ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC & KỸ THUẬT MÁY TÍNH

BÀI TẬP LỚN
HỆ ĐIỀU HÀNH (CO2018)

CHỦ ĐỀ

Simple Operating System

Lớp: L02 – Nhóm 7 – Học Kỳ HK251
Ngày nộp: 15/12/2025

GVHD: Thầy Nguyễn Minh Tâm

Sinh viên thực hiện	Mã số sinh viên
Mai Xuân Bách	2310197
Võ Đặng Thanh Bình	2310316
Trần Minh Dương	2310609
Vũ Thành Đạt	2310718
Phạm Công Võ	2313946

Ho Chi Minh, Tháng 12/2025

PHÂN CÔNG NHIỆM VỤ VÀ KẾT QUẢ THỰC HIỆN ĐỀ TÀI CỦA TỪNG THÀNH VIÊN NHÓM 7

STT	Họ và tên	MSSV	Nhiệm vụ	Kết quả	Chữ ký
1	Mai Xuân Bách	2310197	• Hiện thực Memory và Paging • Trả lời câu hỏi	100%	
2	Võ Đặng Thanh Bình	2310316	• Hiện thực Memory và Paging • Trả lời câu hỏi	100%	
3	**Trần Minh Dương**	2310609	• Hiện thực Memory và Paging • Trả lời câu hỏi • Tổng hợp, chỉnh sửa code • Viết report	100%	
4	Vũ Thành Đạt	2310718	• Hiện thực Memory và Paging • Trả lời câu hỏi	100%	
5	Phạm Công Võ	2313946	• Hiện thực Scheduling • Trả lời câu hỏi • Viết báo cáo	100%	

Mục lục

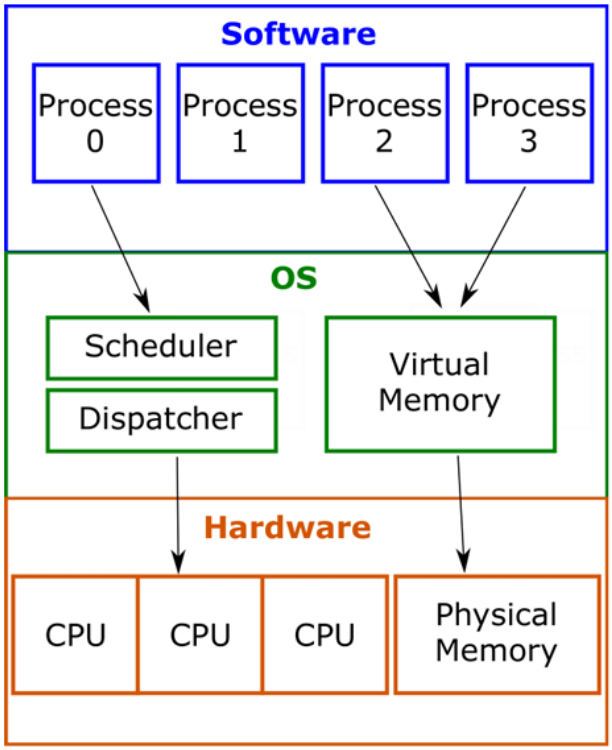
Danh sách hình vẽ

1 Phần Mở Đầu

1.1 Giới thiệu Hệ điều hành (Operating System)

Trong các hệ thống máy tính hiện đại, hệ điều hành **(Operating System - OS)** được xem là *"bộ não điều phối"* của toàn bộ hệ thống. OS không chỉ cung cấp nền tảng cho mọi ứng dụng vận hành mà còn kiểm soát cách tài nguyên phần cứng - CPU, bộ nhớ, thiết bị I/O, bus hệ thống - được phân chia và sử dụng một cách hiệu quả, bảo mật và ổn định. Ở tầng bề mặt, người dùng chỉ thấy những tác vụ quen thuộc như khởi động một ứng dụng, cấp phát bộ nhớ hay thực hiện I/O. Tuy nhiên, phía sau đó là hàng loạt cơ chế tinh vi hoạt động liên tục bên trong nhân hệ điều hành, đảm bảo cho hệ thống luôn vận hành đúng và đạt hiệu năng cao.

Một chương trình chỉ có thể chạy khi OS cấp CPU cho nó; nó chỉ có thể truy cập vùng nhớ khi OS tạo ánh xạ địa chỉ và bảo vệ không gian riêng; nó chỉ có thể tương tác với thiết bị ngoại vi thông qua driver và hệ thống ngắt do OS quản lý. Vì vậy, hiệu năng tổng thể của máy tính và trải nghiệm người dùng luôn phụ thuộc trực tiếp vào cách hệ điều hành tổ chức, giám sát và điều phối tài nguyên.



Hình 1: *Kiến trúc tổng quan của hệ điều hành.*

Báo cáo của IEEE Computer Society (2024) chỉ ra rằng 87% lỗi nghiêm trọng trong các hệ thống đa nhiệm bắt nguồn từ ba mô-đun lõi của OS:

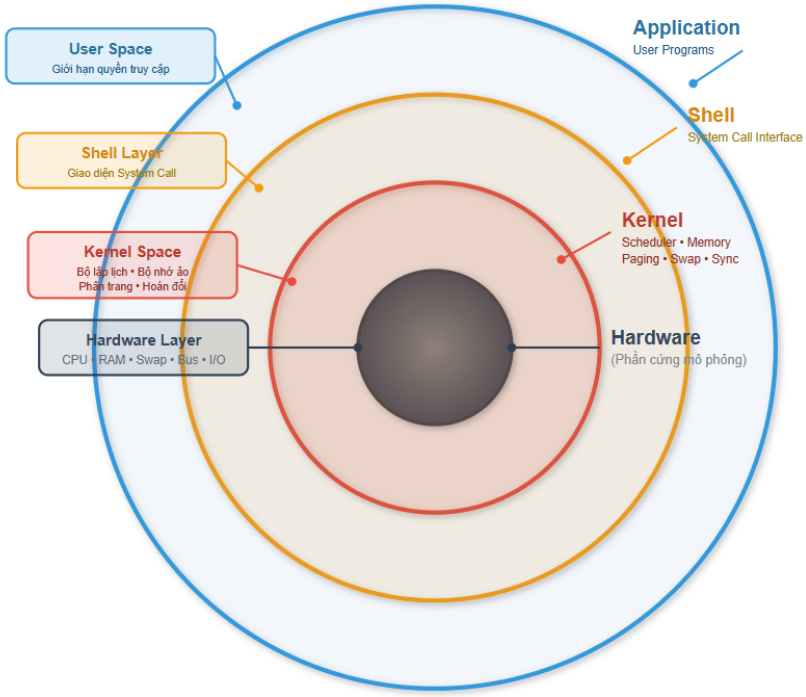
- **Bộ lập lịch (Scheduler)** - quyết định tiến trình nào được thực thi và kiểm soát thứ tự, mức ưu tiên.
- **Hệ thống bộ nhớ ảo (Virtual Memory System)** - chịu trách nhiệm phân vùng, bảo vệ và ánh xạ không gian địa chỉ.
- **Cơ chế đồng bộ hóa (Synchronization)** - đảm bảo an toàn khi nhiều tiến trình hoặc nhiều CPU truy cập tài nguyên dùng chung.

Điều này cho thấy việc hiểu rõ cách OS vận hành không chỉ quan trọng trong học thuật mà còn mang tính thực tiễn cao trong công nghiệp, nơi các lỗi như *deadlock*, *race condition*, *starvation* hay *memory corruption* có thể gây ra sự cố nghiêm trọng.

Dựa trên yêu cầu đó, bài tập lớn *Simple Operating System* được xây dựng nhằm cung cấp một mô hình hệ điều hành thu gọn nhưng vẫn bảo toàn các cơ chế nền tảng như trong nhân Linux. Thay vì quan sát ở mức ứng dụng, sinh viên trực tiếp hiện thực và kiểm chứng các mô-đun của hệ điều hành, qua đó hình thành tư duy thiết kế hệ thống (system-level thinking) - năng lực quan trọng đối với phát triển kernel, hệ thống nhúng và các nền tảng hiệu năng cao.

1.2 Kiến trúc hệ thống Simple Operating System

Kiến trúc của hệ điều hành mô phỏng được xây dựng theo mô hình ba tầng, phản ánh cấu trúc điển hình của một hệ điều hành hiện đại:



Hình 2: *Kiến trúc phân tầng - OS Layers*

1. **User Space:** chịu trách nhiệm thực thi các chương trình mô phỏng và phát sinh yêu cầu đến nhân thông qua cơ chế lời gọi hệ thống. Tầng này bị giới hạn quyền truy cập và không can thiệp trực tiếp vào tài nguyên phần cứng.

2. **Kernel Space:** là tầng điều khiển trung tâm, nơi triển khai các mô-đun cốt lõi của hệ điều hành, bao gồm bộ lập lịch tiến trình (*Scheduler*), bộ quản lý bộ nhớ ảo (*Virtual Memory Manager*), hệ thống phân trang (*Paging System*), cơ chế hoán đổi (*Swap Manager*), giao diện lời gọi hệ thống (*System Call Interface*) và lớp đồng bộ hóa (*Synchronization Layer*). Tất cả quyết định về phân phối và quản lý tài nguyên của hệ thống đều được xử lý tại tầng này.

3. **Hardware Layer (mô phỏng):** mô phỏng các thành phần phần cứng như bộ xử lý đa lõi, bộ nhớ chính, thiết bị hoán đổi, bus bộ nhớ và thiết bị I/O. Tất cả hoạt động của nhân hệ điều hành đều dựa trên lớp phần cứng mô phỏng này.

1.3 Yêu cầu đề tài

Đề tài *Simple Operating System* hướng đến việc xây dựng một hệ điều hành thu gọn nhưng vẫn phản ánh đầy đủ bản chất vận hành của các cơ chế lõi trong OS hiện đại. Sinh viên không chỉ mô phỏng hành vi bề mặt, mà phải trực tiếp hiện thực những mô-đun nền tảng vốn tồn tại trong nhân Linux. Điều này đòi hỏi mỗi mô-đun phải được thiết kế đúng nguyên lý, hoạt động ổn định và phối hợp chặt chẽ trong một kiến trúc hệ thống thống nhất.

Các mô-đun bắt buộc cần triển khai bao gồm:

- **Bộ lập lịch Multi-Level Queue (MLQ):** tổ chức tiến trình theo nhiều mức ưu tiên, phân phối CPU bằng time-slice, hỗ trợ preemption và mô phỏng tinh thần của O(1) Scheduling trong Linux.
- **Quản lý bộ nhớ ảo (Virtual Memory Manager):** tổ chức không gian địa chỉ thông qua `vm_area` và `vm_region`, thực hiện cấp phát–thu hồi–mở rộng heap bằng `sbrk`, quản lý phân mảnh bằng free-list.
- **Hệ thống phân trang (Paging System):** hỗ trợ phân trang một cấp (32-bit) và phân trang nhiều cấp (64-bit), thiết lập ánh xạ địa chỉ ảo–vật lý, xử lý page fault và duy trì các thuộc tính bảo vệ trang.
- **Cơ chế hoán đổi (Swapping):** mô phỏng quá trình swap-in/swap-out khi RAM thiếu, lựa chọn trang nạn nhân theo chiến lược gần với OS thực.
- **System Call Interface:** truyền tham số từ user-space, kiểm tra tính hợp lệ của vùng nhớ và thực hiện chuyển chế độ user → kernel an toàn.
- **Đồng bộ hóa (Synchronization Layer):** bảo vệ tài nguyên dùng chung như ready-queue và free-frame-list, ngăn ngừa race condition và bảo đảm tính nhất quán dữ liệu trong hệ thống.

Bên cạnh phần hiện thực, đề tài yêu cầu phân tích các vấn đề lý thuyết như ưu điểm MLQ, vai trò segmentation, lý do phân trang đa cấp trong 64-bit, so sánh paging–segmentation, tác động của system call chậm và hệ quả khi thiếu đồng bộ hóa.

Để triển khai các mô-đun trên, sinh viên cần nắm các kiến thức nền tảng gồm: nguyên lý lập lịch, tổ chức bộ nhớ ảo, cấu trúc bảng trang và phân trang nhiều cấp, cơ chế hoán đổi, nguyên lý lời gọi hệ thống và các kỹ thuật đồng bộ hóa cơ bản. Những kiến thức này là nền tảng để sinh viên thiết kế và vận hành chính xác hệ điều hành mô phỏng.

1.4 Mục đích nghiên cứu của đề tài

Mục tiêu của đề tài là đánh giá khả năng thiết kế và hiện thực các cơ chế cốt lõi của hệ điều hành trong một môi trường mô phỏng có thể kiểm chứng. Cụ thể:

- Hiện thực và phân tích các cơ chế nền tảng như lập lịch, phân trang, cấp phát bộ nhớ, hoán đổi và xử lý lời gọi hệ thống.
- Quan sát và đánh giá các tình huống thực tế: deadlock, race condition, lỗi trang, thrashing hoặc hư hỏng bộ nhớ.
- Tạo nền tảng thực hành vững chắc cho các môn học nâng cao như thiết kế hệ điều hành, kiến trúc máy tính, lập trình hệ thống và an ninh mức nhân.

2 Phần Nội Dung

2.1 Bộ lập lịch (Scheduler)

2.1.1 Scheduler và vai trò trong hệ điều hành

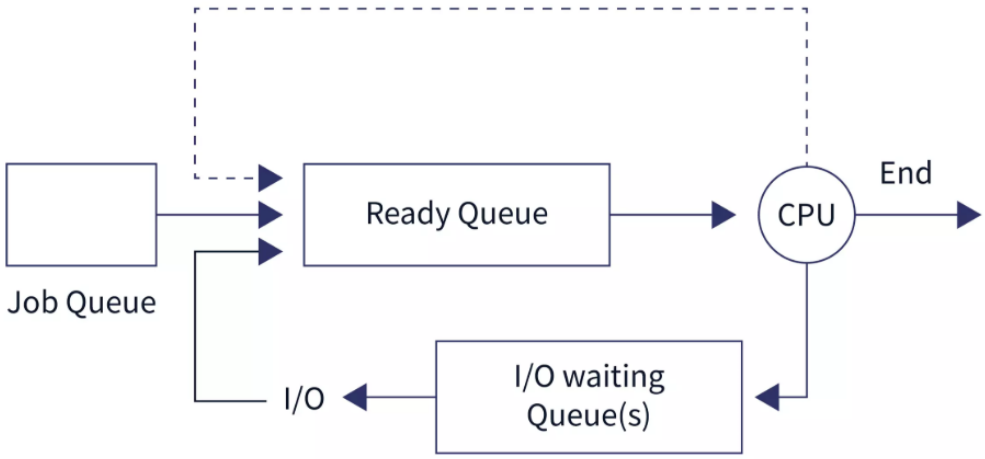
Scheduler là một trong những thành phần cốt lõi và quan trọng nhất của hệ điều hành, giữ vai trò điều phối và phân phối tài nguyên CPU cho các tiến trình đang hoạt động trong hệ thống. Trong môi trường đa nhiệm (*multitasking*), dù có thể tồn tại nhiều tiến trình đồng thời, nhưng tại một thời điểm, mỗi CPU chỉ có thể thực thi duy nhất một tiến trình. Do đó, Scheduler chịu trách nhiệm quyết định:

- Tiến trình nào được cấp CPU,
- Thời điểm tiến trình đó được thực thi,
- Khoảng thời gian mà tiến trình được phép sử dụng CPU.

Một Scheduler được xem là hoạt động hiệu quả khi đồng thời đảm bảo các mục tiêu sau:

- Tối ưu mức sử dụng CPU, hạn chế tình trạng CPU nhàn rỗi,
- Giảm thời gian chờ của các tiến trình,
- Đảm bảo tính công bằng trong việc phân phối tài nguyên,
- Hạn chế hiện tượng đói tài nguyên (*starvation*), tức là tình trạng một tiến trình bị chờ vô hạn do không bao giờ được cấp CPU.

Về mặt tổ chức hoạt động, các tiến trình trong hệ điều hành lần lượt đi qua các trạng thái như: *Job Queue* → *Ready Queue* → *CPU* → *I/O Waiting Queue* → quay lại *Ready Queue* hoặc kết thúc. Trong đó, Scheduler đảm nhiệm vai trò lựa chọn tiến trình từ *Ready Queue* để cấp phát CPU, đồng thời xử lý việc đưa tiến trình về hàng đợi chờ khi xảy ra I/O hoặc khi hết thời gian xử lý.



Hình 3: *Sơ đồ tổng quát quá trình điều phối tiến trình của Scheduler*

Như vậy, Scheduler giữ vai trò trung tâm trong việc quyết định hiệu năng, độ ổn định và khả năng đáp ứng của toàn hệ thống.

2.1.2 Tiến trình, PCB và Ready Queue

a) Tiến trình và PCB (Process Control Block)

Trong hệ điều hành, mỗi chương trình khi được nạp vào bộ nhớ và bắt đầu thực thi sẽ được quản lý dưới dạng một *tiến trình* (*process*). Để theo dõi và điều khiển hoạt động của từng tiến trình, hệ điều hành sử dụng một cấu trúc dữ liệu chuyên dụng gọi là **PCB (Process Control Block)**.

PCB đóng vai trò là đơn vị quản lý trung tâm của tiến trình, lưu trữ đầy đủ các thông tin phục vụ cho quá trình lập lịch và chuyển ngữ cảnh, bao gồm:

- Mã định danh tiến trình (*PID*),
- Độ ưu tiên (*Priority*),
- Bộ đếm chương trình (*Program Counter – PC*),
- Trạng thái các thanh ghi (*Registers*),
- Vùng lệnh (*Code segment*),
- Thông tin quản lý bộ nhớ (bảng trang, heap, stack,...).

