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

HỆ ĐIỀU HÀNH - CO2018

BÁO CÁO BÀI TẬP LỚN
SIMPLE OPERATING SYSTEM IMPLEMENTATION

Danh sách nhóm thực hiện

Giảng viên hướng dẫn: Hoàng Lê Hải Thanh
Sinh viên thực hiện: Hồ Đăng Khoa - 2211588
Hoàng Sĩ Xuân Sơn - 2212937
Bùi Đăng Khoa - 2211581
Nguyễn Thị Thúy Nga - 2212168

TP Hồ Chí Minh, Tháng 11 năm 2024

Danh sách thành viên

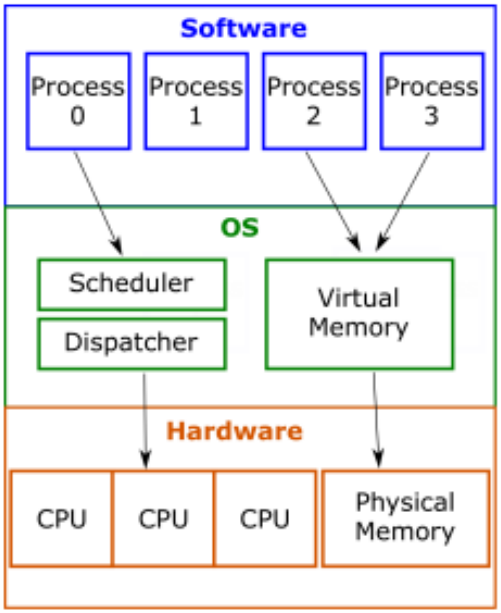
STT	Họ và tên	MSSV	Nhiệm vụ	Đóng góp (%)
1	Hồ Đăng Khoa	2211588	– Hiện thực phần Scheduling – Trả lời câu hỏi – Làm báo cáo	100%
2	Hoàng Sỹ Xuân Sơn	2212927	– Hiện thực phần Memory và Paging – Trả lời câu hỏi – Làm báo cáo	100%
3	Bùi Đăng Khoa	2211581	– Hiện thực phần Memory và Paging – Hiện thực phần Synchronization – Trả lời câu hỏi	100%
4	Nguyễn Thị Thúy Nga	2212168	– Hiện thực phần Memory và Paging – Hiện thực phần Synchronization – Trả lời câu hỏi	100%

Mục lục

Chương 1

Phần Mở Đầu

1.1 Giới thiệu đề tài.



Hình 1.1: Mô hình hệ điều hành.

Hệ điều hành (Operating System) là một phần mềm quản lý và điều khiển tài nguyên của máy tính, bao gồm phần cứng và phần mềm, để cung cấp môi trường làm việc cho người dùng và ứng dụng. Hệ điều hành đóng vai trò trung tâm trong việc quản lý các hoạt động của máy tính, bao gồm việc quản lý bộ nhớ, định thời,

giao tiếp với phần cứng, và cung cấp giao diện người dùng.

Hệ điều hành bao gồm khả năng tối ưu hóa hiệu suất của máy tính, giúp người dùng dễ dàng tương tác với các ứng dụng và tài nguyên khác nhau. Nó cung cấp một giao diện người dùng thân thiện, cho phép thực hiện các tác vụ một các dễ dàng thông qua các ứng dụng và các công cụ quản lý hệ thống. Hơn nữa, hệ điều hành cung cấp các tính tăng bảo mật để bảo vệ dữ liệu và thông tin cá nhân của người dùng khỏi mối đe dọa từ bên ngoài.

1.2 Yêu cầu đề tài.

1. Yêu cầu Kiến thức: Tìm hiểu về cấu trúc Multi-level queue Scheduling, một cấu trúc dùng để sắp xếp các tác vụ dựa trên độ ưu tiên của các tác vụ ấy, kiến thức về Heap, bộ nhớ ảo, bộ nhớ thực và ánh xạ giữa hai bộ nhớ dựa trên kỹ thuật phân trang, các kiến thức về tài nguyên, cách bảo vệ tài nguyên khi hiện thực.

2. Yêu cầu Thiết kế: Hiện thực 2 thành phần chính của một Hệ điều hành đơn giản bao gồm Scheduler và Virtual Memory. Trong đó:

    - Scheduler: Xác định tiến trình (process) nào sẽ được thực thi trên CPU tại một thời điểm nhất định.
    - Virtual Memory: Mỗi tiến trình có một vùng riêng trong bộ nhớ ảo. Nhiệm vụ của bộ nhớ ảo là thực hiện các thao tác ánh xạ và biên dịch địa chỉ của vùng nhớ ấy cung cấp bởi các tiến trình tới địa chỉ vật lý tương ứng trên bộ nhớ vật lý

1.3 Mục tiêu nghiên cứu.

- Giúp sinh viên có cái nhìn tổng quan về hệ điều hành, đồng thời có thêm kiến thức về Scheduling, Virtual Memory.

- Giúp sinh viên hiểu được nguyên lý cũng như cách thức hoạt động của một hệ điều hành đơn giản. Đồng thời hiểu được từng vai trò, ý nghĩa và cách thực hiện của từng module trong hệ điều hành, cách nó hoạt động.

Chương 2

Nội dung.

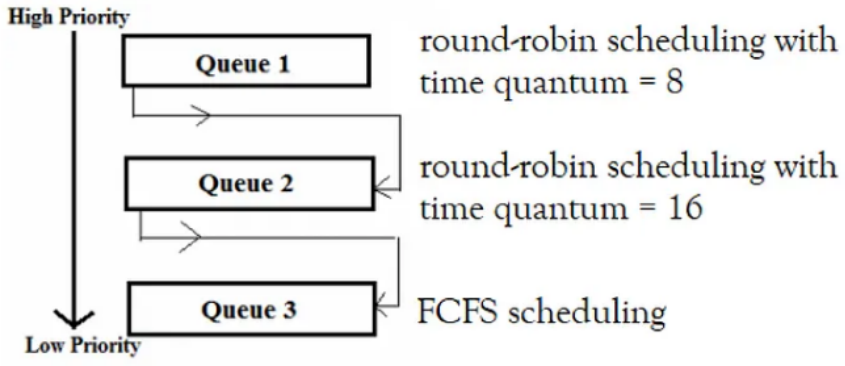
2.1 Scheduler.

2.1.1 Lý thuyết

Chính sách định thời mà hệ điều hành được trang bị là cơ chế Multi Level Queue (MLQ) kết hợp Round Robin. Các giá trị đặc trưng khi làm việc với MLQ:

- Fixed priority: độ ưu tiên của hàng đợi. Hệ điều hành phục vụ từ hàng đợi có độ ưu tiên cao đến hàng đợi có độ ưu tiên thấp. Giá trị này càng thấp thì có độ ưu tiên càng cao, và vì giá trị này được cố định nên có thể có tình trạng starvation.
- Time slice: mỗi hàng đợi được nhận một khoảng thời gian chiếm CPU và phân phối cho các process trong hàng đợi khoảng thời gian đó.

Trong hệ thống, mỗi tiến trình có một giá trị `prio` nhất định thể hiện độ ưu tiên của một tiến trình và được dùng để xác định `ready_queue` nào sẽ được dùng để chứa tiến trình đó, giá trị `prio` càng thấp thể hiện cho độ ưu tiên càng cao.



