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


# HỆ ĐIỀU HÀNH (CO2017)

**Bài Tập Lớn**

# Simple Operating System

Giảng viên hướng dẫn: Hoàng Lê Hải Thanh
Sinh viên: Ngô Quang Anh - 2252029.
Trần Ngọc Khánh Huy - 2252265.
Hoàng Nghĩa Hiếu - 2052989.
Nguyễn Quốc Qui - 2053383.

# Mục lục

# 1 Danh sách thành viên & Đóng góp

| Số thứ tự | Họ và tên | Mã số sinh viên | Phần trăm |
|---|---|---|---|
| 1 | Trần Ngọc Khánh Huy | 2252265 | 100% |
| 2 | Ngô Quang Anh | 2252029 | 100% |
| 3 | Hoàng Nghĩa Hiếu | 2052989 | 100% |
| 4 | Nguyễn Quốc Qui | 2053383 | 50% |

# 2 Giới thiệu

Bài tập lớn lần này là thiết kế một hệ điều hành đơn giản bao gồm hai thành phần chính:

- Scheduler (and Dispatcher) : Giải thuật định thời xác định process được thực hiện trên CPU chỉ định.
- Virtual memory engine (VME): Phân cách các không gian bộ nhớ của mỗi tiến trình với nhau. Bộ nhớ vật lý (RAM) được chia sẻ bởi nhiều tiến trình nhưng mỗi tiến tình không biết tới sự tồn tại của tiến trình khác. Việc này được thực hiện bằng cách mỗi tiến trình có không gian bộ nhớ ảo riêng của nó và động cơ bộ nhớ ảo sẽ ánh xạ và dịch các địa chỉ ảo do các tiến trình cung cấp thành các địa chỉ vật lý tương ứng.

# 3 Scheduler

Đối với mỗi chương trình mới, bộ nạp sẽ tạo một tiến trình mới và gán một PCB (Process Control Block - Bộ quản lý tiến trình) cho nó. Bộ nạp sau đó đọc và sao chép nội dung của chương trình vào đoạn văn bản của tiến trình mới (được trỏ bởi con trỏ mã trong PCB của tiến trình). PCB của tiến tình được đẩy vào **ready-queue** liên quan có cùng mức ưu tiên với giá trị **prio** của tiến trình này. Sau đó nó chờ được cấp phát CPU. CPU chạy các tiến trình theo thuật toán **Round-Robin**. Mỗi tiến trình được phép chạy trong **time slice**. CPU sau đó chọn một tiến trình khác từ hàng đợi sãn sàng và tiếp tục

Multi-level queue scheduling là một cấu trúc giải thuật để sắp xếp các tác vụ dựa trên mức độ ưu tiên của các tác vụ ấy và cung cấp tài nguyên tương ứng. Để thiết kế được scheduler thì cần phải có những hiểu biết về các thành phần liên quan đến Multi-level queue (MLQ) scheduling gồm:

- Hiện thực queue : Gồm có các hàm enqueue, dequeue
- Priority và số slot trong queue dựa trên priority

Thông qua bài tập lớn này, chúng ta sẽ hiểu rõ hơn về nguyên lý hoạt động của giải thuật này và tác dụng của nó trong Hệ điều hành.

**Câu hỏi: Lợi ích của giải thuật lập lịch được sử dụng trong bài tập lớn này so với các thuật toán lập lịch khác mà bạn đã học là gì ?**

Về giải thuật **First Come First Served (FCFS)**

- Do sử dụng chiến lược không ưu tiên (nonpreemptive), nếu một tiến trình bắt đầu, CPU sẽ thực thi tiến trình đó cho đến khi hoàn thành.
- Nếu một tiến trình có thời gian CPU dài, các tiến trình phía sau (có thể là các tiến trình với thời gian CPU ngắn) trong hàng đợi phải chờ rất lâu → Không công bằng đối với các tiến trình.