Scheduler không làm việc trực tiếp với chương trình, mà thông qua PCB để thực hiện các thao tác như cấp phát CPU, lưu trạng thái khi ngắt, và khôi phục tiến trình khi tiếp tục thực thi. Do đó, PCB giữ vai trò then chốt trong việc đảm bảo tính chính xác, an toàn và liên tục của quá trình chuyển đổi giữa các tiến trình.

b) Ready Queue và cơ chế chờ CPU

Sau khi được tạo và đủ điều kiện thực thi, các tiến trình sẽ được đưa vào **Ready Queue (hàng đợi sẵn sàng)** để chờ được cấp phát CPU. Ready Queue là cấu trúc dữ liệu dùng để quản lý danh sách các tiến trình đang ở trạng thái sẵn sàng chạy.

- Trong hệ điều hành đơn giản, hệ thống chỉ sử dụng một Ready Queue duy nhất.
- Trong các hệ thống hiện đại, Ready Queue có thể được tổ chức thành nhiều hàng đợi ứng với các mức độ ưu tiên khác nhau nhằm nâng cao hiệu quả lập lịch.

Scheduler có nhiệm vụ lựa chọn tiến trình phù hợp từ Ready Queue để cấp CPU. Khi tiến trình:

- Hết thời gian xử lý nhưng chưa hoàn thành → được đưa lại vào Ready Queue,
- Hoàn thành chương trình → sẽ bị loại khỏi hệ thống.

Cơ chế này đảm bảo CPU được phân phối cho các tiến trình theo đúng chính sách lập lịch, đảm bảo tính luân phiên, công bằng và hiệu quả trong việc sử dụng tài nguyên.

2.1.3 Multi-Level Queue (MLQ) kết hợp Round Robin

Hệ điều hành trong bài tập lớn áp dụng cơ chế lập lịch **Multi-Level Queue (MLQ)** kết hợp với thuật toán **Round Robin**, cho phép tổ chức các tiến trình thành nhiều hàng đợi độc lập theo mức độ ưu tiên, đồng thời đảm bảo tính công bằng trong từng nhóm ưu tiên. Đây là mô hình lập lịch phù hợp với các hệ thống đa nhiệm, nơi các tiến trình có yêu cầu xử lý khác nhau về thời gian đáp ứng và mức độ quan trọng.

Trong mô hình MLQ, mỗi tiến trình được gán một giá trị ưu tiên *prio* thỏa mãn:

$$prio \in [0, MAX\_PRIO - 1] \tag{1}$$

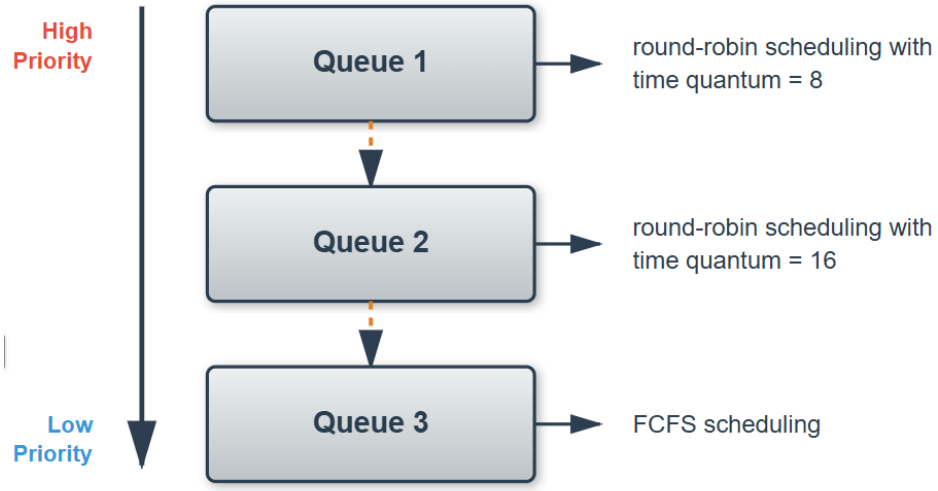
trong đó *prio* càng nhỏ thì mức độ ưu tiên càng cao. Giá trị này được sử dụng để xác định *ready queue* mà tiến trình sẽ được đưa vào. Mỗi hàng đợi có độ ưu tiên cố định (*fixed priority*) và Scheduler luôn phục vụ từ hàng đợi có độ ưu tiên cao nhất xuống thấp nhất.

Để điều tiết thời gian sử dụng CPU giữa các hàng đợi, mỗi hàng đợi được cấp một số lượt chiếm CPU hữu hạn, gọi là *slot*, được xác định theo công thức:

$$slot = MAX\_PRIO - prio \tag{2}$$

Theo cơ chế này, hàng đợi có độ ưu tiên càng cao thì càng được cấp nhiều slot, trong khi các hàng đợi ưu tiên thấp chỉ nhận được số lượt thực thi hạn chế. Scheduler sẽ cho hàng đợi có mức ưu tiên cao nhất thực thi trước cho đến khi sử dụng hết toàn bộ slot được cấp, sau đó CPU mới được chuyển sang hàng đợi có mức ưu tiên thấp hơn. Khi tất cả các hàng đợi đều đã dùng hết slot, hệ thống sẽ tái cấp phát slot theo công thức ban đầu và bắt đầu một chu kỳ lập lịch mới.

Bên trong mỗi hàng đợi MLQ, các tiến trình được điều phối theo thuật toán **Round Robin**. Theo đó, mỗi tiến trình chỉ được phép chiếm CPU trong một khoảng thời gian xác định (*time slice*). Khi hết *time slice*, nếu tiến trình chưa hoàn thành thì sẽ bị đưa về cuối hàng đợi, còn CPU được chuyển sang tiến trình tiếp theo trong cùng hàng đợi. Cơ chế này đảm bảo không có tiến trình nào chiếm CPU liên tục trong thời gian dài, đồng thời duy trì tính công bằng giữa các tiến trình có cùng mức ưu tiên.



Hình 4: *Mô hình Multi-Level Queue kết hợp Round Robin với các mức time quantum khác nhau*

Ví dụ trong Linux

Trong hệ thống Linux, miền ưu tiên được xác định như sau:

$$MAX\_PRIO = 140, \quad prio = 0 \ldots 139 \tag{3}$$

Quan hệ giữa *prio*, *slot* và mức độ ưu tiên được thể hiện trong Bảng 2.

Bảng 2: *Quan hệ giữa prio, slot và mức độ ưu tiên trong Linux*

prio	slot = 140 − prio	Mức ưu tiên
0	140	Cao nhất
1	139	Rất cao
⋮	⋮	⋮
138	2	Rất thấp
139	1	Thấp nhất

Từ Bảng 2, có thể thấy hàng đợi có *prio = 0* được cấp 140 slot CPU, trong khi hàng đợi có *prio = 139* chỉ được cấp 1 slot. Cơ chế này đảm bảo các tiến trình ưu tiên cao được xử lý nhanh, trong khi các tiến trình nền vẫn được thực thi mà không làm ảnh hưởng đến hiệu năng tổng thể của hệ thống.

Sự kết hợp giữa MLQ và Round Robin tạo ra cơ chế lập lịch vừa đảm bảo tính ưu tiên, vừa duy trì tính công bằng, đồng thời nâng cao tính ổn định và hiệu quả sử dụng CPU của hệ thống.

CÂU HỎI

What is the advantage of the scheduling strategy used in this assignment in comparison with other scheduling algorithms you have learned?

Trả lời:

- Thuật toán lập lịch sử dụng trong bài tập lớn là **Multi-Level Queue (MLQ) kết hợp Round Robin**, đây là một mô hình lập lịch hiện đại, phù hợp với hệ điều hành đa nhiệm và thể hiện nhiều ưu điểm vượt trội so với các thuật toán truyền thống như *FCFS, SJF, ...*
- **Lập lịch theo độ ưu tiên**: mỗi tiến trình được gán một mức ưu tiên *prio* và được đưa vào hàng đợi tương ứng. Scheduler luôn lựa chọn tiến trình từ hàng đợi có độ ưu tiên cao nhất trước, nhờ đó các tiến trình quan trọng, yêu cầu thời gian đáp ứng nhanh được xử lý kịp thời, giúp cải thiện rõ rệt tính đáp ứng của hệ thống so với FCFS và Round Robin cơ bản.
- **Phân bổ CPU theo slot** với công thức

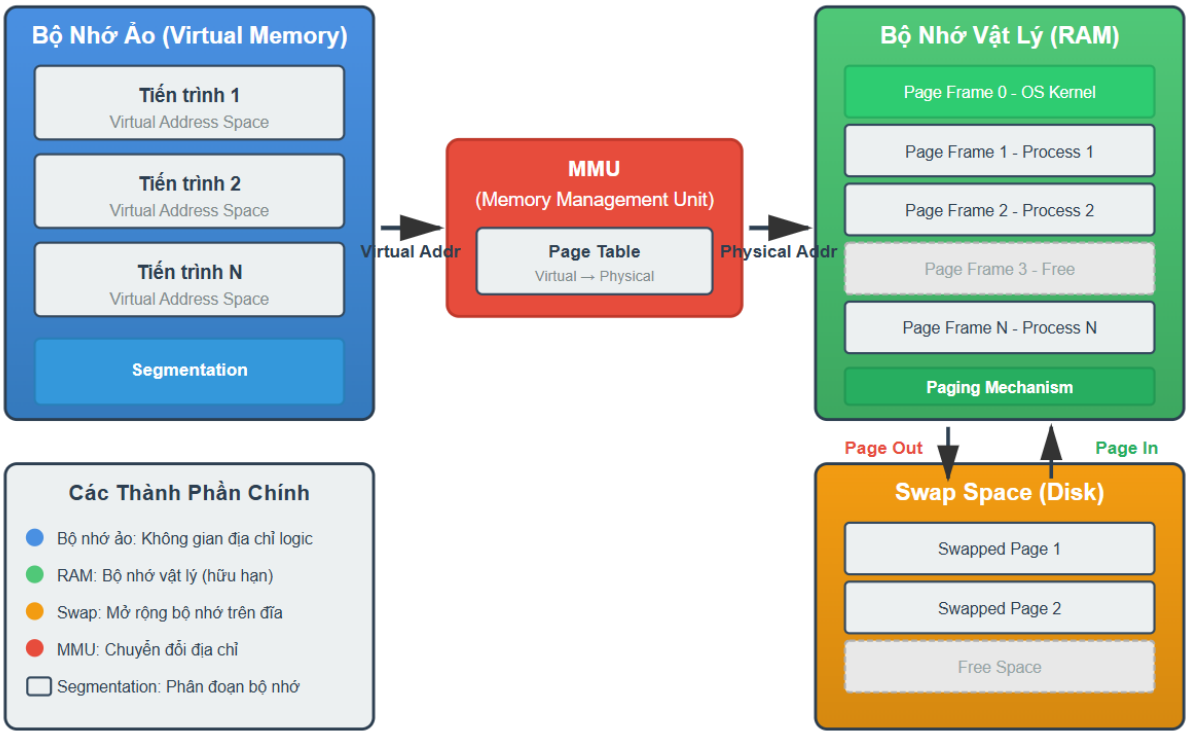
$$\text{slot} = \text{MAX\_PRIO} - \text{prio}$$

  cho phép điều tiết tài nguyên CPU một cách định hướng. Các hàng đợi ưu tiên cao được cấp nhiều lượt xử lý hơn, trong khi các hàng đợi ưu tiên thấp bị giới hạn số lượt thực thi.
- **Round Robin trong từng hàng đợi** đảm bảo tính công bằng giữa các tiến trình cùng mức ưu tiên, ỗi tiến trình chỉ được phép sử dụng CPU trong một khoảng thời gian xác định (time slice), giúp loại bỏ hiện tượng độc chiếm CPU (hạn chế của FCFS và SJF không ngắt).
- **Hỗ trợ lập lịch có quyền ngắt (preemptive scheduling)**: Khi tiến trình có độ ưu tiên cao hơn xuất hiện, hệ thống có thể ngay lập tức ngắt tiến trình đang chạy để chuyển CPU cho tiến trình ưu tiên cao hơn, đảm bảo tiến độ cho các tác vụ quan trọng và nâng cao khả năng đáp ứng thời gian thực của hệ thống.

2.2 Quản lý bộ nhớ (Memory Management)

2.2.1 Tổng quan về quản lý bộ nhớ

Quản lý bộ nhớ (Memory Management) là một trong những thành phần trọng yếu trong kiến trúc của hệ điều hành, chịu trách nhiệm tổ chức, phân phối và bảo vệ tài nguyên bộ nhớ trong suốt vòng đời hoạt động của tiến trình. Trong *Simple Operating System*, mô-đun quản lý bộ nhớ được thiết kế dựa trên các nguyên lý cơ bản của hệ điều hành hiện đại, kết hợp đồng thời **bộ nhớ ảo *(virtual memory)***, **phân đoạn *(segmentation)***, **phân trang *(paging)*** và **cơ chế hoán đổi *(swap)*** nhằm đảm bảo hệ thống vận hành hiệu quả ngay cả khi dung lượng RAM bị hạn chế.



Hình 5: *Tổng quan hệ thống quản lý bộ nhớ trong Simple OS*

Một đặc điểm quan trọng của Simple OS là mỗi tiến trình được cung cấp một không gian địa chỉ ảo độc lập (per-process virtual address space). Cơ chế này mang lại các lợi ích chính sau:

- **Cô lập tiến trình (Process Isolation):** giúp tiến trình không thể truy cập hoặc ghi đè vùng nhớ của tiến trình khác, tăng cường tính ổn định và an toàn cho hệ thống.
- **Trừu tượng hóa bộ nhớ (Memory Abstraction):** tiến trình làm việc hoàn toàn với địa chỉ ảo mà không cần quan tâm vị trí thực tế của dữ liệu trong bộ nhớ vật lý.
- **Tối ưu hóa tài nguyên:** hệ thống có thể linh hoạt cấp phát, thu hồi và tái sử dụng bộ nhớ, qua đó nâng cao hiệu suất và giảm phân mảnh.

Kiến trúc quản lý bộ nhớ của Simple OS bao gồm nhiều lớp chức năng liên kết chặt chẽ:

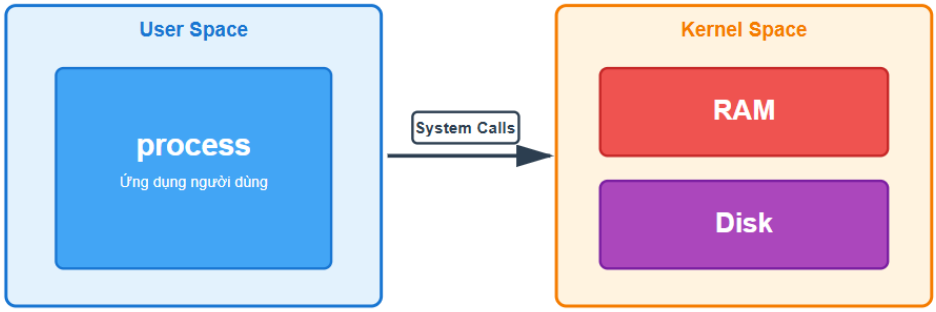
- Tổ chức không gian bộ nhớ ảo theo các vùng *code, data, heap, stack*.
- Cơ chế phân đoạn và ánh xạ vùng nhớ, quản lý các vùng liên tục thông qua danh sách free-region và bảng mô tả vùng đã cấp phát.

- Cơ chế phân trang (paging) cho phép ánh xạ linh hoạt giữa page ảo và frame vật lý, đồng thời theo dõi trạng thái của từng trang (present, swapped, dirty).
- Bộ nhớ vật lý và vùng hoán đổi (swap space), hỗ trợ lưu trữ tạm thời khi RAM không đủ.

Các phần tiếp theo của mục 2.2 sẽ trình bày chi tiết cơ chế tổ chức không gian nhớ, hoạt động phân trang, tiến trình hoán đổi và quá trình dịch địa chỉ ảo–vật lý trong Simple OS.

2.2.2 User/Kernel Memory Separation

Trong các hệ điều hành hiện đại, việc tách biệt không gian bộ nhớ người dùng (User Space) và không gian bộ nhớ nhân hệ điều hành (Kernel Space) là nguyên tắc cốt lõi nhằm đảm bảo tính an toàn, ổn định và khả năng kiểm soát tài nguyên. Dù chỉ là mô hình mô phỏng, *Simple Operating System* vẫn áp dụng đầy đủ cơ chế này, phản ánh sát cách vận hành của các hệ điều hành thực như Linux.



Hình 6: *Phân tách không gian bộ nhớ: User Space và Kernel Space trong Simple OS*

1. Khái niệm về phân tách User–Kernel

Không gian bộ nhớ của hệ thống được chia thành hai miền chức năng tách biệt:

- **User Space**: môi trường thực thi của các tiến trình ứng dụng. Các tiến trình ở đây hoạt động với mức quyền hạn chế và chỉ có thể tiếp cận tài nguyên hệ thống thông qua các lời gọi hệ thống (system calls).
- **Kernel Space**: vùng đặc quyền cao nhất của hệ điều hành, nơi vận hành các thành phần quan trọng như bộ lập lịch, bảng tiến trình, bộ quản lý bộ nhớ vật lý và hệ thống bảng trang.

Trong Simple OS, kernel là thành phần duy nhất được phép:

- truy cập trực tiếp RAM và không gian swap;
- quản lý và cập nhật page table của từng tiến trình;
- cấp phát, theo dõi và thu hồi các vùng nhớ ảo;
- thao tác lên các cấu trúc dữ liệu trọng yếu như *memphy_struct*, *mm_struct*, và *vm_area_struct*.

2. Lý do cần phân tách User–Kernel

a. Bảo vệ kernel khỏi lỗi của tiến trình.

Các lỗi phổ biến như ghi sai địa chỉ, tràn bộ đệm hoặc dereference con trỏ sai có thể khiến toàn bộ hệ thống sập nếu tiến trình được phép truy cập vào Kernel Space. Cơ chế tách biệt giúp giới hạn lỗi trong phạm vi tiến trình gây ra.