Hình 2.1: Ý tưởng Multi Level Queue

Ban đầu, các hàng đợi có độ ưu tiên khác nhau sẽ có số lần sử dụng CPU khác nhau (thể hiện qua giá trị **slot**, tính bằng công thức `MAX_PRIO - prio`), hàng đợi có độ ưu tiên càng cao thì càng có nhiều lượt sử dụng CPU.
Hàng đợi có độ ưu tiên cao nhất sẽ được chọn để thi thực trước cho đến khi đã sử dụng hết số slot cho phép. Lúc này, CPU sẽ chuyển sang thực thi các tiến trình trong hàng đợi có độ ưu tiên thấp hơn. Tất cả hàng đợi sẽ được áp dụng cơ chế Round Robin, tức là việc thực thi một hàng đợi chính xác là thay phiên nhau thực thi các tiến trình trong hàng đợi ấy theo một giá trị **time slice** cho trước, một lần thay phiên được tính là một lần sử dụng CPU.
Đến một thời điểm mà tất cả các hàng đợi đều sử dụng hết số slot ban đầu, hệ thống sẽ cấp phát lại số **slot** cho từng hàng đợi theo công thức ban đầu và sau đó lại tiếp thực hiện công việc định thời.4

2.1.2 Trả lời câu hỏi.

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:
Trong bài tập lớn này, việc sử dụng **hàng đợi ưu tiên** mang lại nhiều ưu điểm so với các thuật toán định thời khác (FCFS, SJR hay RR). Đầu tiên, **hàng đợi ưu tiên** dùng để quản lý các tiến trình dựa trên độ ưu tiên của chúng. Khi một tiến trình mới được thêm vào hệ thống, nó được đưa vào **hàng đợi ưu tiên** dựa trên độ ưu tiên của nó so với các tiến trình khác đang chờ. Điều này cho phép các tiến trình quan trọng được ưu tiên hơn các tiến trình ít quan trọng hơn, dẫn đến hiệu suất và khả năng đáp ứng của hệ thống tốt hơn.
Bên cạnh đó, việc sử dụng **hàng đợi ưu tiên** cho phép định thời trong môi trường ưu tiên (preemptive scheduling). Nói cách khác, khi một tiến trình mới có độ ưu

tiên cao hơn được thêm vào, nó có thể ngắt bỏ hoặc tạm dừng tiến trình có độ ưu tiên thấp hơn đang chạy để định thời tiến trình mới bất cứ lúc nào. Điều này giúp công việc định thời được đảm bảo rằng các tiến trình quan trọng, có độ ưu tiên cao được hoàn thành đúng thời hạn và không lãng phí các tài nguyên cho các tiến trình có độ ưu tiên thấp hơn.

2.2 Memory Management.

2.2.1 Lý thuyết

Trong cơ chế quản lý bộ nhớ của hệ điều hành, một lượng vùng nhớ nhất định sẽ được cấp phát cho các tiến trình trên bộ nhớ chính để các tiền trình có thể được thực thi bởi CPU. Tùy vào các cơ chế nhất định mà các tiến trình sẽ được sử dụng bộ nhớ chính theo các cách khác nhau.
Paging là cơ chế chia bộ nhớ vật lý thành các khối bằng nhau gọi là **frame**, cũng như chia không gian địa chỉ luận lý (logical address space) thành các **page**. Hệ điều hành sẽ thực hiện ánh xạ các page và frame với nhau, từ đó cung cấp bộ nhớ cho từng tiến trình.
Cơ chế paging được thực hiện dựa trên nhiều thành phần khác nhau. Sau đây là mô tả các thành phần để hiện thực cơ chế paging.

2.2.1.1 The virtual memory mapping in each process

Vùng không gian bộ nhớ ảo được tổ chức như một ánh xạ bộ nhớ cho từng PCB của mỗi một tiến trình. Trong mỗi tiến trình, một bộ nhớ ảo được sử dụng để quản lý địa chỉ cho tiến trình đó. Các tiến trình không dùng chung không gian địa chỉ ảo. Không gian này có thể được chia thành nhiều vùng khác nhau gọi là **virtual memory area**. Từ góc nhìn của tiến trình, vùng địa chỉ ảo này bao gồm các vùng liên tục `vm_areas`. Trong mỗi area lại được chia ra thành các **memory region** nhỏ hơn. Các region có thể không liền kề nhau.
Muốn áp dụng cơ chế paging, ta cần chia các các memory area thành nhiều trang và có một bảng phân trang để quản lý các trang này. CPU tạo ra các địa chỉ, chúng đều có thể truy cập vào một trang chỉ định, được chia ra làm hai phần:

- **Page number:** Cho biết chỉ số trang của địa chỉ này, có thể dùng làm chỉ mục cho bảng phân trang để ánh xạ tới frame tương ứng trong bộ nhớ vật lý.

- **Page offset:** Cho biết vị trí của địa chỉ đó trong cùng 1 trang. Khi kết hợp với base address từ bảng phân trang sẽ tạo ra một địa chỉ vật lý.

2.2.1.2 The system physical memory

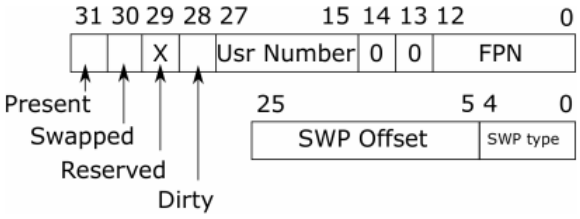
Bộ nhớ vật lý được chia ra làm 2 loại: Bộ nhớ RAM và bộ nhớ SWAP:

- RAM được CPU truy xuất trực tiếp và các tiến trình chỉ có thể được thực thi sau khi đã nằm trong RAM.
- SWAP là một thiết bị thứ cấp. Để có thể được chạy bởi CPU, các dữ liệu trong SWAP ứng với tiến trình đó phải được nạp lên RAM.

Trong thực tế, kích thước của RAM không đủ lớn để nạp frame cho mọi tiến trình. Do đó, cơ chế swapping được sử dụng để chuyển các frame qua lại giữa RAM và bộ nhớ vật lý khác (trong Bài tập lớn này sử dụng SWAP). Điều này giúp cho RAM có thể giải phóng nhiều frame cho các tiên trình khác, cũng như cho phép thực thi các chương trình có kích thước lớn hơn nhiều so với kích thước của RAM.

2.2.1.3 Paging-based address translation scheme

Cơ chế Dịch (translation) hỗ trợ cho cả segmentation và segmentation with paging. Mỗi tiến trình đều có một bảng phân trang để thực hiện thao tác ánh xạ từ địa chỉ luận lý sang địa chỉ vật lý (từ page tới các frame tương ứng). Cấu trúc của một entry của bảng phân trang trong Bài tập lớn như trong Hình 2.2 sau:



Hình 2.2: Cấu trúc entry của bảng phân trang

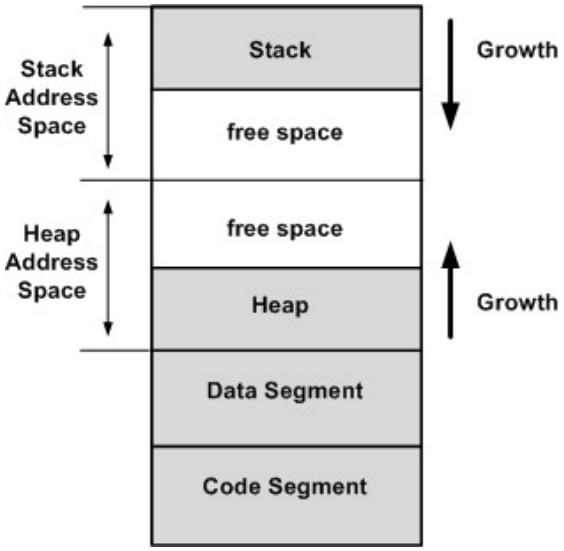
Ý nghĩa của cấu trúc trên hình như sau:

- Bit 31: present
- Bit 30: swapped
- Bit 29: reserved
- Bit 28: dirty
- Bits 15-27 user-defined numbering if present
- Bits 13-14 zero if present

- Bits 0-12 page frame number (FPN) if present
- Bits 25-5: swap offset if swapped
- Bits 4-0: swap type if swapped

2.2.1.4 Multiple memory segment

Phân đoạn bộ nhớ là một kỹ thuật trong đó bộ nhớ của hệ thống được chia thành các phần riêng biệt, mỗi phần được gọi là một "segment". Mỗi segment có thể chứa dữ liệu khác nhau tùy thuộc vào loại thông tin mà nó lưu trữ, ví dụ như mã chương trình, dữ liệu hoặc các ngăn xếp. Các phân đoạn này có thể được quản lý độc lập, giúp tối ưu hóa việc sử dụng bộ nhớ và tránh được các tình trạng lãng phí bộ nhớ. Các loại phân đoạn phổ biến bao gồm Code Segment (chứa mã chương trình), Data Segment (chứa dữ liệu toàn cục), Stack Segment (quản lý bộ nhớ cho các hàm và biến cục bộ), và Heap Segment (dùng cho việc cấp phát bộ nhớ động).



