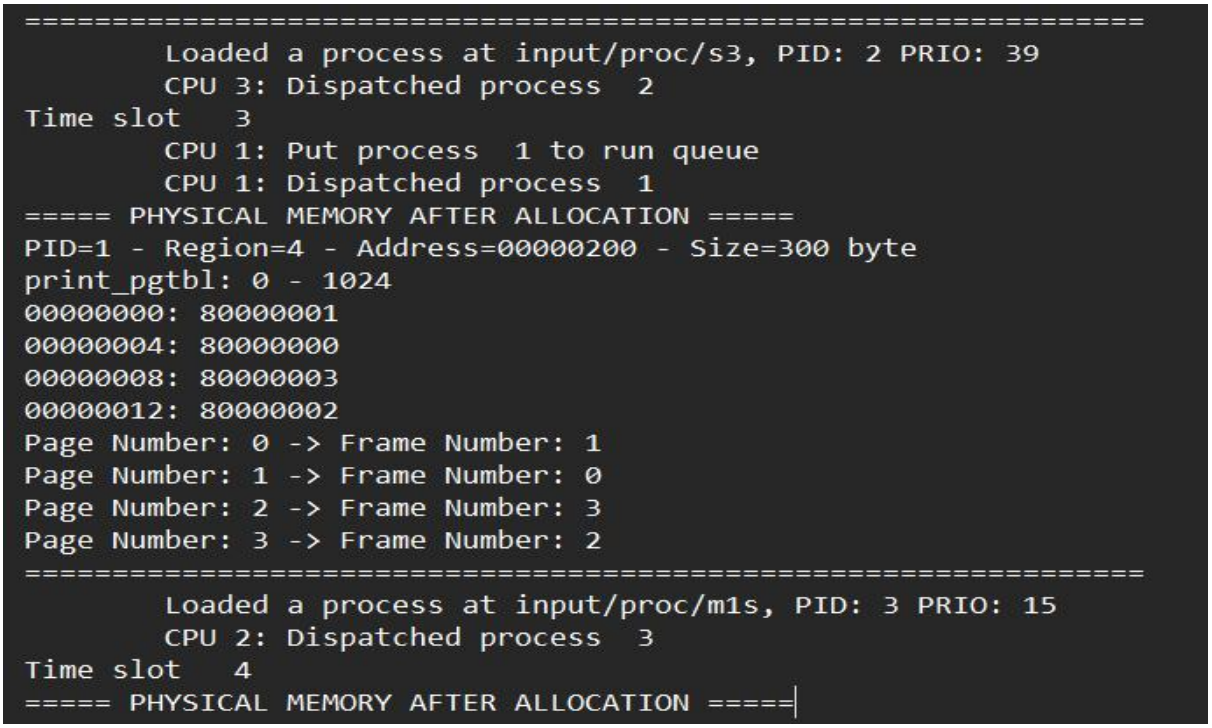
- **Ưu điểm:**
    - **Kết hợp lợi ích:** Tận dụng được khả năng phân chia logic và bảo vệ theo segment, đồng thời khắc phục được phân mảnh ngoại (external fragmentation) nhờ paging bên trong mỗi segment.
    - **Quản lý bộ nhớ linh hoạt:** Cho phép cấp phát bộ nhớ vật lý không liên tục cho các segment logic liên tục, dễ dàng quản lý các vùng nhớ có kích thước thay đổi (như heap, stack).