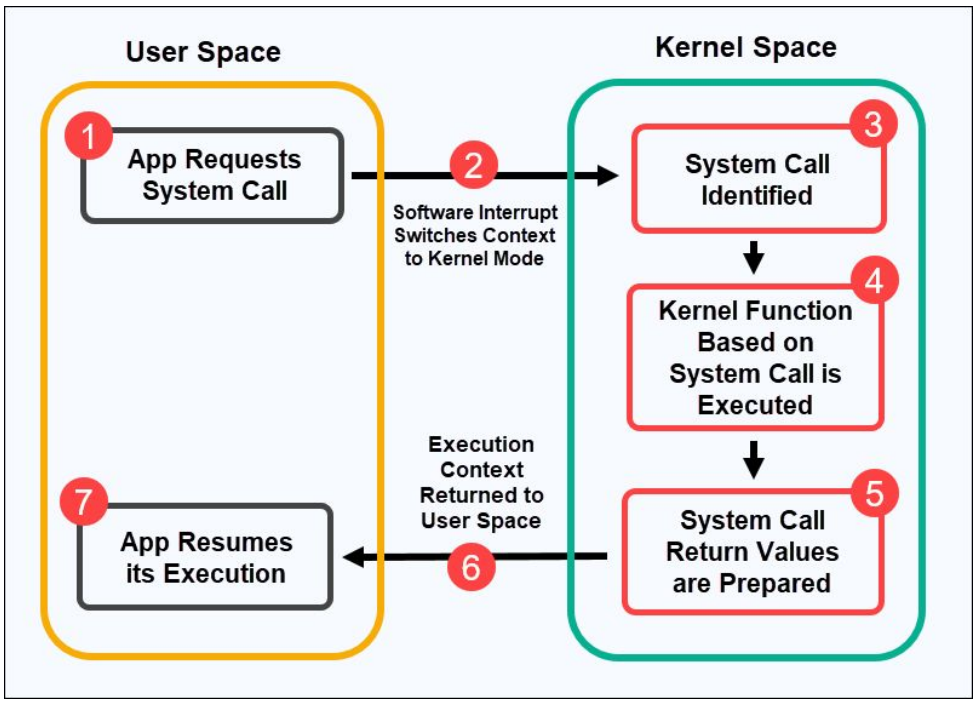
b. Ngăn chặn hành vi độc hại.

Kernel quản lý dữ liệu và tài nguyên quan trọng. Việc ngăn không cho tiến trình người dùng truy cập trực tiếp giúp tránh nguy cơ bị sửa đổi trái phép hoặc phá hoại hệ thống.

c. Kiểm soát truy cập phần cứng và tài nguyên.

Mọi thao tác liên quan đến RAM, thiết bị I/O hoặc chuyển đổi ngữ cảnh đều phải thông qua kernel để đảm bảo tính nhất quán và sự ổn định của hệ thống.

3. Cách Simple OS mô phỏng cơ chế phân tách



Hình 7: *Luồng tương tác giữa User Program, System Call và Kernel trong Simple OS*

a. User program chỉ thao tác trên địa chỉ ảo.

Mọi yêu cầu đọc hoặc ghi đều được chuyển qua cơ chế phân trang; tiến trình người dùng không thể truy cập trực tiếp các frame vật lý trong RAM.

b. Kernel kiểm soát toàn bộ bộ nhớ hệ thống.

Kernel chịu trách nhiệm quản lý RAM & SWAP, duy trì page table, xử lý ánh xạ page–frame và thực hiện hoán đổi trang khi cần thiết.

c. Các thao tác bộ nhớ được thực thi thông qua System Calls.

Khi tiến trình yêu cầu cấp phát, truy cập hoặc giải phóng bộ nhớ, nó phải gửi yêu cầu thông qua lời gọi hệ thống, ví dụ:

`SYSCALL 17` $\rightarrow$ `SYSMEM_OP`

Kernel sẽ kiểm tra tính hợp lệ của yêu cầu, xác thực quyền truy cập, thực thi thao tác và trả kết quả cho tiến trình người dùng.

Cơ chế này đảm bảo rằng toàn bộ hoạt động bộ nhớ trong User Space đều được giám sát, kiểm soát và tuân thủ chặt chẽ các nguyên tắc an toàn của hệ điều hành.

2.2.3 The Virtual Memory Mapping & Segmentation

Cơ chế quản lý bộ nhớ ảo đóng vai trò trung tâm trong các hệ điều hành hiện đại, cho phép mỗi tiến trình vận hành trong một không gian bộ nhớ độc lập, an toàn và được trừu tượng hóa hoàn toàn khỏi bộ nhớ vật lý. Trong *Simple Operating System*, mô hình này được tái hiện thông qua hai thành phần cốt lõi: *Virtual Memory Mapping* và *Segmentation*. Hai cơ chế này phối hợp nhằm bảo đảm việc cấp phát, tổ chức và bảo vệ bộ nhớ diễn ra hiệu quả trong suốt vòng đời của tiến trình.

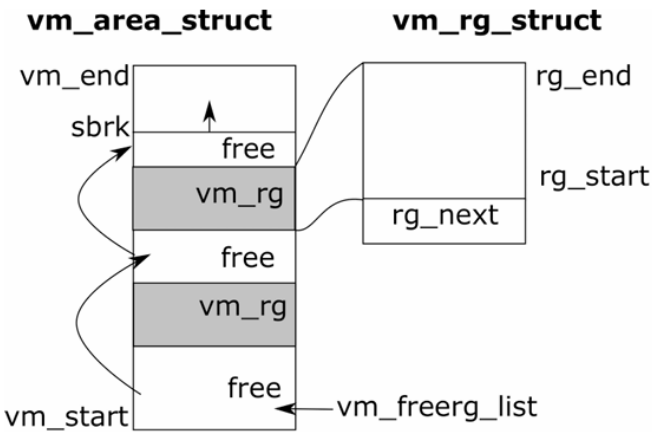
a) Virtual Memory Mapping in Each Process

Trong Simple OS, mỗi tiến trình được tạo ra cùng với một không gian địa chỉ ảo riêng biệt, được quản lý bởi cấu trúc `mm_struct`. Không gian này không được chia sẻ giữa các tiến trình và được tổ chức thành nhiều *Virtual Memory Areas (VMAs)*. Mỗi VMA đại diện cho một vùng chức năng của chương trình như mã lệnh, dữ liệu tĩnh, vùng heap động hoặc ngăn xếp.

Các VMA được tổ chức dưới dạng danh sách liên kết thông qua cấu trúc `vm_area_struct`, cho phép hệ thống:

- mở rộng hoặc thu hẹp vùng nhớ một cách linh hoạt;
- tổ chức lại không gian bộ nhớ ảo khi tiến trình thay đổi nhu cầu bộ nhớ;
- bảo đảm rằng mỗi vùng nhớ có phạm vi truy cập và giới hạn rõ ràng.

Bên trong từng VMA, bộ nhớ tiếp tục được chia thành các *memory region* được quản lý bởi cấu trúc `vm_rg_struct`. Các region này mô phỏng các thực thể dữ liệu như biến, mảng hoặc đối tượng của chương trình. Simple OS sử dụng bảng ký hiệu `symrgtbl[]` để quản lý một tập giới hạn các region nhằm đơn giản hóa mô hình bộ nhớ.



Hình 8: *Cấu trúc tổ chức của các vùng nhớ ảo (VMA) và các vùng nhớ (Memory Regions)*

Khi CPU thực hiện một truy cập bộ nhớ, địa chỉ ảo được sinh ra và được tách thành hai thành phần:

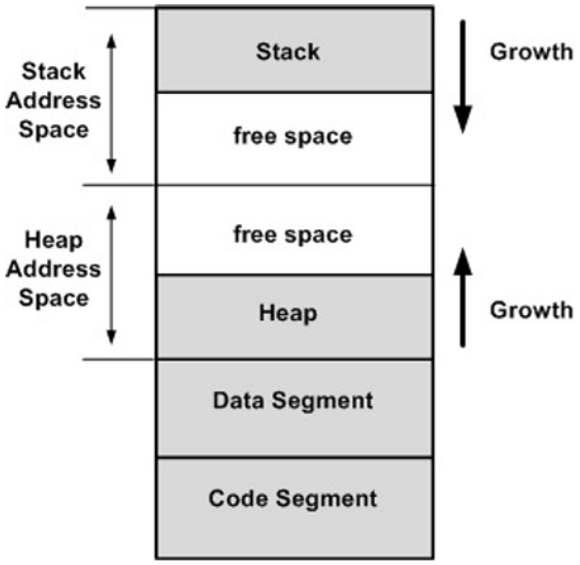
- **Page number:** chỉ số trang dùng để tra bảng trang (page table), từ đó xác định khung trang tương ứng trong bộ nhớ vật lý;
- **Page offset:** phần bù bên trong trang, biểu thị vị trí cụ thể của dữ liệu.

Bằng cách kết hợp địa chỉ khung trang và phần bù, bộ xử lý tạo ra địa chỉ vật lý, giúp hệ thống ánh xạ linh hoạt giữa bộ nhớ ảo và vật lý, duy trì không gian ảo liên tục và hỗ trợ swap khi RAM hạn chế.

b) Phân đoạn bộ nhớ (Segmentation)

Phân đoạn (*segmentation*) là kỹ thuật tổ chức bộ nhớ dựa trên các đơn vị logic độc lập gọi là *segment*. Mỗi segment phục vụ một chức năng hoặc loại dữ liệu cụ thể trong tiến trình. Các segment phổ biến gồm:

- **Code Segment**: lưu trữ mã lệnh của chương trình;
- **Data Segment**: chứa biến toàn cục và dữ liệu tĩnh;
- **Heap Segment**: phục vụ cấp phát động;
- **Stack Segment**: lưu trữ khung ngăn xếp cho lời gọi hàm và biến cục bộ.



Hình 9: *Cấu trúc các segment trong không gian bộ nhớ của tiến trình*

Mỗi segment được đặc trưng bởi hai tham số then chốt:

- **Base address**: địa chỉ bắt đầu của segment trong không gian ảo;
- **Limit**: kích thước của segment.

Nhờ mô hình này, hệ điều hành có thể kiểm soát chặt chẽ phạm vi truy cập của từng segment. Mọi thao tác vượt quá giới hạn đều bị ngăn chặn, giúp loại bỏ các lỗi như:

- tràn bộ nhớ (*buffer overflow*);
- truy cập vượt phạm vi segment;
- ghi đè dữ liệu nhạy cảm của tiến trình.

Phân đoạn còn mang lại sự linh hoạt lớn trong cấp phát, bởi các segment có thể được đặt tại các vị trí khác nhau mỗi khi tiến trình được nạp vào bộ nhớ. Điều này giúp giảm phân mảnh và tăng khả năng tận dụng tài nguyên.

Ngoài ra, segmentation còn nâng cao khả năng cô lập lỗi. Khi một segment bị khai thác hoặc hỏng, lỗi có thể được giới hạn trong phạm vi segment đó mà không lan sang các vùng khác – đặc biệt quan trọng trong môi trường đa nhiệm.

CÂU HỎI

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 phân đoạn bộ nhớ (multiple memory segments/areas) trong Simple Operating System mang lại nhiều ưu điểm quan trọng trong tổ chức không gian địa chỉ và quản lý tài nguyên của tiến trình. Các lợi ích chính có thể tóm lược như sau:

1. Tổ chức bộ nhớ có cấu trúc và dễ quản lý

Không gian bộ nhớ của tiến trình được chia thành các phân đoạn logic như code, data, heap và stack. Cách phân chia này giúp hệ thống quản lý từng vùng một cách độc lập, dễ dàng mở rộng hoặc thu hẹp khi tiến trình thay đổi nhu cầu sử dụng. Việc tách biệt các loại dữ liệu cũng hạn chế xung đột và giảm thiểu lãng phí tài nguyên.

2. Tối ưu hoá phân bổ và sử dụng bộ nhớ

Việc tách biệt từng segment cho phép hệ thống tối ưu hoá từng vùng theo đặc tính riêng:

- *Code segment*: có thể đặt ở chế độ chỉ đọc, bảo vệ an toàn mã lệnh.
- *Heap*: mở rộng linh hoạt phục vụ cấp phát động.
- *Stack*: hỗ trợ hiệu quả cơ chế lời gọi hàm và quản lý biến cục bộ.

Cách tổ chức này giảm phân mảnh bộ nhớ, tăng hiệu quả sử dụng tài nguyên và cải thiện hiệu năng tổng thể của hệ thống.

3. Ánh xạ linh hoạt vào bộ nhớ vật lý

Thiết kế nhiều phân đoạn tạo điều kiện cho việc ánh xạ rời rạc vào RAM:

- mỗi segment có thể được đặt tại các vị trí không liên tục;
- tiến trình không cần yêu cầu một vùng vật lý liền mạch;
- hệ thống tận dụng tốt các khoảng trống nhỏ trong bộ nhớ vật lý.

Điều này làm giảm phân mảnh ngoài và tối ưu khả năng sử dụng RAM của hệ thống.

4. Tăng cường bảo mật và kiểm soát truy cập

Mỗi segment gắn với một địa chỉ cơ sở và giới hạn (base, limit), kèm các quyền truy cập phù hợp. Nhờ đó, hệ thống dễ dàng phát hiện và ngăn chặn truy cập vượt vùng, hạn chế ghi sai vào vùng mã lệnh, cũng như bảo vệ dữ liệu nhạy cảm. Cơ chế này góp phần tăng tính ổn định cho tiến trình trong môi trường đa nhiệm.

5. Thuận lợi cho việc mở rộng và mô phỏng hệ điều hành hiện đại

Thiết kế này phù hợp để mô phỏng cấu trúc bộ nhớ của các hệ điều hành thực tế, đồng thời dễ dàng mở rộng lên các mô hình segmentation–paging hoặc full virtual memory. Nhờ cấu trúc rõ ràng, hệ thống cũng hỗ trợ hiệu quả cho mục đích giảng dạy và nghiên cứu.

2.2.4 Paging Mechanism

Cơ chế phân trang (*paging*) là một thành phần quan trọng trong thiết kế bộ nhớ ảo của *Simple Operating System*. Thông qua việc chia nhỏ không gian địa chỉ ảo thành các trang có kích thước cố định và ánh xạ chúng tới các khung trang trong bộ nhớ vật lý, hệ thống đạt được khả năng quản lý bộ nhớ linh hoạt, ổn định và nhất quán. Mục này trình bày bốn thành phần cơ bản của cơ chế phân trang được áp dụng trong hệ thống.

(1) Khái niệm Paging

Paging là kỹ thuật phân chia không gian địa chỉ ảo thành các trang (*pages*) có kích thước bằng nhau, cho phép ánh xạ rời rạc tới những khung trang (*frames*) trong bộ nhớ vật lý.

Cách tổ chức này mang lại nhiều lợi ích quan trọng:

- loại bỏ hoàn toàn hiện tượng phân mảnh ngoài;
- cho phép cấp phát bộ nhớ không cần liên tục trên RAM;
- bảo đảm sự cô lập không gian địa chỉ giữa các tiến trình;
- tạo điều kiện thuận lợi cho việc theo dõi, mở rộng và thu hồi vùng nhớ.

Nhờ những ưu điểm này, paging trở thành cơ chế quản lý bộ nhớ tiêu chuẩn trong hầu hết các hệ điều hành hiện đại.

(2) Cấu trúc địa chỉ ảo (Virtual Address Structure)

Trong *Simple OS*, mỗi địa chỉ ảo do CPU tạo ra được phân tách thành hai trường:

- **Page Number (p)**: xác định chỉ số trang trong không gian địa chỉ ảo và được dùng để tra cứu bảng trang;
- **Page Offset (d)**: biểu thị vị trí cụ thể bên trong trang.

Hệ thống hỗ trợ hai kích thước trang cố định là **256B** và **512B**, tùy theo cấu hình thực nghiệm.

Khi truy cập bộ nhớ, CPU sử dụng *Page Number* để tìm khung trang vật lý tương ứng trong bảng trang; *Page Offset* sau đó được kết hợp với địa chỉ khung trang để tạo ra địa chỉ vật lý cuối cùng.

(3) Page Table

Mỗi tiến trình được cấp một *page table* độc lập, được quản lý thông qua trường `pgd` trong cấu trúc `mm_struct`. Page table mô tả ánh xạ giữa các trang ảo và khung trang vật lý, đồng thời lưu trữ nhiều thông tin trạng thái của mỗi trang, bao gồm:

- **present**: trang đang nằm trong bộ nhớ RAM;
- **swapped**: trang đã được chuyển ra thiết bị hoán đổi (trình bày tại Mục 2.2.5);
- **dirty**: trang đã bị ghi dữ liệu;
- các trường mã hoá hoặc đánh dấu khác theo yêu cầu quản lý của hệ thống.

Nhờ vậy, mỗi tiến trình sở hữu một không gian địa chỉ hoàn toàn tách biệt, góp phần tăng cường an toàn và ổn định cho toàn bộ hệ thống.

(4) Multi-level Paging

Trong các kiến trúc có không gian địa chỉ lớn, bảng trang đơn cấp trở nên không hiệu quả do tiêu tốn nhiều bộ nhớ. Vì vậy, mô hình *multi-level paging* được áp dụng nhằm tổ chức page table theo dạng phân cấp, với các ưu điểm:

- giảm đáng kể dung lượng bộ nhớ dành cho bảng trang;
- chỉ khởi tạo các bảng con cho những vùng địa chỉ thực sự được sử dụng;
- tăng khả năng mở rộng và tối ưu hóa cấu trúc lưu trữ bộ nhớ.

Ví dụ:

- kiến trúc **32-bit** thường sử dụng phân trang hai cấp;
- kiến trúc **64-bit** mở rộng thành bốn hoặc năm cấp: PGD → P4D → PUD → PMD → PT.