Hình 2.3: Phân đoạn bộ nhớ.

Khi một chương trình được nạp vào bộ nhớ, hệ điều hành sẽ phân bổ bộ nhớ cho các phân đoạn khác nhau. Mỗi phân đoạn này được xác định bởi hai yếu tố quan trọng: địa chỉ cơ sở và độ dài. Địa chỉ cơ sở là vị trí bắt đầu của phân đoạn trong bộ nhớ, trong khi độ dài xác định phạm vi bộ nhớ mà phân đoạn chiếm dụng. Nhờ vào việc sử dụng địa chỉ cơ sở và độ dài, hệ điều hành có thể dễ dàng quản lý bộ nhớ và điều phối các phân đoạn một cách hiệu quả.

Một đặc điểm quan trọng của kỹ thuật phân đoạn bộ nhớ là sự linh hoạt trong việc cấp phát và giải phóng bộ nhớ. Mỗi phân đoạn có thể được tải vào bộ nhớ ở các vị trí khác nhau mỗi khi chương trình chạy, giúp các chương trình có thể hoạt động trong các môi trường bộ nhớ hạn chế mà không gặp phải vấn đề tràn bộ nhớ hoặc sự cố thiếu bộ nhớ.

Việc sử dụng nhiều phân đoạn bộ nhớ mang lại nhiều lợi ích lớn cho hệ thống. Đầu tiên, nó giúp tối ưu hóa việc phân bổ bộ nhớ, đảm bảo rằng các phân đoạn có thể được sử dụng một cách hiệu quả mà không gây lãng phí tài nguyên. Thứ hai, việc chia bộ nhớ thành các phân đoạn độc lập giúp hệ thống dễ dàng bảo vệ bộ nhớ và ngăn ngừa các truy cập sai vào bộ nhớ, từ đó giảm thiểu nguy cơ gặp phải lỗi tràn bộ nhớ hay truy cập ngoài phạm vi.

Ngoài ra, việc sử dụng nhiều phân đoạn bộ nhớ cũng giúp chương trình hoạt động ổn định hơn. Ví dụ, nếu một phân đoạn bị lỗi, hệ thống có thể chỉ cần xử lý sự cố trong phân đoạn đó mà không ảnh hưởng đến các phân đoạn khác. Điều này đặc biệt quan trọng trong các hệ thống đa nhiệm, nơi mà các chương trình có thể chạy song song nhưng vẫn phải đảm bảo tính ổn định cho mỗi chương trình riêng biệt.

2.2.2 Trả lời câu hỏi

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:
Một vài ưu điểm của việc hiện thực thiết kế nhiều memory segments hay memory areas trong bài tập lớn này như sau:

- Tổ chức và quản lý bộ nhớ một cách hiệu quả: Bằng cách chia bộ nhớ thành nhiều phân đoạn, hệ thống có được một cách bố trí có cấu trúc và tổ chức hơn, cho phép các loại dữ liệu khác nhau (như mã lệnh, dữ liệu, stack, và heap) nằm trong các đoạn riêng biệt. Sự tách biệt này giúp quản lý và truy xuất dữ liệu thông tin dễ dàng, tránh xung đột và chồng chéo trong việc sử dụng bộ nhớ. Tối ưu hóa việc phân bổ bộ nhớ cho các tiến trình và giảm thiểu lãng phí bộ nhớ. Mỗi phân đoạn có thể được tối ưu hóa cho một mục đích cụ thể. Ví dụ, phân đoạn stack để xử lý các lời gọi hàm và biến cục bộ, trong khi phân đoạn heap hỗ trợ phân bổ bộ nhớ động. Việc phân đoạn giảm thiểu tình trạng phân mảnh và nâng cao hiệu suất sử dụng bộ nhớ.

- Các phân đoạn có thể được ánh xạ sang bộ nhớ vật lý với các địa chỉ khác nhau. Một tiến trình không cần phải được lưu trữ trong một khối liên tục của bộ nhớ vật lý. Nhờ vào khả năng ánh xạ này, hệ thống có thể sắp xếp một tiến trình tại các vị trí không liền kề, giúp tận dụng khoảng trống nhỏ trong bộ nhớ. Điều này giúp tối ưu việc sử dụng bộ nhớ và giảm thiểu lãng phí phân mảnh, đồng thời cho phép sử dụng hiệu quả hơn các vùng bộ nhớ vật lý hiện có.

- Tăng cường tính bảo mật và cô lập. Về tính bảo mật, các phân đoạn bộ nhớ có thể truy cập riêng, hạn chế truy cập vào các khu vực nhất định và tăng cường bảo mật. Về sự cô lập, giúp ngăn chặn truy cập trái phép hoặc thay đổi dữ liệu ngoài ý muốn giữa các phân đoạn, giúp bảo mật các khu vực nhạy ca,r của bộ nhớ.

Câu hỏi: What will happen if we divide the address to more than 2-levels in the paging memory management system?
Trả lời:
Trong hệ thống quản lý bộ nhớ sử dụng phân trang, khi địa chỉ được thiết kế thành hai cấp, hệ thống sẽ trở thành một mô hình phân trang nhiều cấp. Trong mô hình này, không gian bộ nhớ ảo được chia nhỏ thành các vùng quản lý riêng biệt. Khi một tiến trình cần truy cập bộ nhớ ảo, hệ thống sẽ xử lý địa chỉ qua từng cấp: từ cấp trên xác định vùng địa chỉ của cấp dưới, và tiếp tục cho đến khi tìm ra địa chỉ trong bộ nhớ vật lý.
Ưu điểm của phân trang đa cấp:

- Quản lý bộ nhớ hiệu quả hơn:** Việc chia địa chỉ thành nhiều cấp cho phép hệ thống tổ chức không gian địa chỉ tốt hơn, đặc biệt trong các hệ thống có dung lượng lớn. Mỗi cấp chỉ xử lý một phần nhỏ, giúp tối ưu việc phân bổ tài nguyên và tiết kiệm bộ nhớ.

- Giảm dung lượng lưu trữ bảng trang:** Không cần lưu toàn bộ bảng trang lớn trong bộ nhớ, chỉ các bảng cần thiết mới được nạp, giúp tiết kiệm tài nguyên bộ nhớ.

- 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 và cải thiện tốc độ truy xuất.

Nhược điểm của phân trang đa cấp:

- Phức tạp trong quản lý: Hệ thống phân trang nhiều cấp yêu cầu thuật toán và cấu trúc dữ liệu phức tạp hơn, dẫn đến tăng độ khó trong việc quản lý bộ nhớ.

- Chi phí tính toán cao: Việc duyệt qua từng cấp của bảng trang để xác định địa chỉ cần thiết có thể làm tăng thời gian trễ và chi phí tính toán, gây ảnh hưởng đến hiệu suất trong các hệ thống đòi hỏi tốc độ cao.

Câu hỏi: What is the advantage and disadvantage of segmentation with paging?

Trả lời:
Lý do để tách dữ liệu thành hai phân đoạn khác nhau, cụ thể là data segment và heap segment, nằm ở việc quản lý bộ nhớ hiệu quả.
Data segment được sử dụng cho việc cấp phát bộ nhớ tĩnh. Phân đoạn dữ liệu phát triển theo chiều hướng lên từ địa chỉ 0 đến địa chỉ bộ nhớ tối đa. Nó thường được sử dụng cho các biến toàn cục (global variables) và các biến tĩnh (static variables) được biết tại thời điểm biên dịch. Bằng cách cấp phát các biến này trong một phân đoạn riêng biệt, nó đảm bảo rằng việc sử dụng bộ nhớ có thể dự đoán được và dễ quản lý hơn.
Heap segment được sử dụng cho việc cấp phát bộ nhớ động và phát triển theo chiều hướng xuống từ địa chỉ bộ nhớ tối đa đến địa chỉ 0. Vùng heap được sử dụng cho các biến được cấp phát trong thời gian chạy, chẳng hạn như các biến được cấp phát bằng cách sử dụng `malloc()` và `free()` trong C. Việc tách bộ nhớ động vào một phân đoạn riêng biệt cho phép việc cấp phát và giải phóng bộ nhớ linh hoạt, thích ứng với nhu cầu của chương trình khi nó chạy.
Việc chia nhỏ bộ nhớ thành các phân đoạn khác nhau cho phép áp dụng các cơ chế quản lý khác nhau sao cho phù hợp với từng loại dữ liệu, giảm tình trạng lãng phí tài nguyên. Đồng thời, việc chia nhỏ giúp giảm thiểu phân mảnh nội và phân mảnh ngoại, giúp tối ưu hóa việc sử dụng bộ nhớ.
Ngoài ra, việc phân chia còn đáp ứng được thời gian sống của các biến trong từng phân đoạn riêng biệt. Biến trong data segmenet thường tồn tại trong toàn bộ thời gian thực thi chương trình, còn biến trong heap segment có thể được tạo và hủy bất cứ lúc nào trong suốt quá trình thực thi.
Việc phân chia này giúp hệ điều hành dễ dàng cấp phát, quản lý và truy cập bộ nhớ, vì nó có thể phân biệt rõ ràng giữa các biến tĩnh và động.