    - **Hỗ trợ chia sẻ và bảo vệ tốt:** Có thể chia sẻ các segment cụ thể (như code) giữa các tiến trình trong khi vẫn giữ các segment khác (như data, stack) riêng tư. Quyền truy cập có thể định nghĩa ở cấp segment.
    - **Hỗ trợ không gian địa chỉ logic:** Phù hợp với cách lập trình viên tư duy về chương trình (code, data, stack riêng biệt).
- **Nhược điểm:**
    - **Độ phức tạp cao:** Yêu cầu cả bảng đoạn (segment table) và bảng trang (page table), làm cho quá trình dịch địa chỉ phức tạp hơn và tốn nhiều thời gian hơn.
    - **Overhead bộ nhớ:** Cần thêm không gian để lưu trữ cả hai loại bảng.
    - **Yêu cầu phần cứng phức tạp:** MMU phải hỗ trợ cả hai cơ chế dịch
    - **Phân mảnh nội (Internal Fragmentation):** Vẫn tồn tại phân mảnh nội trong trang cuối cùng của mỗi segment được cấp phát.
    - **Khó tối ưu:** Quản lý nhiều segment nhỏ có thể tạo ra overhead

2.3. Phân tích kết quả thực thi

Cấp phát bộ nhớ (ALLOC):

- Khi một tiến trình gọi lệnh `ALLOC size reg`, ví dụ trong `os_1_mlq_paging` tại Time slot 2, PID 1 thực hiện `ALLOC 300 0`. Output cho thấy: `===== PHYSICAL MEMORY AFTER ALLOCATION =====` `PID=1 - Region=0 - Address=00000000 - Size=300 byte` `print_pgtbl: 0 - 512` `00000000: 80000001` (Page 0 valid, mapped to Frame 1) `00000004: 80000000` (Page 1 valid, mapped to Frame 0) `Page Number: 0 -> Frame Number: 1` `Page Number: 1 -> Frame Number: 0` Điều này cho thấy 300 byte đã được cấp phát tại địa chỉ ảo `0x00000000` (trong `Region=0`). Vì kích thước trang là 256B (mặc định), việc cấp phát này chiếm 2 trang ảo (Page 0 và Page 1). Bảng trang (`print_pgtbl`) được cập nhật để ánh xạ Page 0 tới Frame 1 và Page 1 tới Frame 0 trong bộ nhớ vật lý. Ký hiệu `80000001` và `80000000` là các PTE, bit cuối cùng (bit 31 - Present) là 1, cho thấy trang có trong RAM, các bit đầu (0-12) là số khung (FPN).
- Tương tự, tại Time slot 3, PID 1 cấp phát tiếp `ALLOC 300 4` tại địa chỉ ảo `0x00000200` (Region=4), chiếm Page 2 và Page 3, được ánh xạ tới Frame 3 và Frame 2. Bảng trang được mở rộng.



Hình 2: Output cấp phát bộ nhớ cho PID 1 tại Time slot 3

Giải phóng bộ nhớ (FREE):

- Khi tiến trình gọi `FREE reg`, ví dụ trong `os_1_mlq_paging` tại Time slot 4, PID 1 thực hiện `FREE 0`. Output cho thấy: `===== PHYSICAL MEMORY AFTER DEALLOCATION =====` `PID=1 - Region=0` `print_pgtbl: 0 - 1024` (Bảng trang không thay đổi ngay lập tức) Việc giải phóng chỉ đánh dấu vùng nhớ ảo là trống (trong `vm_freerg_list`)

Ghi bộ nhớ (WRITE):

- Ví dụ trong `os_1_mlq_paging` tại Time slot 6, PID 1 thực hiện `WRITE 100 1 20` (ghi giá trị 100 vào vùng nhớ 1, offset 20). Output: `===== PHYSICAL MEMORY AFTER WRITING =====` `write region=1 offset=20 value=100` `print_pgtbl: 0 - 1024` (Bảng trang không đổi) `================================================================` `===== PHYSICAL MEMORY DUMP =====` `BYTE 00000114: 100` (Giá trị 100 được ghi vào địa chỉ vật lý tương ứng) `===== PHYSICAL MEMORY END-DUMP =====` Địa chỉ ảo `Region=1` (được cấp phát ở slot 5 tại `0x00000000`) + `offset=20` (0x14) = `0x00000014`. Địa chỉ ảo này thuộc Page 0. Page 0 được ánh xạ tới Frame 1. Do đó, giá trị 100 được ghi vào byte tại offset 0x14 trong Frame 1. Memory dump cho thấy `BYTE 00000114: 100`, địa chỉ `0x114` có thể là địa chỉ vật lý tương ứng trong Frame 1 (Frame 1 * PageSize + Offset = 1 * 256 + 20 = 276

= 0x114).