2.2.5 The system physical memory

Trong mô hình của *Simple Operating System*, mỗi tiến trình sở hữu một không gian địa chỉ ảo riêng biệt, song toàn bộ các ánh xạ từ không gian ảo cuối cùng đều hướng về một kiến trúc bộ nhớ vật lý thống nhất của hệ thống. Kiến trúc này bao gồm hai thành phần trọng yếu: **RAM** (bộ nhớ chính) và **SWAP** (bộ nhớ thứ cấp). Sự phối hợp giữa hai loại bộ nhớ này cho phép hệ thống duy trì hoạt động ổn định ngay cả trong điều kiện tài nguyên hạn chế.

(1) RAM - Bộ nhớ chính được CPU truy cập trực tiếp

RAM là thành phần trung tâm của hệ thống bộ nhớ vật lý. Đây là vùng mà CPU có thể truy cập trực tiếp thông qua bus địa chỉ, cho phép thực thi các lệnh đọc/ghi tức thời với độ trễ tối thiểu. Mọi tiến trình muốn được thực thi đều phải có các trang của mình hiện diện trong RAM.

Các đặc điểm quan trọng của RAM:

- là nơi lưu trữ các trang đang hoạt động của tiến trình với tốc độ truy xuất cao;
- hỗ trợ thực thi trực tiếp từ CPU mà không cần qua trung gian;
- dung lượng hạn chế, đặc biệt trong các hệ thống nhúng hoặc cấu hình tối giản.

Do đó, RAM là tài nguyên cần được quản lý tinh vi thông qua cơ chế phân trang để bảo đảm tính ổn định và hiệu năng.

(2) SWAP - Bộ nhớ thứ cấp mở rộng dung lượng cho hệ thống

SWAP đóng vai trò như một vùng bộ nhớ phụ, có nhiệm vụ lưu trữ các trang không đủ chỗ nằm trong RAM. Khác với RAM, SWAP **không hỗ trợ truy cập trực tiếp từ CPU**; mọi dữ liệu trong SWAP muốn được sử dụng đều phải được đưa trở lại RAM.

Những đặc trưng chính của SWAP:

- dung lượng lớn và chi phí thấp, phù hợp để dự trữ các trang ít sử dụng;
- không hỗ trợ truy cập ngẫu nhiên từ CPU, chỉ có thể thao tác thông qua trao đổi dữ liệu với RAM;
- cho phép hệ thống vận hành các tiến trình có kích thước vượt quá dung lượng RAM vật lý.

SWAP giúp hệ điều hành vượt qua giới hạn vật lý của RAM, giữ cho hệ thống hoạt động ổn định trong môi trường tải nặng.

(3) Mối quan hệ RAM–SWAP trong quản lý bộ nhớ

Khi RAM không còn đủ khung trang trống để cấp phát cho tiến trình, hệ điều hành thực hiện chu trình hoán đổi trang (*swapping*). Chu trình này gồm hai thao tác cơ bản:

- **Swap-out**: các trang ít được truy cập bị đẩy từ RAM xuống SWAP nhằm giải phóng khung trang;
- **Swap-in**: khi một trang bị swap-out được truy cập lại, hệ thống phải nạp trang đó trở lại RAM.

Cơ chế hoán đổi giúp RAM luôn duy trì đủ frame rỗi cho các tiến trình đang hoạt động. Hệ thống có thể xử lý tiến trình lớn hơn nhiều so với dung lượng RAM thực tế, đồng thời tối ưu tái sử dụng bộ nhớ vật lý. RAM thực thi trực tiếp, còn SWAP lưu trữ tạm thời, phối hợp hoạt động liên tục và ổn định.

CÂU HỎI

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

Trả lời:

Khi địa chỉ được chia thành nhiều hơn 2 cấp trong hệ thống phân trang, hệ thống sẽ sử dụng **bảng trang nhiều cấp (multi-level paging)**. Địa chỉ ảo được xử lý qua từng cấp: từ cấp trên xác định vùng địa chỉ của cấp dưới, cho đến khi xác định được địa chỉ vật lý trong bộ nhớ.

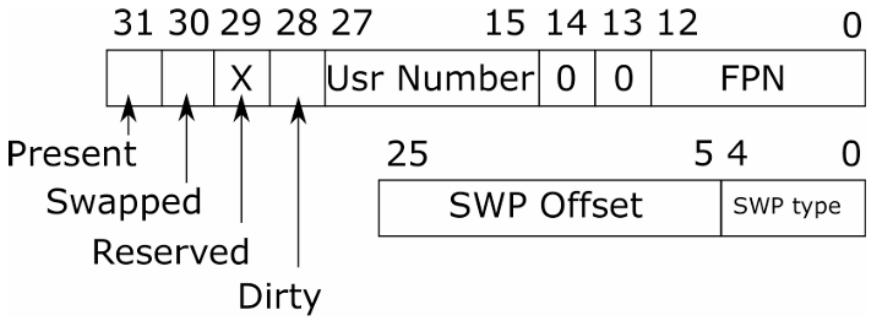
- **Ưu điểm:**
  - **Tiết kiệm bộ nhớ:** Phân trang nhiều cấp giúp giảm lượng bộ nhớ cần thiết cho bảng trang, vì mỗi cấp chỉ chứa một số mục nhỏ hơn. Do đó, tổng bộ nhớ dùng để lưu bảng trang giảm, tối ưu hóa việc sử dụng tài nguyên.
  - **Cải thiện hiệu suất tra cứu:** Số lượng mục trong mỗi cấp nhỏ hơn so với bảng trang đơn cấp, giúp rút ngắn thời gian tra cứu địa chỉ và nâng cao tốc độ truy xuất bộ nhớ. Việc này đặc biệt quan trọng trong các hệ thống cần đáp ứng nhanh các truy cập bộ nhớ.
  - **Linh hoạt và mở rộng:** Phân trang nhiều cấp cho phép tổ chức không gian bộ nhớ linh hoạt hơn. Điều này đặc biệt hữu ích với các hệ thống có yêu cầu bộ nhớ thay đổi, vì bảng trang có thể được điều chỉnh để phù hợp với nhu cầu của từng tiến trình.
- **Nhược điểm:**
  - **Tăng độ phức tạp quản lý:** Việc triển khai phân trang nhiều cấp làm hệ thống quản lý bộ nhớ trở nên phức tạp hơn, khiến việc thiết kế, triển khai và gỡ lỗi khó khăn hơn.
  - **Tăng chi phí truy cập:** Mặc dù giảm dung lượng bộ nhớ những mỗi lần tra cứu địa chỉ ảo sang địa chỉ vật lý phải đi qua nhiều cấp bảng trang, làm tăng thời gian truy xuất bộ nhớ. Điều này có thể ảnh hưởng đến hiệu suất, đặc biệt trong các hệ thống yêu cầu tốc độ xử lý cao.
  - **Nguy cơ phân mảnh bộ nhớ:** Các mục bảng trang không liên tục có thể dẫn đến phân mảnh bộ nhớ, làm giảm hiệu năng tổng thể và tăng chi phí truy cập khi hệ thống cần quản lý nhiều tiến trình đồng thời.

2.2.6 Paging-based address translation scheme

Cơ chế dịch địa chỉ dựa trên phân trang (paging-based address translation) là thành phần cốt lõi của hệ thống bộ nhớ ảo trong *Simple Operating System*. Cơ chế này cho phép ánh xạ linh hoạt giữa không gian địa chỉ ảo của tiến trình và bộ nhớ vật lý, bảo đảm mỗi tiến trình quan sát một không gian liên tục và ổn định, ngay cả khi dữ liệu thực tế được phân bố rời rạc trong RAM hoặc được chuyển sang SWAP khi thiếu bộ nhớ chính.

1. Page Table Entry (PTE) Structure

Mỗi tiến trình duy trì một bảng trang độc lập dùng để ánh xạ từng trang ảo sang một khung trang vật lý. Mỗi mục PTE được biểu diễn bằng một trường 32-bit, mã hóa đầy đủ trạng thái và vị trí hiện tại của trang.



Hình 10: *Page Table Entry Format (32-bit) in Simple OS*

Ý nghĩa các trường trong PTE:

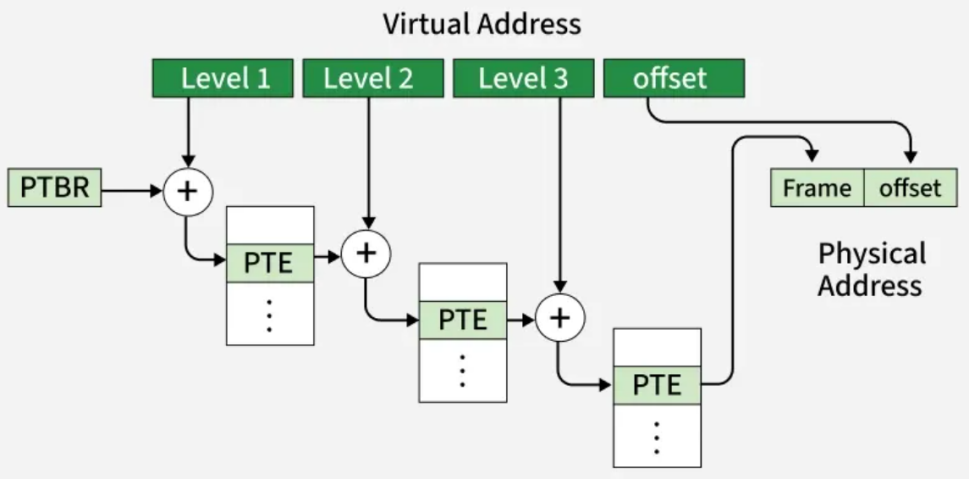
- **Bits 0–12:** Frame Number (FPN) – chỉ số khung trang trong RAM khi trang hiện diện.
- **Bits 13–14:** Reserved – đặt bằng 0 khi trang nằm trong RAM.
- **Bits 15–27:** User-defined metadata – lưu thông tin bổ trợ.
- **Bits 0–4 (swap mode):** Swap Type – loại cấu trúc lưu trữ trong SWAP.
- **Bits 5–25 (swap mode):** Swap Offset – vị trí trang trong thiết bị SWAP.
- **Bit 28:** Dirty bit – báo hiệu trang đã bị ghi.
- **Bit 30:** Swapped – trang đang nằm ngoài RAM.
- **Bit 31:** Present – trang có đang nằm trong RAM hay không.

Nhờ cấu trúc mã hóa này, kernel có thể quản lý đầy đủ chu kỳ sống của trang, hỗ trợ hiệu quả việc cấp phát, truy cập và hoán đổi.

2. Virtual-to-Physical Address Translation Pipeline

Khi CPU thực hiện một truy cập bộ nhớ, nó sinh ra một địa chỉ ảo (virtual address). Địa chỉ này được phân tách thành:

- **VPN (Virtual Page Number):** chỉ số trang trong bảng trang.
- **Offset:** vị trí cụ thể bên trong trang.



Hình 11: *Minh họa quy trình dịch địa chỉ ảo sang địa chỉ vật lý trong cơ chế phân trang nhiều cấp.*

Quá trình dịch địa chỉ trong hệ thống phân trang của Simple OS được thực hiện qua các bước sau:

1. **Tra cứu bảng trang:** Kernel sử dụng Virtual Page Number (VPN) để truy cập vào mục Page Table Entry (PTE) tương ứng, từ đó xác định trạng thái hiện tại của trang (present, swapped, hoặc chưa cấp phát).
2. **Trang hiện diện trong RAM (present = 1):**
   - Trích xuất số khung trang (Frame Number – FPN) từ PTE.
   - Kết hợp FPN với phần bù (offset) để tạo ra địa chỉ vật lý hoàn chỉnh.
   - CPU truy cập trực tiếp vào RAM thông qua địa chỉ vật lý vừa được tính toán.
3. **Trang nằm trong SWAP (present = 0, swapped = 1):**
   - Kernel thực hiện *swap-in*, nạp trang từ thiết bị SWAP vào một khung trang trống trong RAM.
   - Nếu RAM không còn frame trống, hệ thống lựa chọn một trang nạn nhân và thực hiện *swap-out* để giải phóng không gian.
   - PTE được cập nhật lại trạng thái (present, swapped, frame number), sau đó kernel tiếp tục xử lý lại yêu cầu dịch địa chỉ.
4. **Trang chưa được cấp phát:**
   - Kernel cấp phát một trang mới trong RAM theo cơ chế on-demand.
   - Khởi tạo nội dung trang và cập nhật PTE tương ứng trước khi tiếp tục xử lý truy cập.

3. Interactions Among Memory Subsystems

Cơ chế dịch địa chỉ là kết quả của sự phối hợp giữa nhiều thành phần trong hệ thống quản lý bộ nhớ. Các thành phần chính bao gồm:

- **mm_struct:** quản lý tổng thể không gian địa chỉ ảo và con trỏ tới bảng trang.
- **vm_area_struct:** mô tả các vùng bộ nhớ ảo như code, data, heap, stack.
- **Page Table:** lưu ánh xạ trang–khung.
- **memphy_struct:** mô phỏng RAM và thao tác đọc/ghi.

- **memswp_struct:** mô phỏng vùng SWAP và vị trí lưu trữ trang.
- **libmem:** triển khai các hàm `alloc`, `free`, `read`, `write` và kết nối đến các system call của kernel.

Sự kết hợp giữa các thành phần này cho phép Simple OS mô phỏng gần đúng hành vi của các hệ điều hành hiện đại, đồng thời duy trì kiến trúc đơn giản giúp việc nghiên cứu và thực nghiệm trở nên thuận tiện.

CÂU HỎI

What are the advantages and disadvantages of segmentation with paging?

Trả lời:

Segmentation kết hợp với paging là một cơ chế quản lý bộ nhớ lai, tận dụng ưu điểm của phân đoạn (segmentation) để tổ chức bộ nhớ theo cấu trúc logic của chương trình, và ưu điểm của phân trang (paging) để loại bỏ phân mảnh ngoại và hỗ trợ cấp phát linh hoạt. Nhờ đó, hệ thống có thể quản lý các loại dữ liệu tĩnh và động một cách hiệu quả, đồng thời duy trì khả năng truy cập ổn định.

- **Ưu điểm:**
  - **Quản lý bộ nhớ logic và rõ ràng:** Các phân đoạn được tạo ra theo chức năng của chương trình, chẳng hạn data segment chứa các biến toàn cục/tĩnh, còn heap segment phục vụ cấp phát động. Điều này giúp hệ điều hành dễ dàng phân biệt và quản lý các loại dữ liệu khác nhau.
  - **Giảm phân mảnh bộ nhớ:** Việc áp dụng paging bên trong từng segment giúp hạn chế đáng kể phân mảnh nội và ngoại, nhờ kích thước trang cố định và khả năng cấp phát không liên tục.
  - **Linh hoạt trong cấp phát:** Biến trong data segment tồn tại suốt thời gian chạy chương trình, trong khi dữ liệu trong heap có thể được cấp phát và giải phóng tùy theo nhu cầu. Mô hình kết hợp cho phép cả hai hoạt động diễn ra trơn tru mà không gây xung đột vùng nhớ.
  - **Dễ dàng truy cập và bảo trì:** Việc tách bạch giữa biến tĩnh và biến động giúp hệ điều hành đơn giản hóa thao tác cấp phát, thu hồi và truy cập bộ nhớ, giảm lỗi và cải thiện tính ổn định của chương trình.
- **Nhược điểm:**
  - **Tăng độ phức tạp quản lý:** Hệ điều hành phải quản lý đồng thời cả segment và bảng trang trong mỗi segment, dẫn đến yêu cầu thuật toán và cấu trúc dữ liệu phức tạp hơn.
  - **Chi phí truy cập cao hơn:** Mỗi lần truy cập bộ nhớ phải trải qua hai bước: xác định segment trước, sau đó mới tra cứu bảng trang tương ứng. Điều này khiến độ trễ truy cập lớn hơn so với paging đơn thuần.
  - **Yêu cầu hỗ trợ phần cứng hoặc phần mềm bổ sung:** Do cơ chế dịch địa chỉ trong mô hình lai phức tạp hơn, hệ thống có thể cần bộ MMU hoặc cơ chế xử lý bổ sung để đảm bảo hoạt động chính xác.

2.3 Tổng hợp và Đồng bộ hóa (Synchronization)

CÂU HỎI

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

Trả lời:

Nếu *đồng bộ hóa* không được xử lý đúng trong một Simple OS chạy nhiều tiến trình đồng thời, sẽ phát sinh các vấn đề nghiêm trọng do các tiến trình **cùng truy cập tài nguyên chia sẻ**. Các vấn đề chính bao gồm **race condition, hỏng dữ liệu, mất cập nhật, deadlock** và kết quả hệ thống không nhất quán. Khi không có cơ chế đồng bộ, hệ điều hành không thể đảm bảo rằng tài nguyên chia sẻ được truy cập một cách an toàn và nhất quán.

Các vấn đề chính khi thiếu đồng bộ hóa:

- **Data race (tranh chấp dữ liệu):** Xảy ra khi hai hoặc nhiều tiến trình cùng truy cập một biến chia sẻ, ít nhất một tiến trình thực hiện ghi. Điều này dẫn đến dữ liệu không chính xác hoặc không nhất quán.
- **Race condition (điều kiện tranh chấp):** Xảy ra khi kết quả của chương trình phụ thuộc vào thứ tự hoặc thời điểm thực thi của các tiến trình. Thay đổi nhỏ trong thứ tự thực thi có thể dẫn đến kết quả không như mong đợi.
- **Deadlock (tắc nghẽn):** Xảy ra khi hai hoặc nhiều tiến trình chờ lẫn nhau để truy cập tài nguyên, tạo thành vòng phụ thuộc và không tiến triển được.
- **Mất cập nhật (Lost updates):** Khi nhiều tiến trình cập nhật cùng một biến hoặc bộ đếm chung mà không sử dụng khóa, một số thao tác có thể bị ghi đè hoặc bỏ sót, dẫn đến kết quả sai.

Ví dụ trong Simple OS:
Giả sử có một biến đếm chung biểu diễn số lượng tài nguyên sẵn có:

- Tiến trình A đọc giá trị biến đếm là ‘10‘.
- Tiến trình B cũng đọc giá trị biến đếm là ‘10‘ cùng lúc.
- Tiến trình A tăng biến lên ‘11‘ và ghi lại.
- Tiến trình B tăng biến lên ‘11‘ và ghi lại.