2.3 Put in all together (Synchronization)

2.3.1 Trả lời câu hỏi

Câu hỏi: What will happen 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: Do hệ điều hành sử dụng nhiều CPU, việc thiếu cơ chế đồng bộ sẽ gây ra các vấn đề nghiêm trọng như sau:

- **Data race**: Xảy ra khi nhiều tiến trình hoặc luồng cùng truy cập và thực hiện thao tác đọc/ghi trên một vùng dữ liệu đồng thời, dẫn đến kết quả dữ liệu không chính xác hoặc không nhất quán.
- **Race condition**: Tình huống trong đó kết quả của chương trình phụ thuộc vào thứ tự thực thi của các tiến trình hoặc luồng, gây ra kết quả không như mong đợi.
- **Deadlock** : Xảy ra khi các luồng hoặc tiến trình chờ đợi tài nguyên lẫn nhau, tạo thành một vòng tròn phụ thuộc và khiến hệ thống rơi vào trạng thái bế tắc, khiến các tiến trình không thể tiếp tục thực thi.

Chương 3

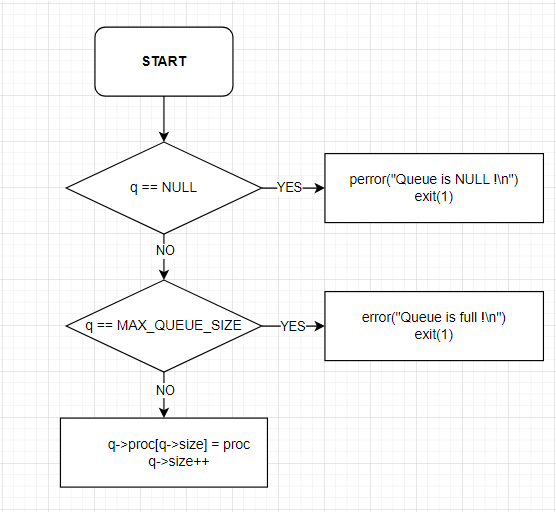
Hiện Thực Và Đánh giá.

3.1 Scheduler

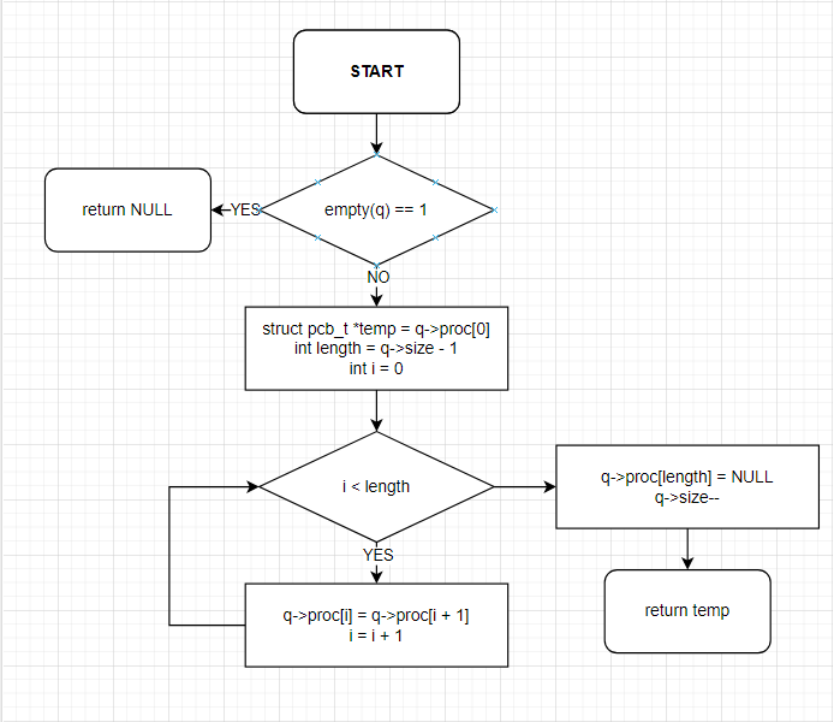
3.1.1 Hiện thực

```
1 struct queue_t {
2         struct pcb_t * proc[MAX_QUEUE_SIZE];
3         int size;
4 };
```

Đầu tiên, ta định nghĩa cấu trúc hàng đợi `queue_t` trong file `queue.h`. Cấu trúc này gồm hai thuộc tính, bao gồm mảng lưu trữ các tiến trình và số tiến trình hiện tại.



Hình 3.1: Enqueue: Thêm phần từ vào cuối hàng đợi.



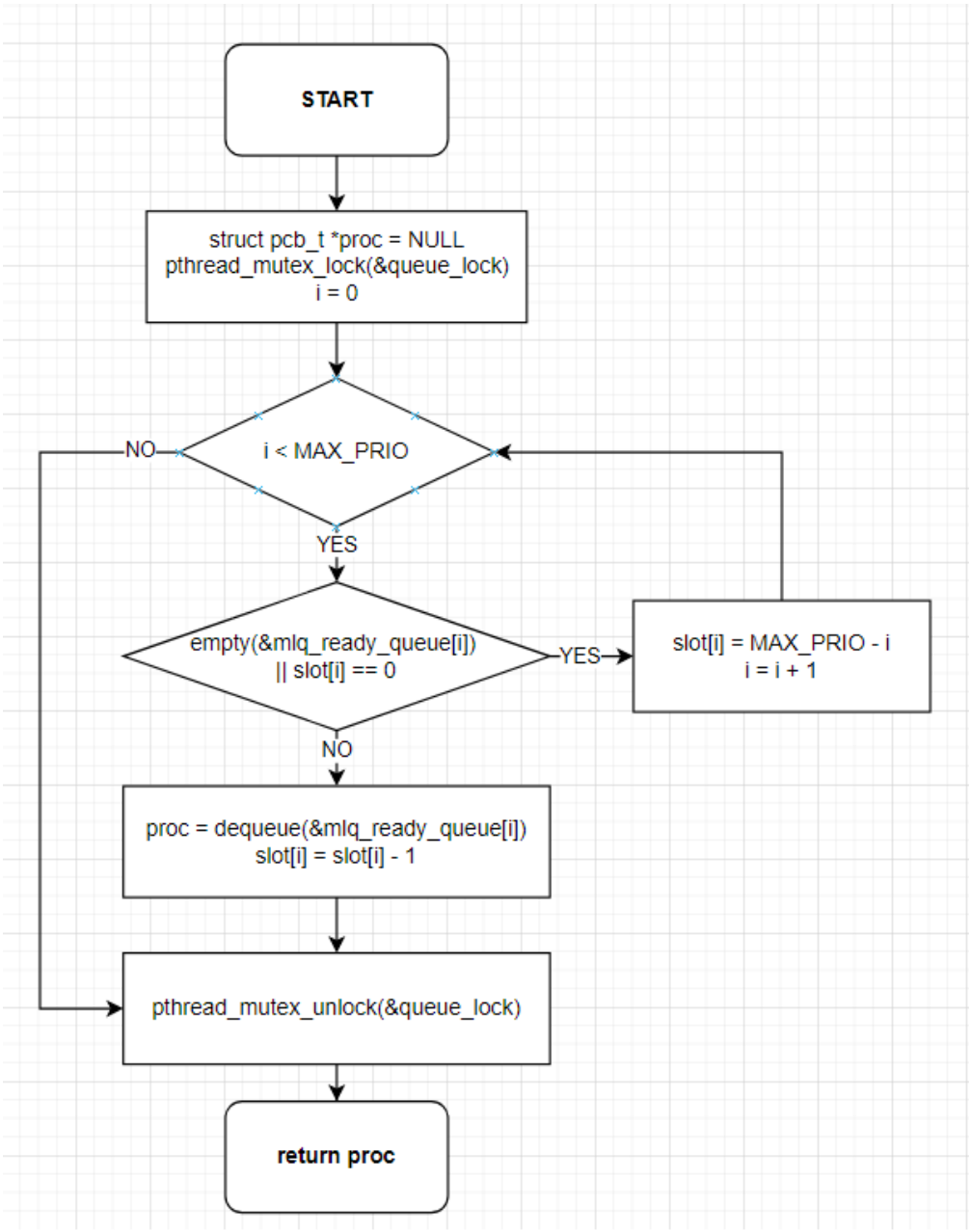
Hình 3.2: Dequeue: Lấy ra phần từ đầu tiên trong hàng đợi