Về giải thuật **Shortest Job First (SJF)**

- Cần ước tính trước thời gian CPU tiếp theo của tiến trình, điều này thường không khả thi trong thực tế.

- Các tiến trình có thời gian CPU dài sẽ phải chờ lâu hơn hoặc bị trì hoãn vô thời hạn khi nhiều tiến trình với thời gian CPU ngắn đồng thời vào hàng đợi → đói tài nguyên (starvation).

Về giải thuật **Round Robin (RR)**

- Nếu thời gian lượng tử (time quantum) quá lớn: RR sẽ giống FCFS.
- Nếu thời gian lượng tử quá nhỏ, việc chuyển đổi ngữ cảnh của CPU sẽ tăng lên nhiều, gây ra chi phí lớn, giảm hiệu quả sử dụng CPU.

Về giải thuật **Priority Scheduling (PS-thường)**:

- Phải có bộ lập lịch cho các tiến trình có độ ưu tiên bằng nhau.
- Các tiến trình có độ ưu tiên thấp hơn có thể không có cơ hội thực thi → đói tài nguyên (starvation).

So vơi các giải thuật định thời được liệt kê ở trên, thuật toán **Multi-Level Queue (MLQ)** có những ưu điểm sau:

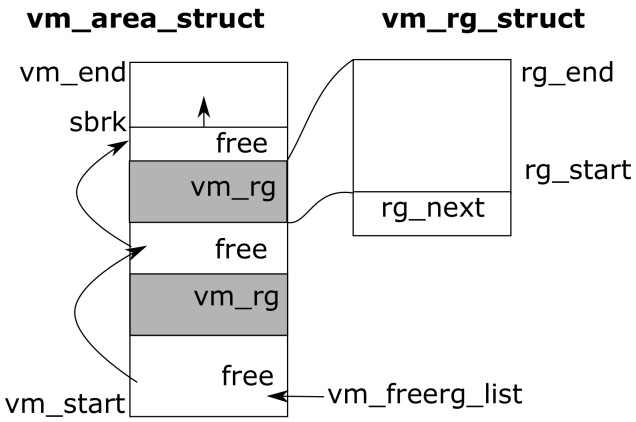
- Thời gian phản hồi: Vì các tiến trình có cùng độ ưu tiên được đặt trong cùng một hàng đợi, các tiến trình có độ ưu tiên cao sẽ được xử lý trước.
- Công bằng: Đảm bảo công bằng cho tất cả các tiến trình trong cùng một hàng đợi bằng cách áp dụng kiểu Round Robin.
- Không đói tài nguyên: Mỗi hàng đợi chỉ có một khoảng thời gian cố định để sử dụng CPU (MAX_PRIO - prio) và khi khoảng thời gian này hết, hệ thống phải chuyển tài nguyên sang tiến trình khác trong hàng đợi kế tiếp.

# 4 Memory Management

## 4.1 The virtual memory mapping in each process

Không gian bộ nhớ ảo được tổ chức dưới dạng một bản đồ bộ nhớ cho mỗi PCB của quá trình. Từ quan điểm của quá trình, địa chỉ ảo bao gồm nhiều `vm_areas` (liền kề). Trong thực tế, mỗi vùng có thể hoạt động như là mã nguồn, ngăn xếp hoặc heap. Do đó, quá trình giữ trong pcb của nó một con trỏ của nhiều vùng nhớ liền kề.

Mỗi vùng bộ nhớ dao động liên tục trong [`vm_start`,vm end]. Dù không gian có trải dài toàn bộ phạm vi, diện tích sử dụng thực tế bị giới hạn bởi đỉnh chỉ vào sbrk. Trong khu vực giữa `vm_start` và sbrk, có nhiều vùng được sử dụng bởi struct `vm_rg_struct` và theo dõi các vị trí trống theo danh sách `vm_freerg_list`. Thông qua thiết kế này, chúng ta sẽ chỉ thực hiện việc phân bổ bộ nhớ vật lý thực tế trong diện tích sử dụng được, qua mô hình sau:




Hình 1: Cấu trúc của vùng bộ nhớ ảo và vùng nhớ

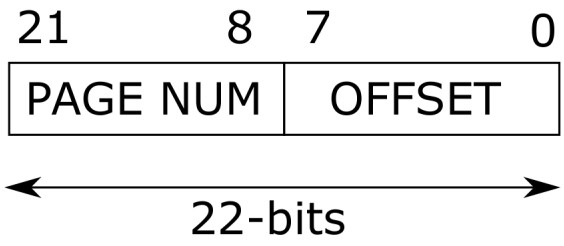
Các vùng nhớ này thực chất hoạt động giống như các biến trong mã nguồn chương trình có thể đọc được của con người. Tạm thời, chúng ta có thể hình dung các vùng này như một tập hợp các vùng có số lượng giới hạn. Chúng ta quản lý chúng bằng cách sử dụng một mảng `symrgtbl[PAGING_MAX_SYMTBL_SZ]`. Kích thước của mảng được cố định bởi hằng số `[PAGING_MAX_SYMTBL_SZ]` đại diện cho số lượng biến tối đa được phép trong mỗi chương trình. Cuối cùng, chúng ta sử dụng cấu trúc `struct vm_rg_struct symrgtbl` để lưu trữ điểm bắt đầu, điểm kết thúc của vùng và con trỏ `rg_next` được dành riêng để theo dõi tập hợp trong tương lai.

Ánh xạ bộ nhớ được biểu diễn bởi cấu trúc `mm_struct`, theo dõi tất cả các vùng nhớ đã đề cập trong một vùng nhớ liên tiếp riêng biệt. Trong mỗi cấu trúc ánh xạ bộ nhớ, nhiều vùng nhớ được trỏ đến bởi danh sách được liên kết `struct vm_area_struct *mmap_list`. Một trường quan trọng tiếp theo là pgd chứa tất cả các mục nhập bảng trang. Mỗi mục nhập là một ánh xạ giữa số trang và số khung trang trong hệ thống quản lý bộ nhớ phân trang. `symrgtbl` là một biểu hiện đơn giản của bảng ký hiệu. Các trường khác chủ yếu được sử dụng để theo dõi một hoạt động cụ thể của

người dùng, ví dụ như caller và `fifo_page` . Chúng tôi tạm thời bỏ qua các trường này vì chúng có thể được sử dụng tùy theo nhu cầu của bạn hoặc loại bỏ hoàn toàn.

Địa chỉ CPU là địa chỉ được CPU tạo ra để truy cập một vị trí bộ nhớ cụ thể. Trong hệ thống quản lý bộ nhớ phân trang, địa chỉ CPU được chia thành hai phần:

- Số trang (Page Number): Xác định vị trí của trang trong không gian địa chỉ logic.
- Độ lệch trang (Page Offset): Xác định vị trí cụ thể của byte hoặc word trong trang.




Hình 2: Địa chỉ CPU

Không gian địa chỉ vật lý của một tiến trình có thể không liên tục. Chúng ta chia bộ nhớ vật lý thành các khối kích thước cố định (gọi là khung trang) với hai kích thước 256B hoặc 512B. Hình 6 trình bày các thiết lập kết hợp khác nhau và cấu hình được đánh dấu là lựa chọn cuối cùng. Đây là thiết lập tham chiếu và có thể được sửa đổi hoặc chọn lại trong các mô phỏng khác. Dựa trên cấu hình CPU 22 bit và kích thước trang 256B, địa chỉ CPU được tổ chức như trong hình 5.

| CPU bus | PAGE size | PAGE bit | No pg entry | PAGE Entry sz | PAGE TBL | OFFSET bit | PGT mem | MEMPHY | fram bit |
|---|---|---|---|---|---|---|---|---|---|
| 20 | 256B | 12 | ~4000 | 4byte | 16KB | 8 | 2MB | 1MB | 12 |
| 22 | 256B | 14 | ~16000 | 4byte | 64KB | 8 | 8MB | 1MB | 12 |
| 22 | 512B | 13 | ~8000 | 4byte | 32KB | 9 | 4MB | 1MB | 11 |
| 22 | 512B | 13 | ~8000 | 4byte | 32KB | 9 | 4MB | 128kB | 8 |
| 16 | 512B | 8 | 256 | 4byte | 1kB | 9 | 128K | 128kB | 4 |