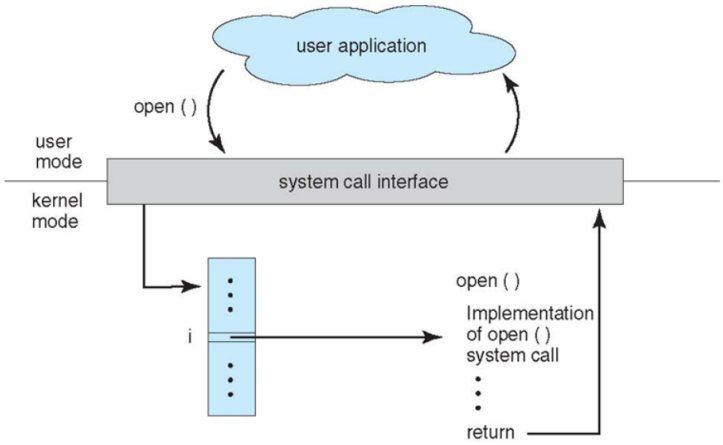
Kết quả cuối cùng là ‘11‘, trong khi lẽ ra phải là ‘12‘. Đây là mất cập nhật (lost update) do race condition.

Kết luận:
Để đảm bảo tiến trình truy cập tài nguyên chia sẻ an toàn, nhất quán và dự đoán được, hệ điều hành cần sử dụng cơ chế đồng bộ hóa, như **mutex, semaphore hoặc spinlock**. Thiếu đồng bộ hóa sẽ khiến hệ thống hoạt động không dự đoán được, kết quả sai hoặc không ổn định.

2.4 System Calls (Giao tiếp hệ thống)

2.4.1 Kiến trúc và cơ chế hoạt động của System Call

System call là cơ chế cho phép chương trình người dùng tương tác với các chức năng đặc quyền của hệ điều hành. Đây là lớp giao tiếp tiêu chuẩn giữa *userspace* và *kernelspace*, đảm bảo mọi thao tác truy cập tài nguyên đều được kiểm soát và xác thực bởi kernel. System call giúp duy trì tính an toàn, tách biệt quyền hạn và sự ổn định hệ thống.



Hình 12: *Quy trình xử lý một system call từ userspace đến kernelspace.*

Về mặt lý thuyết, một system call được xử lý theo chu trình sau:

1. Ứng dụng gọi hàm *wrapper* từ thư viện runtime; hàm này chuẩn bị tham số và syscall index.
2. Tham số được ghi vào các thanh ghi (`a1`, `a2`, `a3`,...) theo quy ước gọi hệ thống.
3. CPU thực thi lệnh *syscall/trap* và chuyển sang *kernel mode*.
4. Kernel tra cứu bảng `syscall_tbl` để xác định handler tương ứng.
5. Hàm xử lý `__sys_xxxhandler()` được thực thi để xử lý logic của system call.
6. Kernel ghi kết quả trả về vào thanh ghi và chuyển CPU trở lại *user mode*.

2.4.2 Quy trình thêm System Call

Việc bổ sung một system call mới gồm ba bước chính:

Bước 1 - Tạo handler trong kernel

```
int __sys_xxxhandler(struct pcb_t *caller, struct sc_regs *regs) {

    return 0;
    }
```

Bước 2 - Thêm vào syscall.tbl

```
440     xxx     sys_xxxhandler
```

Bước 3 - Thêm vào Makefile

```
SYSCALL_OBJ += $(OBJ)/sys_xxxhandler.o
```

Sau đó biên dịch lại kernel (`make all`) và viết chương trình test để gọi system call mới.

2.4.3 Các System Call mới

Hai system call mới được bổ sung:

- **listsyscall(syscall 0)**: Liệt kê toàn bộ system call trong kernel.
- **memmap(syscall 17)**: Hỗ trợ các thao tác quản lý bộ nhớ như:
  - tăng giới hạn VMA (`SYSMEM_INC_OP`);
  - hoán đổi trang (`SYSMEM_SWP_OP`);
  - đọc/ghi bộ nhớ vật lý (`SYSMEM_IO_READ/WRITE`).
- **meminc(syscall 18)**: Tăng giới hạn VMA (bản module hóa)
- **memswp(syscall 19)**: Hoán đổi trang(bản module hóa)

Các system call này góp phần hoàn chỉnh cơ chế quản lý bộ nhớ và thể hiện khả năng mở rộng linh hoạt của Simple OS.

- **Lưu ý:** Tuy việc sử dụng syscall 17 hay syscall 18,19 cho các thao tác tăng giới hạn và hoán đổi trang không có sự khác biệt nhiều về mặt logic và đầu ra, nhưng nhóm vẫn quyết định giữ 2 file này. Bởi vì việc này chứng minh nhóm đã có các khả năng thao tác, chỉnh sửa danh sách syscall.Hơn nữa, còn giúp đơn giản hóa mã nguồn, dễ sửa lỗi và dễ dàng phát triển thêm trong tương lai, có thể dễ dàng sửa đổi tính năng, điều chỉnh logic mà không ảnh hưởng đến toàn bộ hệ thống. Chính vì nhưng lí do trên, nhóm quyết định giữ nguyên cấu trúc này.

2.4.4 Trả lời câu hỏi (Questionnaire)

CÂU HỎI 1

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

Trả lời:

Trong các hệ điều hành như **Simple OS**, khi một **system call (syscall)** được gọi từ **user space** lên **kernel space**, CPU chỉ có một số lượng **thanh ghi (registers) hạn chế** để truyền tham số. Tuy nhiên, nhiều syscall cần truyền các tham số phức tạp như **struct, mảng hoặc con trỏ tới vùng nhớ**, đòi hỏi các cơ chế đặc biệt để vượt qua giới hạn này.

Các cơ chế phổ biến bao gồm:

- **Sử dụng con trỏ tới vùng nhớ (Pointer to memory):** User space chỉ truyền địa chỉ bộ nhớ chứa dữ liệu cho kernel. Kernel sẽ đọc dữ liệu từ địa chỉ này khi thực thi syscall, cho phép truyền các dữ liệu lớn hoặc phức tạp mà không cần nhiều thanh ghi.
- **Truyền tham số qua stack:** Các tham số phức tạp được lưu trên **stack** của tiến trình. Kernel truy xuất stack để lấy tham số khi thực hiện syscall, hữu ích khi tham số vượt quá khả năng chứa của thanh ghi.
- **Đóng gói tham số thành struct (Argument struct):** Gom tất cả tham số vào một **struct duy nhất**, sau đó truyền con trỏ tới struct qua thanh ghi, giúp tiết kiệm thanh ghi và quản lý tham số tập trung.
- **Truyền dữ liệu theo block kèm kích thước (Buffer block with size):** Khi truyền dữ liệu lớn, user space truyền con trỏ tới buffer và kích thước dữ liệu. Kernel đọc dữ liệu dựa trên con trỏ và size, đảm bảo thao tác an toàn và hiệu quả.

Tóm lại, việc truyền tham số phức tạp cho syscall trong môi trường bị giới hạn thanh ghi dựa chủ yếu vào cơ chế **truyền con trỏ tới dữ liệu trong bộ nhớ hoặc struct gom tham số**, giúp syscall linh hoạt, an toàn và có thể xử lý dữ liệu lớn hoặc phức tạp mà không làm tăng áp lực lên thanh ghi.

CÂU HỎI 2

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

Trả lời:

Khi một system call (syscall) trong Simple OS mất quá nhiều thời gian để thực thi, hệ thống sẽ gặp phải các vấn đề nghiêm trọng về hiệu năng, khả năng phản hồi và tính ổn định:

- **Chặn CPU (CPU Blocking):** Syscall chạy trong **kernel mode** có thể chiếm toàn bộ CPU, làm các tiến trình khác phải chờ và giảm hiệu quả sử dụng CPU.
- **Tăng độ trễ cho các tiến trình khác:** Tiến trình trong **ready queue** sẽ bị trì hoãn, gây giảm khả năng phản hồi của hệ thống, đặc biệt với các ứng dụng tương tác thời gian thực.
- **Nguy cơ starvation:** Nếu syscall chiếm CPU lâu, các tiến trình **ưu tiên thấp** có thể bị đợi quá lâu, dẫn đến mất công bằng trong phân bổ tài nguyên.
- **Tranh chấp tài nguyên và trạng thái không nhất quán:** Khi thao tác trên tài nguyên chia sẻ mà không có cơ chế đồng bộ, syscall kéo dài sẽ làm tăng nguy cơ **race condition** hoặc trạng thái hệ thống không ổn định.
- **Timeout hoặc deadlock:** Việc giữ khóa hoặc tài nguyên quá lâu có thể gây **timeout** hoặc **deadlock**, khiến hệ thống ngưng trệ hoặc treo tiến trình.

Các biện pháp khắc phục và giảm thiểu:

- Chia nhỏ công việc, sử dụng các thao tác **non-blocking** hoặc **asynchronous** để tránh chặn CPU lâu.
- Cho phép tiến trình **yield CPU** định kỳ nhằm đảm bảo các tiến trình khác có cơ hội thực thi.
- Sử dụng **kernel thread** hoặc **deferred work** cho các tác vụ nặng để giảm áp lực lên tiến trình chính.
- Thiết lập **timeout** cho syscall để hạn chế thời gian thực thi và bảo vệ hiệu năng hệ thống.

Kết luận:

Syscall thực thi quá lâu sẽ **giảm khả năng phản hồi, tăng độ trễ, gây tranh chấp tài nguyên và nguy cơ deadlock**. Vì vậy, khi thiết kế syscall, cần đảm bảo nó **hiệu quả, không chiếm CPU lâu và an toàn trong môi trường đa tiến trình**.

3 Hiện Thực và Đánh Giá

3.1 Hiện thực Bộ Lập Lịch

3.1.1 Cấu trúc hàng đợi

Hàng đợi (*queue*) là cấu trúc dữ liệu nền tảng được sử dụng để quản lý và tổ chức các tiến trình trong bộ lập lịch. Trong hệ thống đề xuất, hàng đợi được định nghĩa thông qua cấu trúc `queue_t` trong file `queue.h` như sau:

```
struct queue_t {
    struct pcb_t *proc[MAX_QUEUE_SIZE];
    int size;
};
```

Ý nghĩa các thuộc tính:

- `proc[]`: Mảng lưu trữ các con trỏ đến *Process Control Block* (PCB), mỗi phần tử tương ứng với một tiến trình trong hàng đợi.
- `size`: Biểu thị số lượng tiến trình hiện đang tồn tại trong hàng đợi.

Cấu trúc `queue_t` được hiện thực theo mô hình *array-based queue*, với các đặc điểm chính như sau:

- Kích thước cố định, được xác định trước bởi hằng số `MAX_QUEUE_SIZE`.
- Các tiến trình được truy cập theo nguyên tắc FIFO (*First In First Out*).
- Cách cài đặt đơn giản, dễ kiểm tra, dễ theo dõi và thuận lợi cho việc debug trong quá trình mô phỏng bộ lập lịch.

3.1.2 Các thao tác trên hàng đợi (Queue Operations)

Hệ thống hiện thực ba thao tác cơ bản trên hàng đợi trong file `queue.c`, bao gồm `enqueue`, `dequeue` và `purgequeue`. Trong phạm vi mục này, báo cáo tập trung phân tích hai thao tác cốt lõi là `enqueue()` và `dequeue()`.

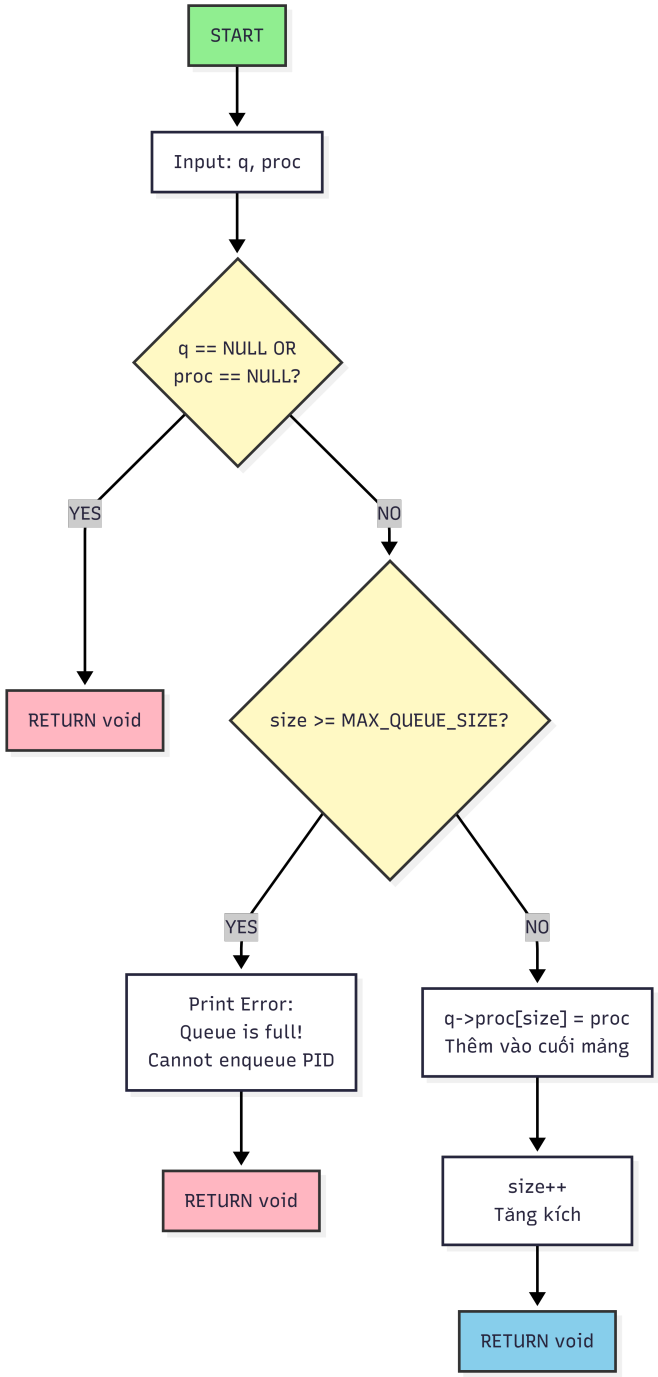
A. Thao tác ENQUEUE - Thêm tiến trình vào hàng đợi

Mô tả: Hàm `enqueue()` có nhiệm vụ thêm một tiến trình mới vào cuối hàng đợi, tuân thủ nguyên tắc FIFO. Thao tác này được sử dụng khi một tiến trình mới được tạo ra hoặc khi tiến trình quay trở lại trạng thái sẵn sàng (*ready state*).

```
void enqueue(struct queue_t *q, struct pcb_t *proc)
{
    if (q == NULL || proc == NULL)
        return;
    if (q->size >= MAX_QUEUE_SIZE) {
        printf("ERROR: Queue is full! Cannot enqueue PID=%d\n", proc
   ->pid);
```

```
        return;
    }
    q->proc[q->size] = proc;
    q->size++;
}
```

Flowchart hàm enqueue():



Hình 13: *Lưu đồ xử lý của hàm enqueue()*

Ví dụ minh họa:
Giả sử hàng đợi có kích thước tối đa `MAX_QUEUE_SIZE = 5`. Tại thời điểm ban đầu, hàng đợi đang chứa hai tiến trình `P1` và `P2`.

Trạng thái ban đầu của hàng đợi:

```
Index:  [0]   [1]   [2]   [3]   [4]
Proc :  [P1]  [P2]  [ ]   [ ]   [ ]
size = 2
```

Với giá trị `size = 2`, vị trí chèn tiếp theo của hàng đợi là `proc[2]`.
Thực hiện thao tác:

```
enqueue(q, P3)
```

Trạng thái hàng đợi sau khi enqueue:

```
Index:  [0]   [1]   [2]   [3]   [4]
Proc :  [P1]  [P2]  [P3]  [ ]   [ ]
size = 3
```

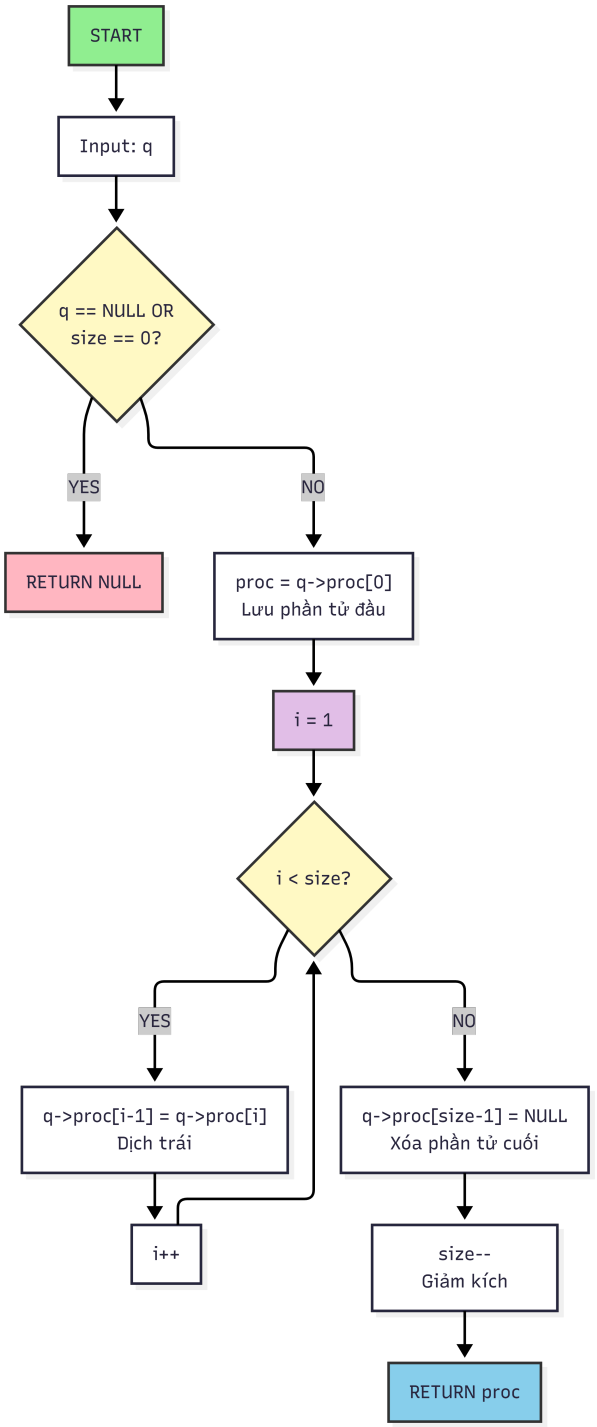
Nhận xét: Tiến trình `P3` được thêm trực tiếp vào cuối hàng đợi tại vị trí `proc[size]` trước đó. Không có phần tử nào bị dịch chuyển, do đó thao tác `enqueue()` có độ phức tạp thời gian là $O(1)$ và tuân thủ đúng nguyên tắc FIFO.