Ta sử dụng hàm `enqueue()` để thêm 1 tiến trình (PCB) mới vào hàng đợi và `dequeue()` để lấy ra tiến trình tiếp theo từ hàng đợi. Hai hàm trên được hiện thực trong File `queue.c`.

```
1 static struct queue_t mlq_ready_queue[MAX_PRIO];
2 static int slot[MAX_PRIO];
```

Để định nghĩa một Scheduler sử dụng MLQ (Multi-level Queue) để quản lý các tiến trình theo thứ tự ưu tiên, ta sẽ định nghĩa một mảng các hàng đợi Queue để chứa các tiến trình, mỗi hàng đợi sẽ tương ứng với một mức độ ưu tiên. Ngoài ra, mảng Slot dùng để lưu trữ số lượng "slot" còn lại cho mỗi mức độ ưu tiên. Các tiến trình có độ ưu tiên càng cao thì sẽ có càng nhiều lượt sử dụng CPU càng nhiều.

Hàm `get_mlq_proc()` để lấy PCB của một tiến trình đang đợi trong hàng đợi ưu tiên thuộc mảng `mlq_ready_queue[MAX_PRIO]` và sử dụng hàm `dequeue()` để lấy ra tiến trình có độ ưu tiên tương ứng với queue chứa trong mảng và kết thúc. Trong trường hợp số "slot" cho mức ưu tiên đó đã hết, tức `slot == 0` hoặc hàng đợi ở mức ưu tiên hiện tại rỗng sẽ đặt lại giá trị `slot` cho mức ưu tiên đó và chuyển sang kiểm tra hàng đợi ở mức ưu tiên thấp hơn.



Hình 3.3: Get_mlq_proc: Lấy ra tiến trình có độ ưu tiên cao nhất.

3.1.2 Kết quả và đánh giá

Nhóm đã tiến hành chạy thử một số testcase nhằm kiểm tra việc hiện thực cơ chế Multilevel Queue (MLQ) cho bộ định thời.

3.1.2.1 File Sched

a/ Nội dung tập tin

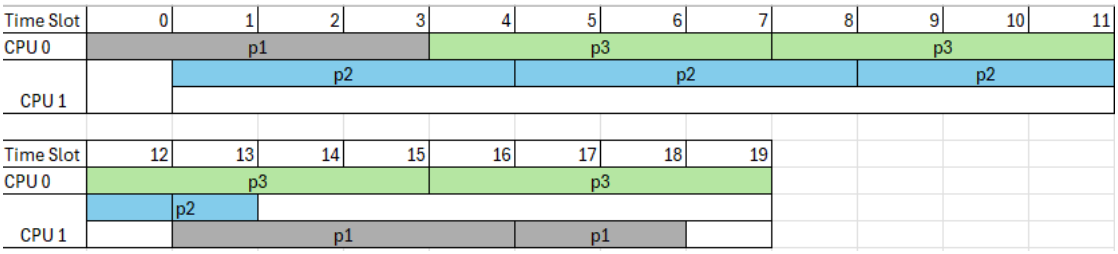
```
4 2 3
0 p1s 1
1 p2s 0
2 p3s 0
```

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
        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  12
        CPU 1: Put process  3 to run queue
        CPU 1: Dispatched process  3
```

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

Hình 3.4: Kết quả của FIle `sched`.



Hình 3.5: Gantt Chart quá trình định thời Sched.

3.1.2.2 File sched_0

a/ Nội dung tập tin

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

b/ Kết quả hiện thực

```
Time slot   0
ld_routine
        Loaded a process at input/proc/s0, PID: 1 PRIO: 1
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  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 3.6: Kết quả của FIle `sched_0`.

Time Slot	0	1	2	3	4	5	6	7	8	9	10	11
CPU 0	s0		s0		s1		s1		s1		s1	
											s0	

Time Slot	12	13	14	15	16	17	18	19	20	21	22
CPU 0	s0		s0		s0		s0		s0		s0

Hình 3.7: Gantt Chart quá trình định thời Sched.

3.2 Memory Management

3.2.1 Hiện thực

3.2.1.1 `mm-memphy.c`

`MEMPHY_dump`

```c
int MEMPHY_dump(struct memphy_struct *mp)
{
    /*TODO dump memphy contnt mp->storage
     *     for tracing the memory content
     */
    printf("----------------MEMORY CONTENT-------------- \n");
    printf("Address: Content \n");
    for (int i = 0; i < mp->maxsz; i++)
      if (mp->storage[i]) printf("0x%08x: %08x \n", i,
      ↪  mp->storage[i]);
    return 0;
}
```

Hàm `MEMPHY_dump` để in nội dung từ `mp->storage` ra màn hình để theo dõi nội dung bộ nhớ. Hàm này được sử dụng cho mục đích debug mỗi khi đọc hay ghi dữ liệu vào bộ nhớ.

3.2.1.2 `mm-vm.c`

`__alloc`

```c
int __alloc(struct pcb_t *caller, int vmaid, int rgid, int size,
            int *alloc_addr) {
  if (size <= 0)
    return -1;
  /*Khởi tạo một cấu trúc vùng nhớ*/
  struct vm_rg_struct rgnode;
  rgnode.vmaid = vmaid; /* TODO: commit the vmaid */
                        /* get_free_vmrg_area SUCCESS */
  if (get_free_vmrg_area(caller, vmaid, size, &rgnode) == 0) {
  ↪  // Tìm không gian vùng nhớ trống có kích thước size
    caller->mm->symrgtbl[rgid].rg_start = rgnode.rg_start;
    caller->mm->symrgtbl[rgid].rg_end = rgnode.rg_end;

    caller->mm->symrgtbl[rgid].vmaid = rgnode.vmaid;

    *alloc_addr = rgnode.rg_start;

    return 0;
  }

```

```
20  /* get_free_vmrg_area FAILED handle the region management
    ↪  (Fig.6)*/
21
22  /* TODO retrive current vma if needed, current comment out due
    ↪  to compiler
23   * redundant warning*/
24  /*Attempt to increate limit to get space */
25  struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm,
    ↪  vmaid); // Lấy vùng nhớ hiện tại
26  int inc_sz = PAGING_PAGE_ALIGNSZ(size);
    ↪  // Tính kích thước cần mở rộng thê,
27  int inc_limit_ret;
    ↪  // Lưu trữ kết quả tăng giưới hạn
28
29  /* TODO retrive old_sbrk if needed, current comment out due to
    ↪  compiler
30   * redundant warning*/
31  int old_sbrk = cur_vma->sbrk;
    ↪  // sbrk cũ của VMA
32
33  /* TODO INCREASE THE LIMIT
34   * inc_vma_limit(caller, vmaid, inc_sz)
35   */
36  /*FAILED increase limit */
37  if (inc_vma_limit(caller, vmaid, inc_sz, &inc_limit_ret) < 0) {
    ↪  // Mở rộng giới hạn VMA
38    printf("Increase the vma_limit failed ! \n");
39    return -1;
40  }
41  /*SUCCESSFUL increase limit */
42
43  /* TODO: commit the limit increment */
44  unsigned long new_sbrk = old_sbrk + size;
45  if (new_sbrk < cur_vma->vm_end) {
46    struct vm_rg_struct *new_free_rg =
47      init_vm_rg(new_sbrk, cur_vma->vm_end, vmaid);
       ↪  // init_vm_rg : cấp phát một vùng nhớ trống (mm.c)
48      enlist_vm_freerg_list(caller->mm, *new_free_rg);
       ↪  // enlist_vm_freerg_list: add new rg to freerg_list
49  }
50  /* TODO: commit the allocation address
51  // *alloc_addr = ...
```

```
52   */
53    caller->mm->symrgtbl[rgid].rg_start = old_sbrk;
54    caller->mm->symrgtbl[rgid].rg_end = new_sbrk;
55    caller->mm->symrgtbl[rgid].vmaid = vmaid;
56    *alloc_addr = old_sbrk;
57
58    return 0;
59  }
```

Hàm `__alloc` dùng để cấp phát các vùng nhớ (region) trong DATA hoặc HEAP, có ba trường hợp cần cấp phát.