Hình 3: Giá trị cấu hình bus địa chỉ CPU khác nhau

Trong tóm tắt về Mô-đun ảo (VM), tất cả các cấu trúc hỗ trợ VM được đặt trong mô-đun mm-vm.c.

## Câu hỏi: Trong hệ điều hành đơn giản này, chúng ta triển khai thiết kế sử dụng nhiều vùng nhớ (segment) hoặc vùng nhớ được khai báo trong mã nguồn. Vậy lợi ích của thiết kế theo vùng nhớ này là gì?

Thiết kế sử dụng nhiều vùng nhớ (segment) hoặc vùng nhớ được khai báo trong mã nguồn trong một hệ điều hành đơn giản mang lại nhiều lợi ích đáng kể, bao gồm:

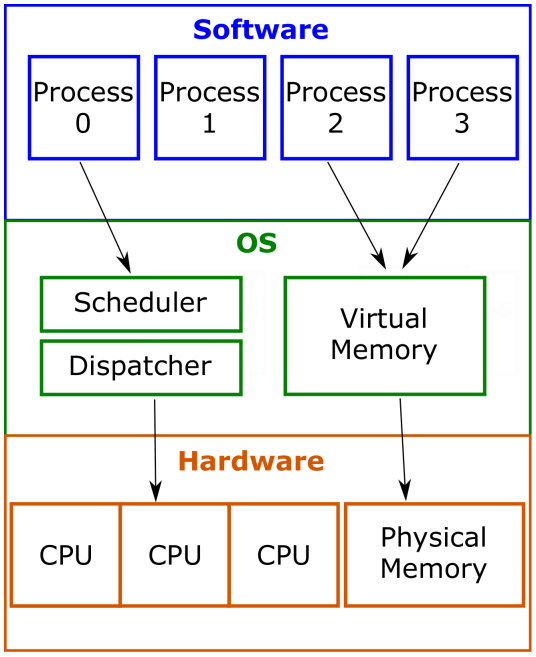
1. Quản lý và bảo Vệ bộ nhớ: Vùng nhớ giúp phân chia bộ nhớ thành các đơn vị logic(mã, dữ liệu, ngăn xếp), giúp dễ quản lý và bảo vệ dữ liệu tránh truy cập trái phép.

2. Chia sẻ bộ nhớ: Các vùng nhớ có thể được chia sẻ giữa các tiến trình, tiết kiệm bộ nhớ vật lý và tăng hiệu quả sử dụng tài nguyên. Điều này khiến cho việc triển khai bộ nhớ ảo dễ dàng hơn, cho phép không gian địa chỉ ảo lớn hơn bộ nhớ vật lý thực tế, hỗ trợ ứng dụng yêu cầu nhiều bộ nhớ.

3. Tối ưu hiệu suất, hỗ trợ phát triển và bảo trì phần mềm: Quản lý vùng nhớ giúp giảm phân mảnh và tối ưu hóa sử dụng bộ nhớ, tăng hiệu suất hệ thống. Việc này giúp cho quản lý các phần của chương trình một cách tách biệt và có tổ chức, cải thiện độ tin cậy và khả năng bảo trì phần mềm.

## 4.2 The system physical memory

Hình 10 cho thấy phần cứng bộ nhớ được cài đặt trên toàn hệ thống. Tất cả các tiến trình đều sở hữu ánh xạ bộ nhớ riêng biệt nhưng tất cả các ánh xạ này đều trỏ đến một thiết bị vật lý duy nhất. Có hai loại thiết bị là RAM và SWAP. Cả hai loại thiết bị này đều có thể được triển khai bởi cùng một thiết bị vật lý như trong file mm-memphy.c với các thiết lập khác nhau. Các thiết lập được hỗ trợ bao gồm: truy cập bộ nhớ ngẫu nhiên, truy cập bộ nhớ tuần tự/liên tiếp (sequential/serial), và dung lượng lưu trữ.