Hình 3: Output ghi bộ nhớ và memory dump tại Time slot

```
================================================================
================================================================
===== PHYSICAL MEMORY AFTER WRITING =====
write region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
        CPU 2: Put process  2 to run queue
        CPU 1: Put process  3 to run queue
        CPU 1: Dispatched process  3
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000114: 100
===== PHYSICAL MEMORY AFTER DEALLOCATION =====
PID=3 - Region=0
print_pgtbl: 0 - 512
00000000: 80000005
00000004: 80000004
Page Number: 0 -> Frame Number: 5
===== PHYSICAL MEMORY END-DUMP =====
Page Number: 1 -> Frame Number: 4
        CPU 2: Dispatched process  2
Time slot   7
================================================================
        Loaded a process at input/proc/m0s, PID: 5 PRIO: 120
```

```
        Loaded a process at input/proc/s2, PID: 4 PRIO: 120
Time slot   6
        CPU 3: Put process  1 to run queue
        CPU 0: Dispatched process  4
        CPU 3: Dispatched process  1
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=1 - Region=1 - Address=00000000 - Size=100 byte
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=3 - Region=1 - Address=0000012C - Size=100 byte
print_pgtbl: 0 - 512
00000000: 80000005
00000004: 80000004
Page Number: 0 -> Frame Number: 5
Page Number: 1 -> Frame Number: 4
================================================================
================================================================
```

Đọc bộ nhớ (READ):

- Ví dụ trong `os_1_mlq_paging` tại Time slot 7, PID 1 thực hiện `READ 1 20 1` (đọc từ vùng 1, offset 20 vào thanh ghi 1). Output: `===== PHYSICAL MEMORY AFTER READING =====` `read region=1 offset=20 value=100` (Đọc thành công giá trị 100 đã ghi trước đó) `print_pgtbl: 0 - 1024` `==========` `=====` `PHYSICAL MEMORY DUMP =====` `BYTE 00000114: 100` `===== PHYSICAL MEMORY END-DUMP =====` Quá trình đọc diễn ra tương tự ghi, MMU dịch địa chỉ ảo `0x00000014` sang địa chỉ vật lý `0x00000114` và đọc giá trị tại đó.

Swapping: Output cung cấp không hiển thị rõ ràng các thông báo về swapping (ví dụ: swap in/out). Tuy nhiên, trong các trường hợp chạy với bộ nhớ nhỏ (`os_1_mlq_paging_small_1k`, `os_1_mlq_paging_small_4k`), cơ chế swapping có thể đã được kích hoạt ngầm khi RAM hết khung trống, nhưng không có thông báo cụ thể trong output này. Hoạt động swapping phức tạp hơn và cần các syscall `MEMSWP` để xử lý page fault.

3. SystemCall

Phân tích ouput theo dòng thời gian:

```
Time slot   0
ld_routine
Time slot   1
Time slot   2
Time slot   3
Time slot   4
Time slot   5
Time slot   6
Time slot   7
Time slot   8
Time slot   9
        Loaded a process at input/proc/sc2, PID: 1 PRIO: 15
```

Time slot 0-8: modun ld_routine khởi tạo hệ thống

Chưa có process nào được nạp

Time slot 9: Process 1 được nạp, PID=1, prio=15

```
Time slot  10
        CPU 0: Dispatched process  1
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=1 - Region=1 - Address=00000000 - Size=100 byte
print_pgtbl: 0 - 256
00000000: 80000000
Page Number: 0 -> Frame Number: 0
================================================================
        Loaded a process at input/proc/sc2, PID: 2 PRIO: 0
```