- Thứ nhất nếu trong area có free region đủ lớn để cấp phát thì tìm kiếm region trống có sẵn trong danh sách chờ của `vm_area` vùng nhớ tương ứng (data / heap) để cấp phát bằng cách gọi hàm `get_free_vmrg_area` .
- Thứ hai là trong `vm_area` không có free region thích hợp, ta mở rộng con trỏ `sbrk` , nếu vùng nhớ từ `sbrk` cũ đến `sbrk` mới không vượt quá `end` của area, cấp phát vùng nhớ đó và trả về địa chỉ vùng nhớ vừa được cấp phát.
- Cuối cùng nếu con trỏ `sbrk` mới vượt quá end, thì phải mở rộng area thêm phù hợp kích thước cần dùng, nếu mở rộng thành công thì sẽ cấp phát region mới.

Chế độ Go Down: Nếu cấp phát cho vùng nhớ HEAP thì giảm sbrk thay thì tăng, và chỉnh sửa một số thao tác phù hợp với việc mở rộng bộ nhớ đi xuống.
`__free`

```
1  int __free(struct pcb_t *caller, int rgid) {
2    struct vm_rg_struct rgnode;
3
4    // Dummy initialization for avoding compiler dummay warning
5    // in incompleted TODO code rgnode will overwrite through
     ↪  implementing
6    // the manipulation of rgid later
7    // rgnode.vmaid = 0; // dummy initialization
8    // rgnode.vmaid = 1; // dummy initialization
9    /*Kiểm tra tính hợp lệ của đầu vào*/
10   if (rgid < 0 || rgid > PAGING_MAX_SYMTBL_SZ) {
11     printf("ERROR: invalid region ID.\n");
```

```
    return -1;
  }
  if (!caller || !(caller->mm)) {
    printf("ERROR: invalid PCB or memory management structure.");
    return -1;
  }
  /* TODO: Manage the collect freed region to freerg_list */
  // Đảm bảo region tồn tại để remove
  struct vm_rg_struct *sym_rg = &(caller->mm->symrgtbl[rgid]);
  if (!sym_rg) {
    printf("ERROR: Region ID %d is not allocated.\n", rgid);
    return -1;
  }
  rgnode.rg_start = sym_rg->rg_start;
  rgnode.rg_end = sym_rg->rg_end;

  /*enlist the obsoleted memory region */
  if (enlist_vm_freerg_list(caller->mm, rgnode) < 0) {
    printf("ERROR: Failed to enlist the free region.\n ");
    return -1;
  }
  return 0;
}
```

Hàm `__free` dùng để giải phóng không gian lưu trữ liên kết với khu vực cụ thể (được xác định bởi `rgid`) trong vùng nhớ ảo có ID là `vmaid` của tiến trình. Quá trình thực hiện bao gồm khởi tạo một đối tượng region chứa thông tin về khu vực cần xóa, xóa thông tin của region này khỏi mảng `symrgtbl`, sau đó thêm region vừa xóa vào danh sách các region trống trong area tương ứng. Tuy nhiên, phương pháp này không giải phóng được các frame đã ánh xạ của region, dẫn đến việc bộ nhớ sử dụng ngày càng tăng và chưa thể ứng dụng hiệu quả trong thực tế.
`get_vm_area_node_at_brk`

```
struct vm_rg_struct* get_vm_area_node_at_brk(struct pcb_t *caller,
↪  int vmaid, int size, int alignedsz)
{
  struct vm_rg_struct *newrg;
  /* TODO retrive current vma to obtain newrg, current comment out
  ↪  due to
   * compiler redundant warning*/
```

```
  struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm,
  ↪  vmaid);
  newrg = malloc(sizeof(struct vm_rg_struct));

  newrg->rg_start = cur_vma->sbrk;
  newrg->vmaid = vmaid;
  newrg->rg_next = NULL;

// #ifdef MM_PAGING_HEAP_GODOWN
//   if (vmaid == 1) {
//     newrg->rg_end = cur_vma->sbrk - size;
//   } else {
//     newrg->rg_end = cur_vma->sbrk + size;
//   }
// #else
//   newrg->rg_end = cur_vma->sbrk + size;
// #endif
  newrg->rg_end = cur_vma->sbrk + size;

  return newrg;
}
```

`validate_overlap_vm_area`

```
int validate_overlap_vm_area(struct pcb_t *caller, int vmaid, int
↪  vmastart, int vmaend)
{
  struct vm_area_struct *vma = caller->mm->mmap;

  /* TODO validate the planned memory area is not overlapped */
  while(vma){
    if(vma->vm_id  != vmaid && vma->vm_start < vmaend &&
    ↪  vma->vm_end > vmastart)
    return -1;
    else vma=vma->vm_next;
  }
  /* END TODO*/

  return 0;
}
```

Hàm `get_vm_area_node_at_brk` thực hiện kiểm tra bằng cách lấy thông tin

vùng nhớ từ memory map của `mm` và duyệt qua tất cả các vùng nhớ ảo (vma) bằng vòng lặp `while`. Nếu địa chỉ bắt đầu và kết thúc của một vma trùng nhau, vma này sẽ bị bỏ qua vì đây là vùng nhớ trống. Đối với các vùng nhớ khác, hàm sẽ kiểm tra vùng nhớ free region liền kề với vùng nhớ đó và so sánh khoảng địa chỉ của vùng nhớ mới với các vma hiện có bằng cách kiểm tra xem có free region nào kết thúc ở giá trị `vmastart` hoặc bắt đầu ở `vmaend`, nếu có hàm sẽ merge hai vùng nhớ đó lại, giải quyết vấn đề phân mảnh ngoại. Nếu có sự chồng chéo, hàm trả về `-1`. Ngược lại, quá trình kiểm tra sẽ tiếp tục cho đến khi tất cả các vma được duyệt qua. `find_victim_page`

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

  /* TODO: Implement the theorical mechanism to find the victim
   ↪  page */
  if (!pg) {
    return -1;
  }
  if (!pg->pg_next) {
    *retpgn = pg->pgn;
    free(pg);
    mm->fifo_pgn = NULL;
  }
  else {
    struct pgn_t *pre = mm->fifo_pgn;
    while (pre->pg_next->pg_next) {
      pre = pre->pg_next;
      pg = pg->pg_next;
      }