Hình 4: Địa chỉ CPU

Mặc dù có nhiều cấu hình có thể, việc sử dụng logic của các thiết bị này vẫn có thể phân biệt được.Thiết bị RAM thuộc hệ thống bộ nhớ chính, có thể truy cập trực tiếp từ bus địa chỉ CPU, nghĩa là có thể đọc/ghi bằng các lệnh CPU. Thiết bị SWAP chỉ là một thiết bị bộ nhớ thứ cấp, và tất cả thao tác dữ liệu được lưu trữ trên SWAP phải được thực hiện bằng cách di chuyển chúng vào bộ nhớ chính. Do không thể truy cập trực tiếp từ CPU, hệ thống thường trang bị một thiết bị SWAP dung lượng lớn với chi phí thấp và thậm chí có thể có nhiều hơn một thiết bị SWAP. Trong thiết lập của chúng tôi, hệ thống hỗ trợ phần cứng được cài đặt với một thiết bị RAM và tối đa 4 thiết bị SWAP.

Cấu trúc `framephy_ struct` chủ yếu được sử dụng để lưu trữ số khung của bộ nhớ vật lý.

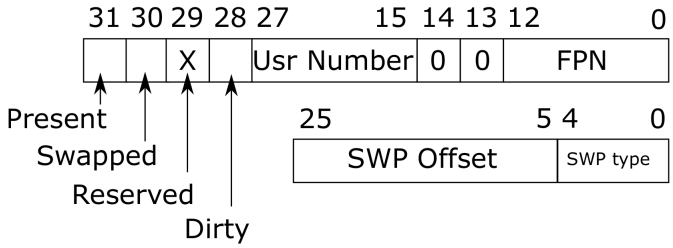
Cấu trúc `memphy_struct` có các trường cơ bản là storage và size. Các trường `free_fp_list` và `used _fp _list` được dành riêng để giữ lại các khung bộ nhớ chưa sử dụng và đã sử dụng tương ứng.

## Câu hỏi: Điều gì sẽ xảy ra nếu chúng ta chia địa chỉ thành nhiều cấp hơn 2 trong hệ thống quản lý bộ nhớ phân trang?

1. Lợi ích
    - Giảm kích thước bảng trang: Mỗi cấp chỉ xử lý một phần nhỏ không gian địa chỉ, dẫn đến tổng kích thước bảng trang nhỏ hơn.
    - Tăng khả năng chia sẻ trang: Các trang có thể dễ dàng được chia sẻ giữa các tiến trình khác nhau, nâng cao hiệu quả sử dụng bộ nhớ.
    - Hỗ trợ không gian địa chỉ lớn hơn: Hệ thống có thể truy cập vào không gian địa chỉ lớn hơn nhiều so với hệ thống chỉ sử dụng phân chia hai cấp.
2. Hạn chế
    - Độ phức tạp tăng lên: Thuật toán dịch địa chỉ trở nên phức tạp hơn, tiềm ẩn ảnh hưởng đến hiệu suất.
    - Tốc độ truy cập bộ nhớ giảm: Việc dịch địa chỉ đa cấp có thể làm tăng thêm gánh nặng cho quá trình truy cập bộ nhớ, dẫn đến giảm tốc độ hoạt động.
    - Chi phí phần cứng cao hơn: Việc triển khai dịch địa chỉ đa cấp có thể yêu cầu hỗ trợ phần cứng bổ sung, làm tăng chi phí hệ thống.