B. Thao tác DEQUEUE - Lấy tiến trình ra khỏi hàng đợi

Mô tả: Hàm `dequeue()` thực hiện việc lấy tiến trình ở đầu hàng đợi (vị trí chỉ số 0). Sau khi phần tử đầu tiên được lấy ra, các phần tử còn lại sẽ được dịch chuyển sang trái nhằm duy trì tính liên tục của mảng.

```
struct pcb_t *dequeue(struct queue_t *q)
{
    if (q == NULL || q->size == 0)
        return NULL;

    struct pcb_t *proc = q->proc[0];

    for (int i = 1; i < q->size; i++) {
        q->proc[i - 1] = q->proc[i];
    }
    q->proc[q->size - 1] = NULL;
    q->size--;

    return proc;
}
```

Flowchart hàm dequeue():



Hình 14: *Lưu đồ xử lý của hàm dequeue()*

Ví dụ minh họa:

Giả sử hàng đợi có kích thước tối đa `MAX_QUEUE_SIZE = 5`. Tại thời điểm ban đầu, hàng đợi đang chứa bốn tiến trình `P1`, `P2`, `P3` và `P4`.

Trạng thái ban đầu của hàng đợi:

```
Index:  [0]   [1]   [2]   [3]   [4]
```

```
Proc :  [P1]  [P2]  [P3]  [P4]  [  ]
size = 4
```

Khi gọi hàm `dequeue()`, tiến trình ở đầu hàng đợi (vị trí `proc[0]`) sẽ được lấy ra để xử lý.

Thực hiện thao tác:

```
dequeue(q)
```

Trạng thái hàng đợi sau khi dequeue:

```
Index:  [0]   [1]   [2]   [3]   [4]
Proc :  [P2]  [P3]  [P4]  [  ]  [  ]
size = 3
```

Nhận xét: Tiến trình `P1` được lấy ra khỏi hàng đợi, các tiến trình còn lại được dịch chuyển sang trái nhằm duy trì tính liên tục của mảng. Do thao tác dịch chuyển $n-1$ phần tử, hàm `dequeue()` có độ phức tạp thời gian là $O(n)$ và vẫn đảm bảo đúng nguyên tắc FIFO.

3.1.3 Cấu trúc MLQ Scheduler

Multi-Level Queue (MLQ) Scheduler là một thuật toán lập lịch phân cấp, trong đó các tiến trình được tổ chức vào nhiều hàng đợi khác nhau dựa trên mức độ ưu tiên. Mỗi hàng đợi đại diện cho một cấp *priority* riêng biệt và được cấp một hạn mức sử dụng CPU (*quota*) nhằm cân bằng giữa hiệu năng và tính công bằng của hệ thống.

Trong hệ thống đề xuất, MLQ Scheduler được hiện thực trực tiếp trong file `sched.c` thông qua các cấu trúc dữ liệu sau:

```
#ifdef MLQ_SCHED
struct queue_t mlq_ready_queue[MAX_PRIO];
static int slot[MAX_PRIO];
#endif
```

Thiết kế này cho phép bộ lập lịch quản lý đồng thời nhiều mức ưu tiên, đồng thời duy trì tính linh hoạt trong việc phân phối tài nguyên CPU.

A. Mảng hàng đợi ưu tiên `mlq_ready_queue[]`

`mlq_ready_queue` là mảng gồm `MAX_PRIO = 140` hàng đợi, mỗi hàng đợi tương ứng với một mức ưu tiên, trong đó giá trị priority càng nhỏ thì mức độ ưu tiên càng cao (priority 0 là cao nhất).

Mỗi tiến trình được đưa vào hàng đợi tương ứng thông qua thuộc tính `proc->prio`. Bộ lập lịch hoạt động theo hai nguyên tắc:

- **Ưu tiên theo priority:** Luôn chọn tiến trình thuộc hàng đợi có mức ưu tiên cao nhất còn sẵn sàng.
- **Round-Robin nội bộ:** Các tiến trình trong cùng một hàng đợi được xử lý luân phiên.

Thiết kế này đảm bảo khả năng đáp ứng nhanh cho các tiến trình quan trọng, đồng thời duy trì tính công bằng giữa các tiến trình có cùng mức ưu tiên.

B. Cơ chế phân bổ CPU theo quota - Mảng `slot[]`

MLQ Scheduler sử dụng mảng `slot[]` để giới hạn số slot CPU tối đa mà mỗi mức priority được phép sử dụng liên tiếp. Quota CPU cho từng mức ưu tiên được xác định theo công thức:

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

Theo cơ chế này:

- Priority cao được cấp nhiều slot CPU hơn.
- Priority thấp vẫn được đảm bảo tối thiểu một slot để tránh starvation.

Giá trị `slot[i]` là hằng số trong suốt thời gian chạy; số slot đã sử dụng tại mỗi mức priority được theo dõi riêng thông qua biến `used_slot`.

C. Khởi tạo MLQ Scheduler

Quá trình khởi tạo bộ lập lịch được thực hiện trong hàm `init_scheduler()`:

```
void init_scheduler(void)
{
    for (int i = 0; i < MAX_PRIO; i++) {
        mlq_ready_queue[i].size = 0;
        slot[i] = MAX_PRIO - i;
    }
    pthread_mutex_init(&queue_lock, NULL);
}
```

D. Thuật toán lựa chọn tiến trình - Hàm `get_mlq_proc()`

Hàm `get_mlq_proc()` là thành phần trung tâm của MLQ Scheduler, chịu trách nhiệm lựa chọn tiến trình tiếp theo để cấp CPU.

1. Quản lý trạng thái lập lịch

```
static int cur_prio = 0;
static int used_slot = 0;
```

Hai biến `static` này cho phép scheduler duy trì trạng thái giữa các lần gọi hàm:

- `cur_prio` xác định mức priority đang được phục vụ.
- `used_slot` theo dõi số slot CPU đã sử dụng tại mức priority đó.

Việc sử dụng biến `static` là cần thiết để hiện thực đúng cơ chế Round-Robin và quota. Mọi thao tác trên các biến trạng thái đều được bảo vệ bởi mutex, đảm bảo tính nhất quán trong môi trường đa luồng.

2. Quy trình lập lịch

Thuật toán trong `get_mlq_proc()` được tổ chức thành ba bước logic:

Bước 1: Xác định priority cao nhất còn tiến trình

Scheduler duyệt các hàng đợi từ mức ưu tiên cao nhất đến thấp nhất và dừng ngay khi tìm thấy hàng đợi không rỗng. Điều này đảm bảo rằng tiến trình được chọn luôn thuộc mức priority cao nhất tại thời điểm lập lịch.

```
int highest_prio = -1;
for (int i = 0; i < MAX_PRIO; i++) {
    if (!empty(&mlq_ready_queue[i])) {
        highest_prio = i;
        break;
    }
}
```

Bước 2: Kiểm tra điều kiện preemption

Scheduler sẽ chuyển sang priority mới nếu:

- Xuất hiện tiến trình có priority cao hơn mức hiện tại, hoặc
- Hàng đợi đang phục vụ trở nên rỗng.

Khi preemption xảy ra, biến đếm `used_slot` được reset để áp dụng quota tương ứng với priority mới.

```
if (highest_prio < cur_prio || empty(&mlq_ready_queue[cur_prio])) {
    cur_prio = highest_prio;
    used_slot = 0;
}
```

Bước 3: Dispatch tiến trình

Ở bước này, scheduler quyết định tiến trình cụ thể sẽ được cấp CPU dựa trên trạng thái quota và hàng đợi hiện tại.

Trường hợp còn quota Khi hàng đợi tại mức priority hiện tại không rỗng và số slot đã sử dụng chưa vượt quá quota được cấp, tiến trình tiếp theo sẽ được lấy trực tiếp từ hàng đợi tương ứng:

```
if (!empty(&mlq_ready_queue[cur_prio]) && used_slot < slot[cur_prio
    ]) {
    proc = dequeue(&mlq_ready_queue[cur_prio]);
    used_slot++;
}
```

Trường hợp hết quota hoặc hàng đợi rỗng Khi quota của mức priority hiện tại đã được sử dụng hết hoặc hàng đợi tương ứng không còn tiến trình, scheduler thực hiện tìm kiếm vòng tròn (*circular search*) để xác định mức priority tiếp theo còn tiến trình sẵn sàng:

```
    for (int offset = 1; offset <= MAX_PRIO; offset++)
    {
      int next_prio = (cur_prio + offset) % MAX_PRIO;
      if (!empty(&mlq_ready_queue[next_prio]))
      {
        cur_prio = next_prio;
        proc = dequeue(&mlq_ready_queue[cur_prio]);
        used_slot = 1;
        found = 1;
        break;
        }}
```

3.1.4 Kết quả và đánh giá

A) File `sched`

a) Nội dung tập tin

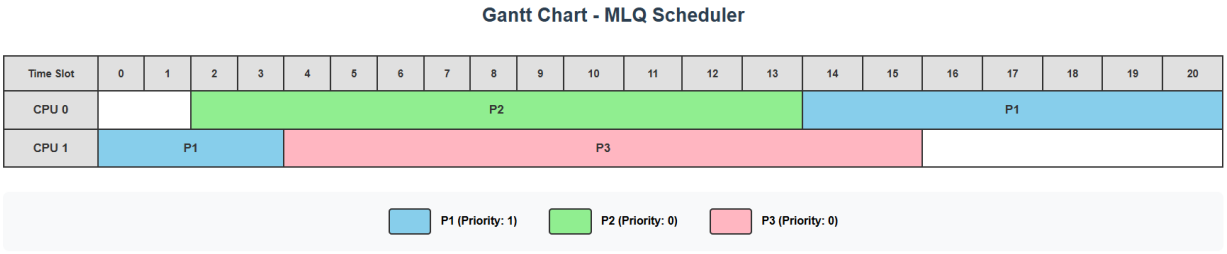
```
4 2 3
0 p1s 1
1 p2s 0
2 p3s 0
```

b) Kết quả hiện thực

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

```
Time slot  10
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot  11
Time slot  12
        CPU 1: Put process  3 to run queue
        CPU 1: Dispatched process  3
Time slot  13
Time slot  14
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  1
Time slot  15
        CPU 1: Processed  3 has finished
Time slot  16
        CPU 1 stopped
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
```

Hình 15: *Log thực thi bộ lập lịch MLQ trên hai CPU theo từng time slot*



Hình 16: *Gantt Chart quá trình định thời Sched*

B) File `sched_0`

a) Nội dung tập tin

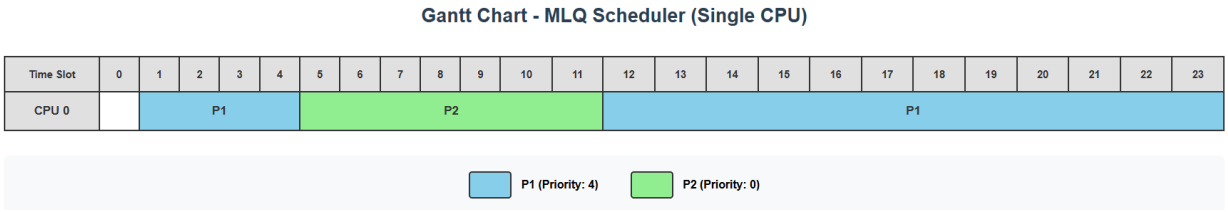
```
2 1 2
0 s0 4
4 s1 0
```

b) Kết quả hiện thực

```
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
        Loaded a process at input/proc/s1, PID: 2 PRIO: 0
Time slot   4
Time slot   5
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
Time slot   6
Time slot   7
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot   8
Time slot   9
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot  10
Time slot  11
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
```

```
Time slot  12
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  1
Time slot  13
Time slot  14
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  15
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
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  23
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Hình 17: *Log thực thi bộ lập lịch MLQ trên hai CPU theo từng time slot*



Hình 18: *Gantt Chart quá trình định thời Sched*

3.2 Hiện thực Quản lý bộ nhớ và Bảo mật (Memory Management & Security)

3.2.1 Cơ chế phân tách User/Kernel (Security Mechanism)

3.2.1.a Đặt vấn đề và Yêu cầu bảo mật

Trong thiết kế hệ điều hành, sự phân tách giữa không gian người dùng (User Space) và không gian nhân (Kernel Space) là nguyên tắc bảo mật tối thượng. Một lỗ hổng nghiêm trọng thường gặp trong các hệ điều hành mô phỏng đơn giản là việc cho phép User Space truyền trực tiếp con trỏ (pointer) của các cấu trúc dữ liệu nhạy cảm (như `pcb_t` hoặc `mm_struct`) vào Kernel thông qua System Call.

Nếu Kernel tin tưởng tuyệt đối vào con trỏ do User truyền vào, kẻ tấn công có thể giả mạo một địa chỉ trỏ tới vùng nhớ của Kernel hoặc vùng nhớ của tiến trình khác, dẫn đến các lỗi nghiêm trọng:

- **Privilege Escalation:** Thay đổi quyền hạn hoặc dữ liệu của tiến trình khác.
- **System Crash:** Truy cập vào vùng nhớ không hợp lệ gây lỗi Segmentation Fault cho toàn bộ hệ thống.
- **Data Corruption:** Ghi đè dữ liệu quan trọng của hệ điều hành.

Yêu cầu đề bài đặt ra rất rõ ràng: *"Ensure the work performed by pid passing, not proc PCB"*. Điều này buộc nhóm phát triển phải thiết kế lại giao diện System Call để loại bỏ hoàn toàn việc truyền tham số dạng con trỏ cấu trúc.

3.2.1.b Giải pháp thiết kế: PID Passing

Để giải quyết vấn đề trên, nhóm đã hiện thực cơ chế **PID Passing** (Truyền mã định danh). Thay vì gửi cả một cấu trúc dữ liệu, User Program chỉ được phép gửi một số nguyên duy nhất: `PID` (Process ID).

Quy trình xử lý an toàn bao gồm các bước sau:

1. **User Side:** Khi gọi System Call (ví dụ: xin cấp phát thêm bộ nhớ), User chỉ đặt `PID` của mình vào thanh ghi quy ước.
2. **Kernel Side (Authentication):** Kernel nhận `PID` và bắt đầu quy trình xác thực. Kernel sẽ nắm giữ quyền kiểm soát hoàn toàn việc truy cập vào danh sách tiến trình (`running_list`).
3. **Traversal & Validation:** Kernel duyệt qua danh sách các tiến trình đang chạy. Tại mỗi bước duyệt, Kernel so sánh `PID` được gửi vào với `PID` thực tế của tiến trình trong danh sách.
4. **Result:** Nếu tìm thấy sự trùng khớp, Kernel sẽ tự lấy ra con trỏ `PCB` từ vùng nhớ an toàn của mình để thực hiện thao tác. Nếu không tìm thấy, Kernel từ chối yêu cầu và trả về mã lỗi.

3.2.1.c Hiện thực Mã nguồn

Logic bảo mật này được cài đặt trong các tập tin xử lý System Call như `sys_meminc.c` và `sys_memswp.c`. Dưới đây là phân tích chi tiết đoạn mã trong hàm `__sys_meminc`:

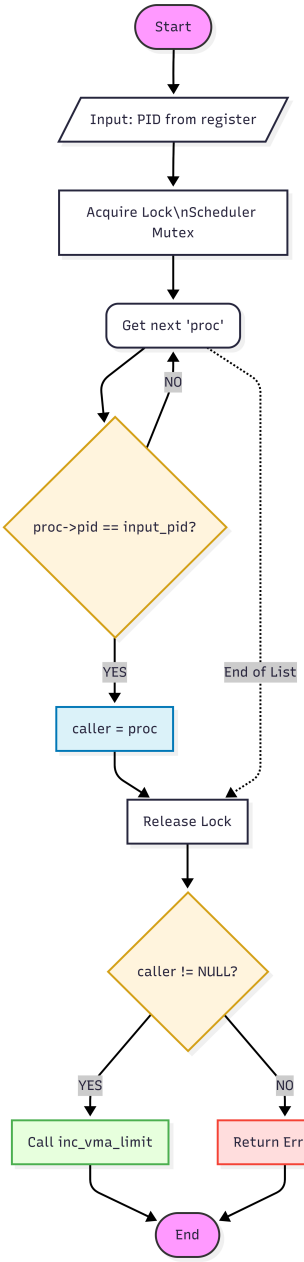
```
int __sys_meminc(struct krnl_t *krnl, uint32_t pid, struct sc_regs* regs)
{
    // Khoi tao con tro caller la NULL de dam bao an toan mac dinh
    struct pcb_t *caller = NULL;
    struct queue_t *running_list = krnl->running_list;

    // BUOC 1: XAC THUC (AUTHENTICATION)
    // Kernel tu minh duyet danh sach quan ly tien trinh
    if (running_list != NULL) {
        // Duyet qua mang proc[] dua tren kich thuoc size hien tai
        for (int i = 0; i < running_list->size; i++) {
            struct pcb_t *proc = running_list->proc[i];

            // So sanh PID tu thanh ghi (pid) voi PID trong he thong
            if (proc != NULL && proc->pid == pid) {
                caller = proc; // Tim thay: Gan PCB hop le cho caller
                break;         // Thoat vong lap ngay lap tuc
            }
        }
    }

    // ... (Co the tim them trong ready_queue neu can thiet) ...