    pg = pg->pg_next;
    pre->pg_next = NULL;
    *retpgn = pg->pgn;
    free(pg);
  }
  return 0;
}
```

Hàm `find_victim_page` được sử dụng để tìm trang (page) cần thay thế (victim page) trong quá trình quản lý bộ nhớ của một tiến trình, dựa trên thuật toán FIFO

(First In, First Out). Đầu tiên, hàm lấy một node từ danh sách các số hiệu trang (page number) theo thứ tự FIFO. Nếu danh sách trống, hàm trả về `-1`. Tiếp theo, hàm sử dụng vòng lặp `while` để duyệt qua từng node trong danh sách, theo dõi node trước (`prev`) và node hiện tại (`pg`). Khi vòng lặp kết thúc, số hiệu trang của node cuối cùng (`pgn`) sẽ được gán vào biến `retpgn` (return page number). Sau đó, hàm xóa node cuối khỏi danh sách bằng cách đặt con trỏ `next` của node trước (`prev`) thành `NULL`.

3.2.1.3 `mm.c`

`vmap_page_range`

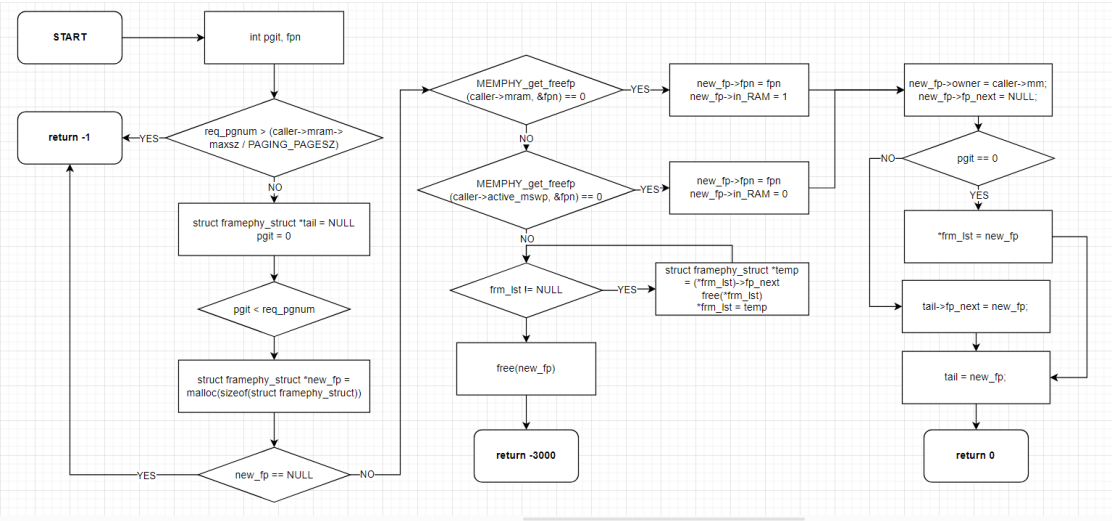
```
int vmap_page_range(struct pcb_t *caller,
                    int addr,
                    int pgnum,
                    struct framephy_struct *frames,
                    struct vm_rg_struct *ret_rg)
{
    // uint32_t * pte = malloc(sizeof(uint32_t));
  struct framephy_struct *fpit = malloc(sizeof(struct
  ↪  framephy_struct));
  // int  fpn;
  int pgit = 0;
  int pgn = PAGING_PGN(addr);

  /* TODO: update the rg_end and rg_start of ret_rg */
  ret_rg->rg_start = addr;
  ret_rg->rg_end = addr + (pgnum * PAGING_PAGESZ);
  ret_rg->vmaid = 0;

  /* TODO map range of frame to address space
   *      in page table pgd in caller->mm
   */
  for (; pgit < pgnum; pgit++) {
    if (fpit == NULL) {
      return -1; // Error: insufficient frames
    }
    uint32_t *pte = &caller->mm->pgd[pgn + pgit];
    pte_set_fpn(pte, fpit->fpn);
    fpit = fpit->fp_next;
    enlist_pgn_node(&caller->mm->fifo_pgn, pgn + pgit);
  }
```

```
30   return 0;
31 }
```

Hàm `vmap_page_range` dùng để ánh xạ một dải trang vào bảng trang của tiến trình (sử dụng danh sách các frame đã được cấp phát từ hàm `alloc_pages_range`) bao gồm các bước sau:

- Xác định thông tin vùng nhớ: Xác định vị trí bắt đầu và kết thúc của vùng nhớ, đồng thời tính toán số lượng page cần ánh xạ vào vùng nhớ (region).
- Ánh xạ từng page vào frame: Mỗi page sẽ được ánh xạ vào các frame tương ứng trong danh sách frame đã được cấp phát. Đồng thời, tạo một mục PTE (Page Table Entry) cho từng page.
- Cập nhật quản lý page: Sau khi ánh xạ hoàn tất, số lượng page đã được ánh xạ sẽ được thêm vào danh sách `fifo_pgn` để quản lý.

`alloc_pages_range`



Hình 3.8: alloc_pages_range: cấp phát phạm vi các pages cho một tiến trình

```
1 int alloc_pages_range(struct pcb_t *caller, int req_pgnum, struct
  ↪  framephy_struct** frm_lst)
2 {
3   int pgit, fpn;// fpn: số khung trang
```

```
4   if(req_pgnum > (caller->mram->maxsz / PAGING_PAGESZ))// kích
    ↪  thước tối đa của bộ nhớ
5     return -1;
6   /* TODO: allocate the page
7   //caller-> ...
8   //frm_lst-> ...
9   */
10  struct framephy_struct *tail = NULL;
11
12  //Vong lap qua tung trang yeu cau
13    for (pgit = 0; pgit < req_pgnum; pgit++) {
14        struct framephy_struct *new_fp = malloc(sizeof(struct
          ↪  framephy_struct));
15        if (!new_fp)
16            return -1;
17
18        //Cap phat mot khung trang trong tu ram
19        if (MEMPHY_get_freefp(caller->mram, &fpn) == 0) {
20            new_fp->fpn = fpn;
21            new_fp->in_RAM = 1; // Khung này trong RAM
22        } else if (MEMPHY_get_freefp(caller->active_mswp, &fpn) ==
          ↪  0) { // Neu khong co khung trang trong tu RAM, vao
          ↪  vungf swap
23            new_fp->fpn = fpn;
24            new_fp->in_RAM = 0; // Khung này trong SWAP
25        } else {
26            // Neu khong du bo nho, giai phong toan bo du lieu da
              ↪  cap phat trc do
27            while (*frm_lst) {
28                struct framephy_struct *temp =
                  ↪  (*frm_lst)->fp_next;
29                free(*frm_lst);
30                *frm_lst = temp;
31            }
32            free(new_fp);
33            return -3000; // Out of memory
34        }
35
36        // Lien ket khung trang vao danh sach
37        new_fp->owner = caller->mm;
38        new_fp->fp_next = NULL;
39
```

```
        if (pgit == 0) {  // Nếu là khung đầu tiên
            *frm_lst = new_fp;
        } else {          // Thêm vào cuối danh sách
            tail->fp_next = new_fp;
        }
        tail = new_fp;

    }
  return 0;
}
```

Hàm `alloc_pages_range` dùng để cấp phát một số lượng trang từ RAM: có 2 trường hợp:

- Có frame trống ở bộ nhớ RAM, sẽ thêm số lượng frame trống đó vào danh sách cấp phát
- Hết frame trống: tìm các victim page để có thể swap frame giữa bộ nhớ RAM với SWAP để lấy đủ số frame ở RAM, nếu không đủ sẽ trả về lỗi -3000.

`init_mm`

```
int init_mm(struct mm_struct *mm, struct pcb_t *caller) {
  /*Khởi tạo hai vùng nhớ*/
  struct vm_area_struct *vma0 = malloc(sizeof(struct
  ↪  vm_area_struct));
  struct vm_area_struct *vma1 = malloc(sizeof(struct
  ↪  vm_area_struct));
  /*Khởi tạo bảng trang*/
  mm->pgd = malloc(PAGING_MAX_PGN * sizeof(uint32_t));

  if (!vma0 || !mm->pgd || !vma1)
    return -1;

  for (int i = 0; i < PAGING_MAX_PGN; i++) {
    mm->pgd[i] = 0;
  }

  /* By default the owner comes with at least one vma for DATA */
  vma0->vm_id = 0;
  vma0->vm_start = 0;
  vma0->vm_end = vma0->vm_start;
  vma0->sbrk = vma0->vm_start;
  /*Khởi tạo và thêm vùng nhớ tự  dođầu vào VMA0*/
```