Time slot 10: chạy Process 1

```
Time slot  11
================================================================
===== PHYSICAL MEMORY AFTER WRITING =====
write region=1 offset=0 value=80
print_pgtbl: 0 - 256
00000000: 80000000
Page Number: 0 -> Frame Number: 0
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 80
===== PHYSICAL MEMORY END-DUMP =====
```

Time slot 11: Process 1 ghi vào bộ nhớ

```
Time slot  12
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=2 - Region=1 - Address=00000000 - Size=100 byte
print_pgtbl: 0 - 256
00000000: 80000001
Page Number: 0 -> Frame Number: 1
================================================================
```

Time slot 12: Chuyển process: Process 1 -> Process 2

```
Time slot  13
================================================================
===== PHYSICAL MEMORY AFTER WRITING =====
write region=1 offset=0 value=80
print_pgtbl: 0 - 256
00000000: 80000001
Page Number: 0 -> Frame Number: 1
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 80
BYTE 00000100: 80
===== PHYSICAL MEMORY END-DUMP =====
Time slot  14
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
================================================================
===== PHYSICAL MEMORY AFTER WRITING =====
write region=1 offset=1 value=48
print_pgtbl: 0 - 256
00000000: 80000001
Page Number: 0 -> Frame Number: 1
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 80
BYTE 00000100: 80
BYTE 00000101: 48
===== PHYSICAL MEMORY END-DUMP =====
Time slot  15
================================================================
===== PHYSICAL MEMORY AFTER WRITING =====
write region=1 offset=2 value=-1
print_pgtbl: 0 - 256
00000000: 80000001
Page Number: 0 -> Frame Number: 1
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 80
BYTE 00000100: 80
BYTE 00000101: 48
BYTE 00000102: -1
===== PHYSICAL MEMORY END-DUMP =====
```

Time slot 13-16: Process 2 ghi và đọc bộ nhớ

```
Time slot  16
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
================================================================
===== PHYSICAL MEMORY AFTER READING =====
read region=1 offset=0 value=80
print_pgtbl: 0 - 256
00000000: 80000001
Page Number: 0 -> Frame Number: 1
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 80
BYTE 00000100: 80
BYTE 00000101: 48
BYTE 00000102: -1
===== PHYSICAL MEMORY END-DUMP =====
================================================================
===== PHYSICAL MEMORY AFTER READING =====
read region=1 offset=1 value=48
print_pgtbl: 0 - 256
00000000: 80000001
Page Number: 0 -> Frame Number: 1
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 80
BYTE 00000100: 80
BYTE 00000101: 48
BYTE 00000102: -1
===== PHYSICAL MEMORY END-DUMP =====
================================================================
===== PHYSICAL MEMORY AFTER READING =====
read region=1 offset=2 value=-1
print_pgtbl: 0 - 256
00000000: 80000001
Page Number: 0 -> Frame Number: 1
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 80
BYTE 00000100: 80
BYTE 00000101: 48
BYTE 00000102: -1
===== PHYSICAL MEMORY END-DUMP =====
The procname retrieved from memregionid 1 is "P0"
```

```
Time slot  17
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  1
================================================================
===== PHYSICAL MEMORY AFTER WRITING =====
write region=1 offset=1 value=48
print_pgtbl: 0 - 256
00000000: 80000000
Page Number: 0 -> Frame Number: 0
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 80
BYTE 00000001: 48
BYTE 00000100: 80
BYTE 00000101: 48
BYTE 00000102: -1
===== PHYSICAL MEMORY END-DUMP =====
```

Time slot 17: Process 2 hoàn tất, chuyển process sang Process 1

```
Time slot  18
================================================================
===== PHYSICAL MEMORY AFTER WRITING =====
write region=1 offset=2 value=-1
print_pgtbl: 0 - 256
00000000: 80000000
Page Number: 0 -> Frame Number: 0
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 80
BYTE 00000001: 48
BYTE 00000002: -1
BYTE 00000100: 80
BYTE 00000101: 48
BYTE 00000102: -1
===== PHYSICAL MEMORY END-DUMP =====
```