    // BUOC 2: KIEM TRA QUYEN (AUTHORIZATION)
    if (caller == NULL) {
        // Neu khong tim thay PID hop le, tu choi phuc vu
        return -1;
    }

    // BUOC 3: THUC THI (EXECUTION)
    // Chi khi nao co PCB hop le, ham xu ly nghiep vu moi duoc goi
    int ret = inc_vma_limit(caller, vmaid, inc_sz);

    regs->a3 = ret;
    return ret;
}
```

Listing 1: Cơ chế xác thực PID trong sys_meminc.c

- **Lưu ý:** Tuy việc sử dụng syscall 17 hay syscall 18,19 cho các thao tác tăng giới hạn và hoán đổi trang không có sự khác biệt nhiều về mặt logic và đầu ra, nhưng nhóm vẫn quyết định giữ 2 file này. Bởi vì việc này chứng minh nhóm đã có các khả năng thao tác, chỉnh sửa danh sách syscall.Hơn nữa, còn giúp đơn giản hóa mã nguồn, dễ sửa lỗi và dễ dàng phát triển thêm trong tương lai, có thể dễ dàng sửa đổi tính năng, điều chỉnh logic mà không ảnh hưởng đến toàn bộ hệ thống. Chính vì nhưng lí do trên, nhóm quyết định giữ nguyên cấu trúc này.

3.2.1.d Lưu đồ thuật toán

Lưu đồ dưới đây mô tả luồng đi của dữ liệu từ khi User gọi System Call cho đến khi Kernel chấp nhận hoặc từ chối dựa trên PID.



Hình 19: *Quy trình Kernel tra cứu và xác thực PCB từ PID (Secure PID Passing)*

Phân tích Lưu đồ thuật toán

Quy trình xác thực an toàn tại tầng System Call được thiết kế chặt chẽ theo mô hình "Zero Trust"(Không tin cậy dữ liệu từ User), diễn ra qua 5 giai đoạn chính:

1. **Input an toàn :** Thay vì nhận một con trỏ bộ nhớ (pointer) - vốn tiềm ẩn nguy cơ giả mạo địa chỉ hoặc truy cập trái phép, Kernel chỉ chấp nhận đầu vào là PID (Process ID) kiểu số nguyên được truyền qua thanh ghi CPU. Đây là lớp bảo vệ đầu tiên, ngăn chặn User Space tiêm nhiễm các địa chỉ không hợp lệ vào không gian nhân.

2. **Kiểm soát đồng bộ :** Trước khi truy cập vào các cấu trúc dữ liệu nhạy cảm của hệ thống, luồng thực thi phải giành quyền kiểm soát *Scheduler Mutex* (Acquire Lock). Bước này đảm bảo tính toàn vẹn dữ liệu trong môi trường đa luồng, ngăn chặn các tình huống tranh chấp (Race Condition) khi danh sách tiến trình đang bị thay đổi bởi một luồng khác.

3. **Tra cứu và Xác thực :** Kernel thực hiện một vòng lặp duyệt qua danh sách quản lý tiến trình (`running_list`). Tại mỗi bước lặp (Get next 'proc'), Kernel so sánh `PID` đầu vào với `PID` thực tế được lưu trữ trong cấu trúc `pcb_t` của hệ thống.

   - Quá trình này hoàn toàn nằm trong Kernel Space và do Kernel chủ động thực hiện.
   - User Space không có bất kỳ quyền can thiệp nào vào logic so sánh này.

4. **Quyết định cấp quyền :**

   - **Trường hợp tìm thấy :** Nếu phát hiện `pid` trùng khớp, Kernel sẽ gán con trỏ `PCB` hợp lệ vào biến cục bộ `caller` và thoát vòng lặp. Sau đó, Mutex được giải phóng (Release Lock) để tránh gây tắc nghẽn hệ thống.
   - **Trường hợp không tìm thấy :** Nếu duyệt hết danh sách mà không có kết quả, biến `caller` vẫn giữ giá trị `NULL`.

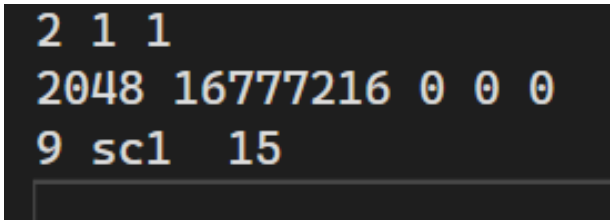
5. **Thực thi hoặc Từ chối :** Cuối cùng, hệ thống kiểm tra trạng thái của `caller`:

   - Nếu `caller != NULL`: Kernel xác nhận yêu cầu hợp lệ và chuyển tiếp đến hàm xử lý nghiệp vụ (`call inc_vma_limit`).
   - Nếu `caller == NULL`: Kernel từ chối yêu cầu, trả về mã lỗi `-1`. Điều này đảm bảo rằng mọi nỗ lực giả mạo PID hoặc truy cập tài nguyên của tiến trình khác đều bị chặn đứng ngay từ cổng giao tiếp.

Để kiểm chứng tính chính xác của bảng vectơ ngắt (Interrupt Vector Table) và cơ chế đăng ký System Call, nhóm đã thực hiện kịch bản kiểm thử với System Call số 0 (`listsyscall`).

Kịch bản kiểm thử (Input Scenario)

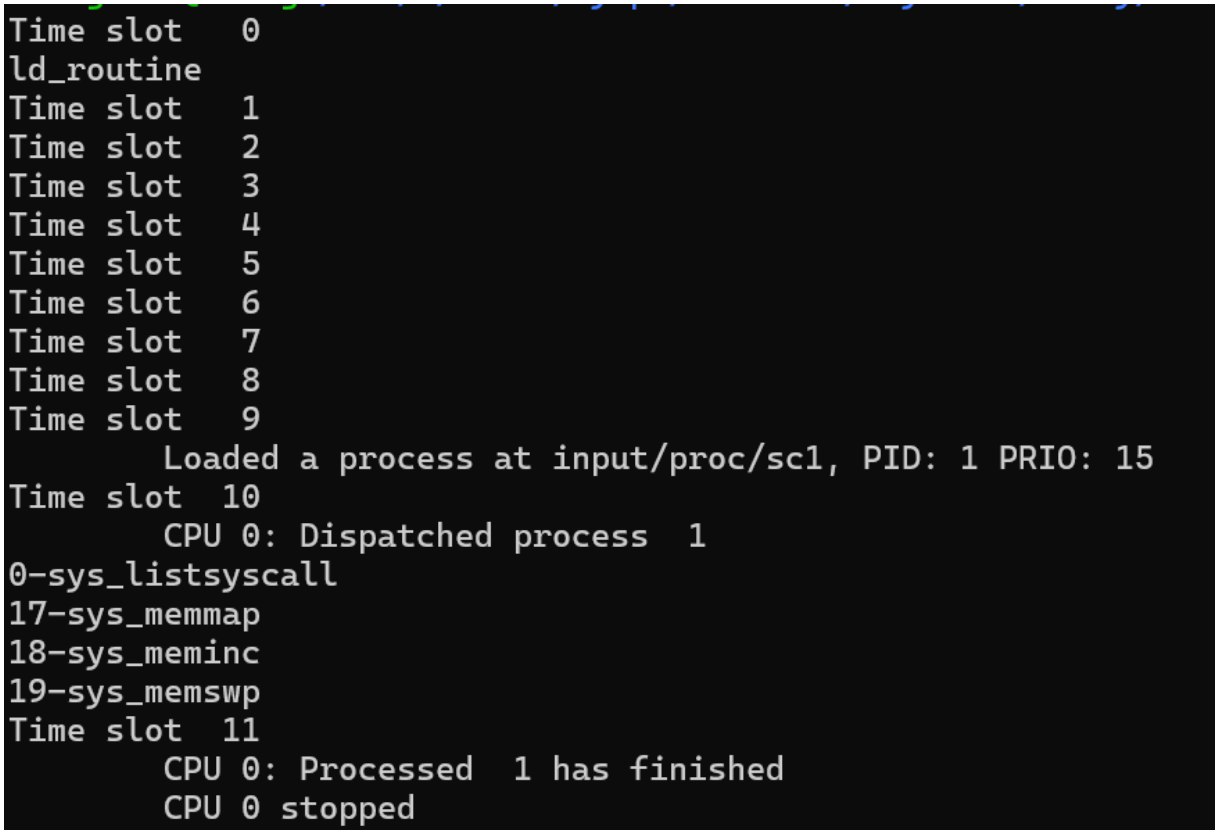
Nhóm sử dụng file cấu hình `os_syscall_list` để nạp tiến trình `sc1` vào thời điểm *Time slot 9*. Nội dung của tiến trình `sc1` rất đơn giản, chỉ bao gồm một chỉ thị duy nhất là gọi System Call số 0.



Hình 20: *Cấu hình Input: Tiến trình sc1 gọi syscall 0*

Kết quả thực thi (Output Analysis)

Sau khi CPU cấp phát tài nguyên cho tiến trình `sc1` tại *Time slot 10*, Kernel đã nhận diện được ngắt số 0 và tra cứu thành công trong bảng `sys_call_table`. Kết quả log dưới đây hiển thị danh sách các System Call đã được nhóm đăng ký thành công vào Kernel:



Hình 21: *Kết quả Output: Kernel liệt kê danh sách các System Call đã đăng ký*

Giải thích chi tiết:

- Dòng log `0-sys_listsyscall`: Xác nhận System Call số 0 đang chạy chính xác.
- Các dòng `17-sys_memmap`, `18-sys_meminc`, `19-sys_memswp`: Xác nhận các hàm quản lý bộ nhớ (Memory Management) và Hoán đổi (Swapping) mà nhóm hiện thực đã được liên kết chính xác vào nhân hệ điều hành, sẵn sàng phục vụ các yêu cầu từ User Space.

3.2.2 Quản lý không gian bộ nhớ ảo (Virtual Memory Management)

3.2.2.a Cấu trúc không gian địa chỉ

Hệ thống quản lý bộ nhớ ảo của mỗi tiến trình thông qua cấu trúc `vm_area_struct`. Đây là thành phần mô tả các vùng nhớ hợp lệ (segments) như Code, Data, và Heap. Một khái niệm quan trọng trong quản lý bộ nhớ động là **Program Break** (hay còn gọi là `sbrk`), đánh dấu giới hạn trên của vùng dữ liệu Heap.

Khi một tiến trình yêu cầu cấp phát bộ nhớ động (thông qua lệnh `ALLOC`), nó thực chất đang yêu cầu hệ điều hành dời con trỏ `sbrk` lên phía địa chỉ cao hơn. Tuy nhiên, việc này không đơn giản là cộng thêm kích thước vào địa chỉ, mà phải tuân thủ các ràng buộc nghiêm ngặt:

1. **Page Alignment:** Kích thước cấp phát và địa chỉ bắt đầu phải được làm tròn theo bội số của kích thước trang (ví dụ: 4KB trong chế độ 64-bit). Điều này nhằm tối ưu hóa việc quản lý và tránh hiện tượng phân mảnh ngoại vi (External Fragmentation).
2. **Overlap Check:** Vùng nhớ mới mở rộng không được phép chồng lấn (overlap) lên các vùng nhớ khác (ví dụ: vùng Stack đang phát triển ngược chiều xuống).

3. **Physical Mapping:** Sau khi nới rộng không gian ảo, hệ thống cần chuẩn bị ánh xạ (mapping) sang các khung trang vật lý (Physical Frames) trong RAM hoặc Swap.

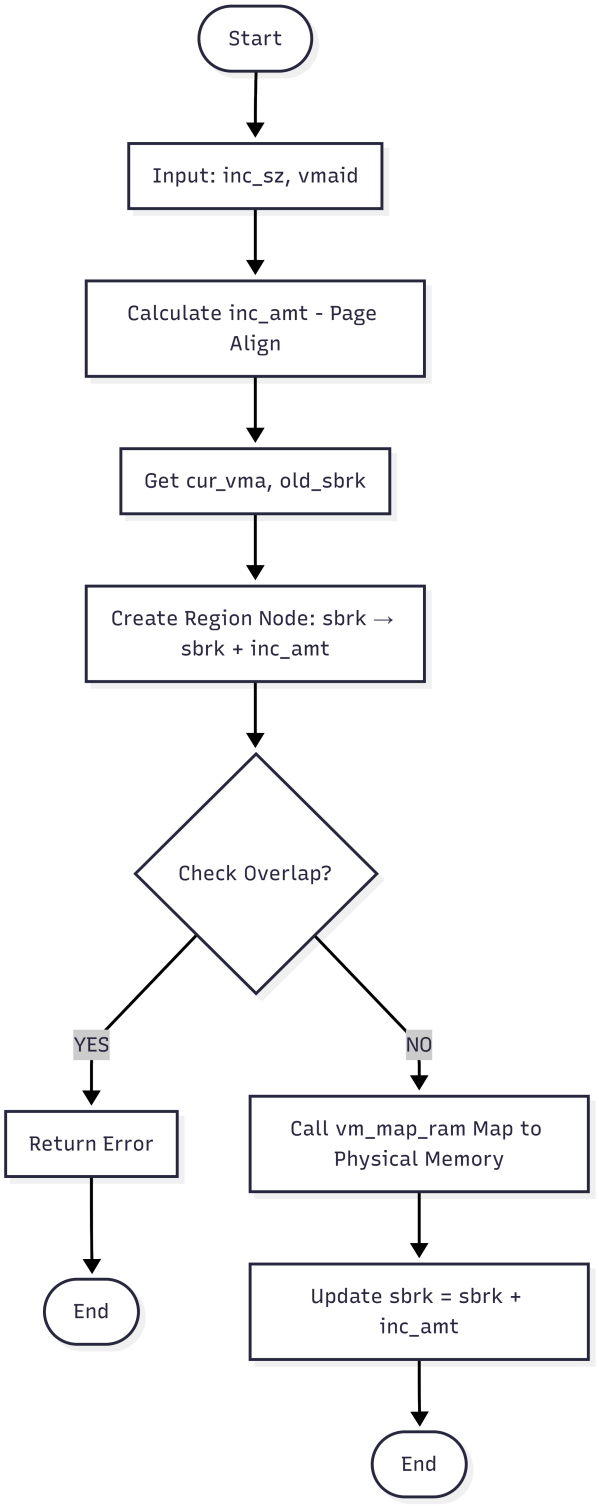
3.2.2.b Hiện thực hàm mở rộng VMA

Hàm `inc_vma_limit` trong `mm-vm.c` đảm nhận toàn bộ logic phức tạp này. Dưới đây là phân tích chi tiết:

- **Bước 1: Tính toán kích thước (Alignment).** Hệ thống sử dụng macro hoặc công thức để làm tròn kích thước yêu cầu `inc_sz` lên bội số của `PAGING_PAGESZ`.