```
21    struct vm_rg_struct *first_rg = init_vm_rg(vma0->vm_start,
      ↪  vma0->vm_end, 0);
22    enlist_vm_rg_node(&vma0->vm_freerg_list, first_rg);
23
24    /*Thiết lập cho heap-VMA1*/
25    vma1->vm_id = 1;
26  #ifdef MM_PAGING_HEAP_GODOWN
27    vma1->vm_start = caller->vmemsz;
28    vma1->vm_end = vma1->vm_start;
29    vma1->sbrk = vma1->vm_start;
30  #endif
31    /*Khởi tạo và thêm vùng nhớ tự  dođầu vào VMA1*/
32    struct vm_rg_struct *heap_rg = init_vm_rg(vma1->vm_start,
      ↪  vma1->vm_end, 1);
33    enlist_vm_rg_node(&vma1->vm_freerg_list, heap_rg);
34
35    /*Liên kết VMA1 và VMA0*/
36    vma0->vm_next = vma1;
37    vma1->vm_next = NULL;
38
39    /* Point vma owner backward */
40    vma0->vm_mm = mm;
41    vma1->vm_mm = mm;
42
43    /* TODO: update mmap */
44    mm->mmap = vma0;
45
46    /* Thiết lập danh sách trang trống */
47    mm->fifo_pgn = NULL;
48    return 0;
49  }
```

inc_vma_limit

```
1   int inc_vma_limit(struct pcb_t *caller, int vmaid, int inc_sz,
2                     int *inc_limit_ret) {
3     struct vm_rg_struct *newrg = malloc(sizeof(struct
      ↪  vm_rg_struct));
4     int inc_amt = PAGING_PAGE_ALIGNSZ(inc_sz);
      ↪  // Kích thước vùng nhớ mới cần cấp phát
5     int incnumpage = inc_amt / PAGING_PAGESZ;
6     struct vm_rg_struct *area =
```

```
7       get_vm_area_node_at_brk(caller, vmaid, inc_sz, inc_amt);
8   struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm,
    ↪  vmaid);
9
10  int old_end = cur_vma->vm_end;
11
12  /*Validate overlap of obtained region */
13  if (validate_overlap_vm_area(caller, vmaid, area->rg_start,
    ↪  area->rg_end) < 0) {
14    printf("Overlaped vm_area regions\n");
15    free(newrg);
16    return -1; /*Overlap and failed allocation */
17  }
18
19 // #ifdef MM_PAGING_HEAP_GODOWN
20 //   if (cur_vma->vm_id == 1) {
21 //     cur_vma->sbrk -= inc_sz;
22 //     if (area->rg_end < cur_vma->vm_end) {
23 //       cur_vma->vm_end -= inc_amt;
24 //     }
25 //   } else {
26 //     cur_vma->sbrk += inc_sz;
27 //     if (area->rg_end > cur_vma->vm_end) {
28 //       cur_vma->vm_end += inc_amt;
29 //     }
30 //   }
31 // #else
32 //   cur_vma->sbrk += inc_sz;
33 //   if (area->rg_end > cur_vma->vm_end) {
34 //     cur_vma->vm_end += inc_amt;
35 //   }
36 // #endif
37 //   *inc_limit_ret = cur_vma->vm_end;
38
39 //   if (vm_map_ram(caller, area->rg_start, area->rg_end, old_end,
    ↪  incnumpage,
40 //                  newrg) < 0) {
41 //     printf("Failed mapping memory to RAM\n");
42 //     free(newrg);
43 //     return -1; /* Map the memory to MEMRAM */
44 //   }
45
```

```
//   printf("Increased vm_area %d\n", vmaid);
  // return *inc_limit_ret;


  cur_vma->vm_end += inc_sz;
  cur_vma->sbrk += inc_sz;

  if (vm_map_ram(caller, area->rg_start, area->rg_end, old_end,
   ↪  incnumpage , newrg) < 0)
    return -1; /* Map the memory to MEMRAM */
  return 0;
}
```

enlist_vm_freerg_list

```
int enlist_vm_freerg_list(struct mm_struct *mm, struct
 ↪  vm_rg_struct *rg_elmt)
{
// struct vm_rg_struct *rg_node = mm->mmap->vm_freerg_list;
  struct vm_rg_struct *rg_node;
  if(rg_elmt->vmaid == 0)
    rg_node = mm->mmap->vm_freerg_list;
  else if(rg_elmt->vmaid == 1)
    rg_node = mm->mmap->vm_next->vm_freerg_list;

#ifdef MM_PAGIMG_HEAP_GODOWN
  if(rg_elmt->vmaid == 0)
    if (rg_elmt->rg_start >= rg_elmt->rg_end)
      return -1;
  if(rg_elmt->vmaid == 1) // HEAP
    if (rg_elmt->rg_start <= rg_elmt->rg_end){
      return -1;
    }

#else
   if (rg_elmt->rg_start >= rg_elmt->rg_end)
    return -1;
#endif
  //when enlist a rg continous to another rg, should merge those 2
  // struct vm_rg_struct *rgit = mm->mmap->vm_freerg_list;
  struct vm_rg_struct *rgit;
  if(rg_elmt->vmaid == 0)
```

```
    rgit = mm->mmap->vm_freerg_list;
  else if(rg_elmt->vmaid == 1)
    rgit = mm->mmap->vm_next->vm_freerg_list;
  while(rgit!= NULL){
    if(rgit->rg_start != rgit->rg_end){
      if(rgit->rg_start == rg_elmt->rg_end){
        rgit->rg_start = rg_elmt->rg_start;
        return 0;
      }
      else if(rgit->rg_end == rg_elmt->rg_start){
        rgit->rg_end = rg_elmt->rg_end;
        return 0;
      }
    }
    rgit = rgit->rg_next;
  }

  if (rg_node != NULL)
    rg_elmt->rg_next = rg_node;

  /* Enlist the new region */
  // mm->mmap->vm_freerg_list = rg_elmt;

  if(rg_elmt->vmaid == 0)
    mm->mmap->vm_freerg_list = rg_elmt;
  else if(rg_elmt->vmaid == 1)
    mm->mmap->vm_next->vm_freerg_list = rg_elmt;
  return 0;
}
```

3.2.2 Kết quả và đánh giá

3.3 Put in all together (Synchronization)

Chương 4

Kết Luận.

Hệ điều hành đóng vai trò vô cùng quan trọng trong các máy tính đa dụng (general-purpose computer) cũng như máy tính cá nhân (personal computer). Trong bài tập lớn này, nhóm đã có cơ hội hiện thực hóa một hệ điều hành đơn giản với các tính năng cơ bản như quản lý bộ nhớ, quản lý tiến trình và lập lịch. Sử dụng ngôn ngữ lập trình C cùng các khái niệm cơ bản của hệ điều hành, nhóm đã xây dựng một môi trường thử nghiệm giúp hiểu rõ hơn về cách thức hoạt động của một hệ điều hành thực tế.

Nhóm đã đối mặt với nhiều thử thách khi hiện thực hóa hệ điều hành đơn giản, nhưng những khó khăn đó đã giúp củng cố kiến thức về các nguyên lý cơ bản của hệ điều hành và quen thuộc hơn với quy trình phát triển hệ điều hành. Đồng thời, nhóm cũng có cơ hội áp dụng kiến thức lý thuyết vào thực tế và hiểu sâu hơn về cách thức hoạt động bên trong của một hệ điều hành.

Tuy nhiên, nhóm vẫn gặp một số hạn chế và sai sót trong quá trình hiện thực hóa, cũng như trong việc phân bổ thời gian để hoàn thành bài tập.

Tóm lại, sau khi hoàn thành bài tập lớn, nhóm đã tích lũy được nhiều kiến thức về hệ điều hành và cách hiện thực một hệ điều hành đơn giản, đồng thời có thêm kinh nghiệm để phát triển trong tương lai.

Tài liệu tham khảo

[1] Abraham Silberschatz, Peter Baer Galvin, Greg Gagne (04/05/2018), *Operating System Concept, Tenth Edition*.

[2] GeeksforGeeks (17/02/2023), *Paged Segmentation and Segmented Paging*. Truy cập tại https://www.geeksforgeeks.org/paged-segmentation-and-segmented-paging/.

[3] GeeksforGeeks (05/05/2023), *Multilevel Queue (MLQ) CPU Scheduling*. Truy cập tại https://www.geeksforgeeks.org/multilevel-queue-mlq-cpu-scheduling/.

[4] GeeksforGeeks (19/01/2024), *Virtual Memory in Operating System*. Truy cập tại https://www.geeksforgeeks.org/virtual-memory-in-operating-system/.

[5] GeeksforGeeks (30/04/2024), *Segmentation in Operating System*. Truy cập tại https://www.geeksforgeeks.org/segmentation-in-operating-system/.