Time slot 18-19: Process 1 đọc, ghi vào bộ nhớ

```
Time slot  19
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
================================================================
===== PHYSICAL MEMORY AFTER READING =====
read region=1 offset=0 value=80
00000000: 80000000
Page Number: 0 -> Frame Number: 0
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 80
BYTE 00000001: 48
BYTE 00000002: -1
BYTE 00000100: 80
BYTE 00000101: 48
BYTE 00000102: -1
===== PHYSICAL MEMORY END-DUMP =====
================================================================
===== PHYSICAL MEMORY AFTER READING =====
read region=1 offset=1 value=48
print_pgtbl: 0 - 256
00000000: 80000000
Page Number: 0 -> Frame Number: 0
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 80
BYTE 00000001: 48
BYTE 00000002: -1
BYTE 00000100: 80
BYTE 00000101: 48
BYTE 00000102: -1
===== PHYSICAL MEMORY END-DUMP =====
================================================================
===== PHYSICAL MEMORY AFTER READING =====
read region=1 offset=2 value=-1
print_pgtbl: 0 - 256
00000000: 80000000
Page Number: 0 -> Frame Number: 0
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 80
BYTE 00000001: 48
BYTE 00000002: -1
BYTE 00000100: 80
BYTE 00000101: 48
BYTE 00000102: -1
===== PHYSICAL MEMORY END-DUMP =====
```

Time slot 20: process 1 hoàn tất, CPU ngừng hoạt động

Sự tương tác giữa 3 modun: Bộ nhớ lưu tên tiến trình(memory storing process name), Quản lý tiến trình của HĐH (OS Process Control), Quản lý hàng đợi lập lịch(Scheduling Queue Management):

Thời điểm 10: Tiến trình 1 được nạp và bắt đầu thực thi

- **Mô-đun tương tác**:
  - **Quản lý tiến trình**: Cấp phát PCB cho PID 1, đặt PRIO = 15.
  - **Bộ nhớ**: Cấp phát bộ nhớ, ánh xạ Page 0 tới Frame 0, lưu trữ dữ liệu tiến trình 1.
  - **Lập lịch**: Đưa tiến trình 1 vào hàng đợi sẵn sàng (Ready Queue).

Thời điểm 12: Tiến trình 1 bị đưa ra khỏi CPU, Tiến trình 2 được nạp và bắt đầu thực thi

- **Mô-đun tương tác**:
  - **Quản lý tiến trình**: Tiến trình 2 được nạp (PID=2), cấp phát PCB và đặt độ ưu tiên PRIO = 0.
  - **Bộ nhớ**: Cấp phát bộ nhớ cho tiến trình 2, ánh xạ Page 0 tới Frame 1.
  - **Lập lịch**: Tiến trình 1 được đưa lại hàng đợi, và CPU 0 chuyển sang thực thi tiến trình 2.

Thời điểm 17: Tiến trình 2 kết thúc và tiến trình 1 tiếp tục thực thi

- **Mô-đun tương tác**:
  - **Quản lý tiến trình**: Tiến trình 2 kết thúc và tiến trình 1 tiếp tục được đưa vào CPU.
  - **Bộ nhớ**: Cập nhật lại bộ nhớ cho tiến trình 1, ghi giá trị mới vào bộ nhớ tại các offset.
  - **Lập lịch**: Tiến trình 1 được đưa vào hàng đợi lại và CPU 0 tiếp tục thực thi.

Thời điểm 19–20: Tiến trình 1 kết thúc

- **Mô-đun tương tác**:
    - **Quản lý tiến trình**: Tiến trình 1 kết thúc, hệ thống sẽ dừng.
    - **Bộ nhớ**: Cập nhật bộ nhớ để phản ánh kết thúc tiến trình.
    - **Lập lịch**: Tiến trình 1 bị loại bỏ khỏi hàng đợi, CPU 0 dừng.