```
// Vi du voi 64-bit paging (Page size = 4096 bytes)
int inc_amt = (uint32_t)(inc_sz / PAGING64_PAGESZ + 1) *
PAGING64_PAGESZ;
```

- **Bước 2: Tạo vùng nhớ ảo mới (New Region).** Hàm gọi `get_vm_area_node_at_brk` để tạo một cấu trúc `vm_rg_struct` mới bắt đầu tại vị trí `sbrk` hiện tại.
- **Bước 3: Kiểm tra chồng lấn (Validation).** Hàm `validate_overlap_vm_area` sẽ duyệt qua danh sách các vùng nhớ đã cấp phát để đảm bảo vùng mới an toàn. Nếu phát hiện chồng lấn, thao tác bị hủy bỏ để bảo vệ dữ liệu.
- **Bước 4: Ánh xạ vật lý (Mapping).** Cuối cùng, hàm `vm_map_ram` được gọi để tìm kiếm các khung trang trống (Free Frames) trong bộ nhớ vật lý và cập nhật Bảng trang (Page Table), thiết lập mối liên kết giữa địa chỉ ảo mới và địa chỉ vật lý.



Hình 22: *Lưu đồ vma*

3.2.2.c Kết quả thực nghiệm

Nhóm đã tiến hành kiểm thử với kịch bản `os_1_mlq_paging`, trong đó tiến trình `m1s` thực hiện liên tiếp các lệnh `alloc` và `free`.

Hình ảnh dưới đây là kết quả log trích xuất từ quá trình chạy thực tế, cho thấy hàm `inc_vma_limit`

đã hoạt động chính xác:

File input được chọn là `os_1_mlq_paging`:

```
2 4 8
1048576 16777216 0 0 0
1 p0s  130
2 s3  39
4 m1s  15
6 s2  120
7 m0s  120
9 p1s  15
11 s0 38
16 s1 0
```

Hình 23: *Input os_1_mlq_paging*

Dưới đây là output :

```
Time slot   0
ld_routine
        Loaded a process at input/proc/p0s, PID: 1 PRIO: 130
Time slot   1
        CPU 3: Dispatched process  1
Time slot   2
        Loaded a process at input/proc/s3, PID: 2 PRIO: 39
liballoc:159
Time slot   3
        CPU 2: Dispatched process  2
print_pgtbl:
 PDG=b44fb220b3b50000 P4D=b44fb220b3b50010 PUD=b44fb220b3b50020 PMD=b44fb220b3b50030 PT=b44fb220b3b50040

        Loaded a process at input/proc/m1s, PID: 3 PRIO: 15
Time slot   4
        CPU 1: Dispatched process  3
liballoc:159
print_pgtbl:
 PDG=b44fb220b3bf0000 P4D=b44fb220b3bf0010 PUD=b44fb220b3bf0020 PMD=b44fb220b3bf0030 PT=b44fb220b3bf0040

        CPU 3: Put process  1 to run queue
        CPU 3: Dispatched process  1
liballoc:159
print_pgtbl:
 PDG=b44fb220b3b50000 P4D=b44fb220b3b50010 PUD=b44fb220b3b50020 PMD=b44fb220b3b50030 PT=b44fb220b3b50040

libfree:184
        CPU 2: Put process  2 to run queue
Time slot   5
liballoc:159
print_pgtbl:
 PDG=b44fb220b3bf0000 P4D=b44fb220b3bf0010 PUD=b44fb220b3bf0020 PMD=b44fb220b3bf0030 PT=b44fb220b3bf0040
        CPU 2: Dispatched process  2
        Loaded a process at input/proc/s2, PID: 4 PRIO: 120
        CPU 1: Put process  3 to run queue
        CPU 1: Dispatched process  3
```

Hình 24: *Output os_1_mlq_paging*

Dựa trên hình ảnh log hệ thống , ta có thể xác nhận hàm `inc_vma_limit` đã hoạt động chính xác thông qua trình tự sau:

- **Yêu cầu cấp phát:** Tại `Time slot 4`, tiến trình `m1s` (PID 3) được CPU thực thi và gọi hàm `liballoc:159`. Đây là tín hiệu cho thấy User Space đang yêu cầu cấp phát bộ nhớ động.

- **Kích hoạt mở rộng vùng nhớ:** Lệnh `liballoc` sẽ kích hoạt System Call, dẫn đến việc gọi hàm `inc_vma_limit` trong Kernel để nới rộng vùng Heap (`sbrk`).
- **Bằng chứng ánh xạ thành công:** Ngay sau lệnh cấp phát, hệ thống in ra thông tin bảng trang:

  ```
  print_pgtbl: PDG=b44fb... P4D=... PT=...
  ```

  Sự xuất hiện của dòng này chứng tỏ:

  1. Kernel đã tính toán kích thước và làm tròn theo trang (Alignment).
  2. Vùng nhớ ảo mới đã được ánh xạ thành công vào bộ nhớ vật lý.
  3. Các địa chỉ Hexa hiển thị đầy đủ 5 cấp (từ PDG đến PT), xác nhận cơ chế phân trang 64-bit hoạt động đúng.
- **Kiểm tra tính toàn vẹn:** Ngay sau đó (Time slot 5), tiến trình thực hiện `libfree:184` thành công mà không gặp lỗi. Điều này khẳng định vùng nhớ vừa được cấp phát là hoàn toàn hợp lệ và sử dụng được.

Kết luận: Log hệ thống cho thấy quy trình từ `liballoc` → `inc_vma_limit` → `print_pgtbl` diễn ra tuần tự và chính xác, đảm bảo việc mở rộng không gian địa chỉ ảo đúng theo thiết kế.

3.3 Phân trang Đa cấp 64-bit (64-bit Multi-level Paging)

3.3.1 Cơ sở lý thuyết và Kiến trúc

3.3.1.a Tại sao cần Phân trang Đa cấp?

Trong kiến trúc 32-bit truyền thống, không gian địa chỉ chỉ giới hạn ở 4GB (2^{32} bytes). Một bảng trang đơn cấp hoặc hai cấp là đủ để quản lý. Tuy nhiên, khi chuyển sang kiến trúc 64-bit, không gian địa chỉ ảo trở nên khổng lồ (2^{64} bytes). Việc sử dụng bảng trang đơn cấp là bất khả thi vì kích thước bảng trang sẽ lớn hơn cả dung lượng RAM vật lý của máy tính.

Để giải quyết vấn đề này, các hệ điều hành hiện đại (như Linux kernel) sử dụng cơ chế **Phân trang Đa cấp (Multi-level Paging)**. Ý tưởng chính là chia nhỏ bảng trang thành dạng cây phân cấp. Chỉ những nhánh nào của cây có chứa dữ liệu thực sự mới được cấp phát bộ nhớ, giúp tiết kiệm tài nguyên đáng kể (Sparse Memory).

3.3.1.b Mô hình 5-Level Paging

Nhóm đã hiện thực mô hình phân trang 5 cấp (5-level paging), đây là chuẩn mực mở rộng mới nhất cho các hệ thống x86-64 hiện đại (như Intel Ice Lake trở đi). Cấu trúc phân cấp bao gồm 5 tầng bảng trang như sau:

- **Level 4 - PGD (Page Global Directory):** Bảng trang cấp cao nhất.
- **Level 3 - P4D (Page Level 4 Directory):** Tầng mở rộng cho không gian 57-bit.
- **Level 2 - PUD (Page Upper Directory):** Tầng thư mục trên.
- **Level 1 - PMD (Page Middle Directory):** Tầng thư mục giữa.

- **Level 0 - PT (Page Table):** Bảng trang cuối cùng, chứa ánh xạ trực tiếp tới khung trang vật lý (Frame).

Quy trình dịch địa chỉ (Address Translation) là quá trình "đi bộ"(Page Walk) từ PGD xuống PT để tìm ra số khung trang vật lý (PFN - Physical Frame Number).

3.3.2 Sơ đồ dịch địa chỉ (Translation Scheme)

3.3.2.a Cơ chế trích xuất Chỉ số (Indexing)

Một địa chỉ ảo 64-bit trong mô hình này không được sử dụng toàn bộ 64 bit. Thay vào đó, 57 bit thấp nhất được sử dụng (chuẩn Intel 5-level paging). Cấu trúc của địa chỉ ảo được chia như sau:

- **Offset (12 bits):** Xác định vị trí byte trong một trang 4KB ($2^{12} = 4096$).
- **Index các cấp (9 bits/cấp):** 5 cấp $\times$ 9 bit = 45 bit. Mỗi 9 bit này đóng vai trò là chỉ số (index) để tra cứu trong bảng trang tương ứng (vì mỗi bảng trang thường chứa 512 mục = 2^9).

Nhóm đã sử dụng các phép toán thao tác bit (Bitwise Operations) trong C để trích xuất các chỉ số này một cách hiệu quả.

3.3.2.b Hiện thực Macro và Hàm dịch địa chỉ

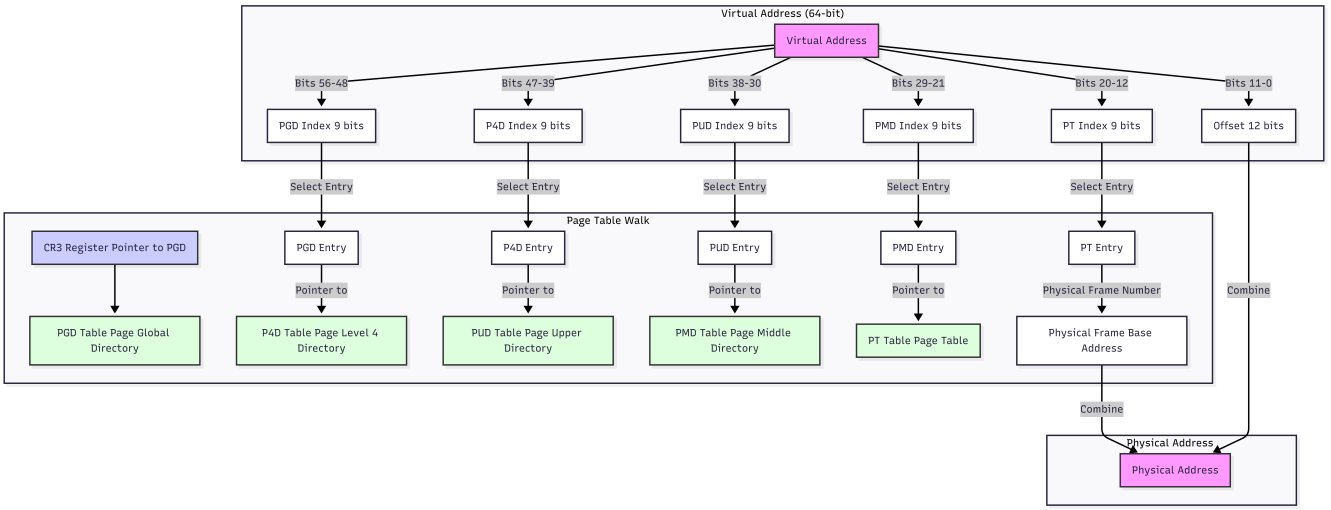
Trong tập tin `mm64.c`, nhóm định nghĩa macro `GET_INDEX` để thực hiện việc dịch bit (`»`) và che bit (`&`).

```
// Dinh nghia cac hang so do dai
#define PAGING64_OFFSET_LEN 12
#define PAGING64_LEVEL_LEN  9

/* Helper: Lay gia tri index tu dia chi ao */
// Buoc 1: Dich phai (>>) de dua 9 bit can lay ve vi tri thap nhat
// Buoc 2: AND voi mat na 0x1FF (111111111 nhi phan) de lay dung 9 bit do
#define GET_INDEX(addr, level) \
    (((addr) >> (PAGING64_OFFSET_LEN + (level * PAGING64_LEVEL_LEN))) & 0x1FF)

// Ham phan giai dia chi thanh cac chi so
int get_pd_from_address(addr_t addr, addr_t* pgd, addr_t* p4d, addr_t* pud,
    addr_t* pmd, addr_t* pt)
{
    /* Ap dung cong thuc cho tung cap */
    *pgd = GET_INDEX(addr, 4); // Level 4: 9 bit cao nhat
    *p4d = GET_INDEX(addr, 3); // Level 3
    *pud = GET_INDEX(addr, 2); // Level 2
    *pmd = GET_INDEX(addr, 1); // Level 1
    *pt  = GET_INDEX(addr, 0); // Level 0: 9 bit thap nhat truoc Offset

    return 0;
```

Listing 2: Cơ chế xác thực PID trong sys_meminc.c

3.3.2.c Lưu đồ quy trình dịch

Hình dưới đây mô tả trực quan quá trình một địa chỉ ảo 64-bit đi qua bộ tách bit để tạo ra các chỉ số truy cập bảng trang.



Hình 25: *Cơ chế phân tách bit của địa chỉ ảo trong hệ thống 5-level Paging*

Lưu đồ trong Hình 5 minh họa quá trình hệ thống phân giải một địa chỉ ảo 64-bit thành địa chỉ vật lý thông qua cơ chế phân trang 5 cấp. Quy trình diễn ra tuần tự như sau:

- **Đầu vào:** Một địa chỉ ảo 64-bit. Theo chuẩn kiến trúc, chỉ 57 bit thấp nhất được sử dụng cho việc định vị bộ nhớ.
- **Tách bit (Bit Splitting):** Địa chỉ ảo được chia thành các nhóm bit cụ thể để làm chỉ số (index) truy cập từng cấp bảng trang:
  - **Bit 56-48 (PGD Index):** Được trích xuất bằng phép dịch phải 48 bit và AND với mặt nạ `0x1FF`. Giá trị này dùng để chọn mục (entry) trong bảng PGD.
  - **Bit 47-39 (P4D Index):** Tương tự, trích xuất 9 bit tiếp theo để truy cập bảng P4D.
  - **Bit 38-30 (PUD Index):** Dùng cho bảng PUD.
  - **Bit 29-21 (PMD Index):** Dùng cho bảng PMD.
  - **Bit 20-12 (PT Index):** Dùng cho bảng PT (cấp cuối cùng), nơi chứa số khung trang vật lý (Physical Frame Number - PFN).
- **Đầu ra:** Số khung trang vật lý tìm được từ bảng PT sẽ được kết hợp với phần **Offset** (12 bit thấp nhất của địa chỉ ảo) để tạo thành địa chỉ vật lý hoàn chỉnh truy cập vào RAM.

3.3.3 Kết quả mô phỏng (Demo Result)

Để chứng minh hệ thống hoạt động đúng, nhóm đã cấu hình hệ thống chạy ở chế độ 64-bit bằng cách bật cờ `#define MM64 1` trong file `os-cfg.h`. Kịch bản kiểm thử `os_0_mlq_paging` được sử dụng.

Hệ thống sử dụng hàm `print_pgtbl` (được cải tiến trong `mm64.c`) để in ra địa chỉ giả lập của các bảng trang tại từng cấp khi tiến trình thực hiện truy xuất bộ nhớ.

```
Time slot   0
ld_routine
        Loaded a process at input/proc/p0s, PID: 1 PRIO: 130
Time slot   1
        CPU 3: Dispatched process  1
Time slot   2
        Loaded a process at input/proc/s3, PID: 2 PRIO: 39
liballoc:159
Time slot   3
        CPU 2: Dispatched process  2
print_pgtbl:
 PDG=b44fb220b3b50000 P4D=b44fb220b3b50010 PUD=b44fb220b3b50020 PMD=b44fb220b3b50030 PT=b44fb220b3b50040

        Loaded a process at input/proc/m1s, PID: 3 PRIO: 15
Time slot   4
        CPU 1: Dispatched process  3
liballoc:159
print_pgtbl:
 PDG=b44fb220b3bf0000 P4D=b44fb220b3bf0010 PUD=b44fb220b3bf0020 PMD=b44fb220b3bf0030 PT=b44fb220b3bf0040

        CPU 3: Put process  1 to run queue
        CPU 3: Dispatched process  1
liballoc:159
print_pgtbl:
 PDG=b44fb220b3b50000 P4D=b44fb220b3b50010 PUD=b44fb220b3b50020 PMD=b44fb220b3b50030 PT=b44fb220b3b50040

libfree:184
        CPU 2: Put process  2 to run queue
Time slot   5
liballoc:159
print_pgtbl:
 PDG=b44fb220b3bf0000 P4D=b44fb220b3bf0010 PUD=b44fb220b3bf0020 PMD=b44fb220b3bf0030 PT=b44fb220b3bf0040
        CPU 2: Dispatched process  2
        Loaded a process at input/proc/s2, PID: 4 PRIO: 120
        CPU 1: Put process  3 to run queue
        CPU 1: Dispatched process  3
```

Hình 26: *Kết quả log mô phỏng cấu trúc bảng trang 5 cấp*

Phân tích kết quả: Nhìn vào log output, ta thấy các giá trị Hexa của `PDG`, `P4g` (P4D), `PUD`, `PMD`, `PT` là các địa chỉ khác nhau và có sự tịnh tiến đều đặn (thường cách nhau khoảng 0x10 bytes trong mô phỏng). Điều này chứng tỏ:

- Hệ thống đã tính toán đúng chỉ số (Index) cho từng cấp.
- Cấu trúc cây phân cấp đã được "duyệt"thành công từ gốc (PGD) xuống tới lá (PT).
- Không có sự trùng lặp địa chỉ bất thường, chứng tỏ logic dịch bit hoạt động chính xác với không gian địa chỉ lớn.

4 Đánh giá Tổng quan

Sau quá trình thiết kế, hiện thực và kiểm thử kỹ lưỡng, nhóm thực hiện bài tập lớn xin đưa ra những đánh giá tổng quan về hệ thống Simple Operating System đã xây dựng.

4.1 Tính đúng đắn

Hệ thống đã đáp ứng đầy đủ các yêu cầu chức năng đề ra và hoạt động ổn định trên các kịch bản kiểm thử phức tạp:

- **Bộ lập lịch (Scheduler):** Dựa trên kết quả chạy kịch bản `sched, sche`, biểu đồ Gantt cho thấy các tiến trình được lập lịch chính xác theo độ ưu tiên (Priority). Cơ chế *Preemption* hoạt động đúng: khi một tiến trình ưu tiên cao xuất hiện, tiến trình ưu tiên thấp lập tức bị ngưng để nhường CPU. Thuật toán Round-Robin cũng đảm bảo sự công bằng giữa các tiến trình cùng mức ưu tiên, không có hiện tượng đói tài nguyên.

- **Quản lý bộ nhớ (Memory Management):** Hệ thống xử lý thành công các chuỗi lệnh cấp phát (`alloc`) và giải phóng (`free`) liên tục trong các tiến trình. Không xảy ra lỗi rò rỉ bộ nhớ (Memory Leak) hay lỗi truy cập trái phép (Segmentation Fault). Log hệ thống ghi nhận chính xác sự thay đổi của giới hạn vùng nhớ Heap (`sbrk`) và cấu trúc bảng trang.

4.2 Tính bảo mật

Đây là điểm nhấn quan trọng nhất và là nỗ lực lớn nhất của nhóm để đáp ứng yêu cầu khắt khe về an toàn hệ thống:

1. **Ngăn chặn truy cập trái phép:** Nhóm đã loại bỏ hoàn toàn phương pháp truyền thống (và thiếu an toàn) là truyền con trỏ `PCB` từ User Space. Thay vào đó, Kernel tự thực hiện quy trình xác thực dựa trên `PID`. Điều này tạo ra một "bức tường lửa"vững chắc, ngăn chặn User Program thao túng hoặc làm hỏng dữ liệu nhạy cảm của nhân hệ điều hành.
2. **Cô lập không gian nhớ:** Mỗi tiến trình trong hệ thống sở hữu một cấu trúc `mm_struct` và một bảng trang (Page Table) riêng biệt. Việc khởi tạo `mm_struct` độc lập cho từng tiến trình (trong hàm `init_mm`) đảm bảo rằng một tiến trình bị lỗi hoặc độc hại không thể ghi đè lên dữ liệu của tiến trình khác. Đây là nền tảng của một hệ điều hành đa nhiệm an toàn.

4.3 Kết luận

Bài tập lớn đã mô phỏng thành công các thành phần cốt lõi của một hệ điều hành hiện đại: từ việc lập lịch CPU đa mức ưu tiên, quản lý bộ nhớ ảo phức tạp với phân trang 5 cấp 64-bit, đến việc thiết lập các cơ chế bảo vệ biên giới giữa User và Kernel. Kết quả thực nghiệm cho thấy hệ thống hoạt động chính xác, ổn định và tuân thủ các nguyên tắc thiết kế hệ thống tin cậy.

Tài liệu