4. OVERALL

Sau khi tiến hành mô phỏng hệ thống, bao gồm các phần **Lập lịch tiến trình (Scheduling)**, **Quản lý bộ nhớ (Memory)** và **Hệ thống gọi (SystemCall)**, chúng ta có thể phân tích và đưa ra các nhận xét tổng thể về hiệu suất hoạt động của hệ thống như sau:

1. **Scheduling**

Hệ thống sử dụng **Multi-Level Queue (MLQ)** để lập lịch, giúp phân loại tiến trình theo mức độ ưu tiên và xử lý chúng trên các hàng đợi khác nhau. Kết quả cho thấy:

- **Biểu đồ Gantt** cho thấy các tiến trình được phân phối CPU theo các mức độ ưu tiên khác nhau. Các tiến trình có độ ưu tiên cao sẽ được thực thi trước, và các tiến trình có độ ưu tiên thấp sẽ phải chờ đợi lâu hơn.
- Hệ thống có thể xử lý hiệu quả các tiến trình yêu cầu CPU cao, nhưng cần cải thiện độ phản hồi của các tiến trình ưu tiên thấp trong trường hợp có nhiều tiến trình yêu cầu tài nguyên.
- **MLQ** giúp hệ thống dễ dàng xử lý các nhóm tiến trình có yêu cầu khác nhau, nhưng vẫn có thể gặp phải tình trạng **starvation** (tiến trình ưu tiên thấp không được thực thi trong một thời gian dài).

2. **Memory**

Với **phân trang hai lớp**, hệ thống quản lý bộ nhớ hiệu quả hơn trong việc phân phối bộ nhớ cho các tiến trình, đặc biệt là khi bộ nhớ được chia thành các khung nhỏ và được ánh xạ đến các trang. Các kết quả cho thấy:

- Bộ nhớ được phân bổ và truy xuất một cách hợp lý, với các tiến trình được cấp phát bộ nhớ ở các **khung bộ nhớ (frames)** khác nhau.
- **Phân trang hai lớp** giúp giảm thiểu phân mảnh bộ nhớ và tăng hiệu quả sử dụng bộ nhớ, nhưng vẫn cần cải thiện việc quản lý khi hệ thống có nhiều tiến trình yêu cầu bộ nhớ lớn.

- Quá trình ghi và đọc vào bộ nhớ được thực hiện chính xác và có sự theo dõi của bảng trang (page table), giúp tăng tính chính xác khi quản lý bộ nhớ trong môi trường đa tiến trình.

3. Systemcall

Trong phần **Syscall**, các mô-đun bộ nhớ, quản lý tiến trình và lập lịch tương tác chặt chẽ với nhau. Các kết quả cho thấy:

- Các **tín hiệu (signals)** được sử dụng để đồng bộ hóa giữa các tiến trình và hệ thống, giúp các tiến trình được đưa vào hàng đợi và thực thi một cách hợp lý.
- **Đồng bộ hóa (synchronization)** giữa các mô-đun giúp tránh các vấn đề như deadlock hoặc race conditions khi các tiến trình truy cập tài nguyên chung.
- Hệ thống có thể xử lý các tín hiệu đồng bộ hiệu quả, đảm bảo rằng tiến trình được thực thi đúng lúc và dữ liệu không bị ghi đè hoặc bị truy xuất đồng thời không an toàn.

4. Cải thiện hệ thống:

- **Cải thiện thuật toán lập lịch** trong MLQ, đặc biệt là giảm thiểu tình trạng **starvation** đối với các tiến trình ưu tiên thấp, có thể áp dụng một cơ chế như **aging** để tăng ưu tiên của tiến trình khi nó bị chờ quá lâu.
- Nếu bộ nhớ không đủ dung lượng chứa, os sẽ mắc kẹt trong một vòng lặp **swap giữa 2 page liên tục**
- **Hệ thống sử dụng mutexlock**, để giảm thời gian chạy có thể sử dụng các lock khác như: Spinlock, Read-write lock