## 4.3 Paging-based address translation scheme

Bản dịch hỗ trợ cả phân đoạn và phân đoạn bằng phân trang. Trong phiên bản này, chúng tôi phát triển một hệ thống phân trang một cấp tận dụng hầu hết một thiết bị RAM và một phần cứng phiên bản SWAP. Chúng tôi được chuẩn bị (mã hóa) với khả năng của nhiều phân đoạn bộ nhớ, nhưng chúng tôi vẫn chủ yếu tập trung vào phân đoạn đầu tiên và là đoạn duy nhất của vùng vm có (vmaid = 0). Các phiên bản tiếp theo sẽ tính đến sơ đồ phân trang đầy đủ của nhiều phân đoạn hoặc có thể chồng chéo/không chồng chéo giữa các phân đoạn




Hình 5: Định dạng bảng trang

Cấu trúc bảng trang này cho phép quá trình không gian người dùng tìm ra khung vật lý của mỗi trang ảo ánh xạ tới. Nó chứa một giá trị 32 bit cho mỗi trang ảo, chứa dữ liệu sau:

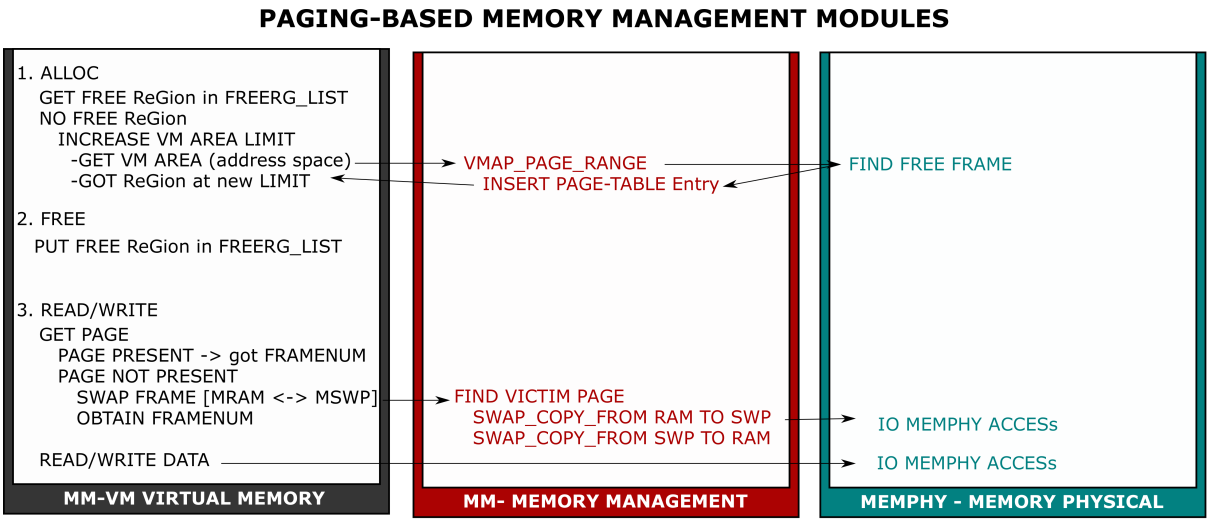
```
    * Bits 0-12  page frame number (FPN) if present
    * Bits 13-14 zero if present
    * Bits 15-27 user-defined numbering if present
    * Bits 0-4   swap type if swapped
    * Bits 5-25  swap offset if swapped
    * Bit  28    dirty
    * Bits 29    reserved
    * Bit  30    swapped
    * Bit  31    presented
```

Để làm việc dựa trên phân trang hệ thống bộ nhớ, chúng ta cần cập nhật cấu trúc này và phần sau sẽ thảo luận về việc sửa đổi cần thiết.Trong mọi trường hợp, mỗi quy trình có một không gian hoàn toàn biệt lập và duy nhất, N quy trình trong cài đặt của chúng tôi dẫn đến N bảng trang và lần lượt, mỗi trang phải có tất cả các mục cho toàn bộ không gian địa chỉ CPU. Đối với mỗi mục, số phân trang có thể có một frame liên quan trong MEMRAM hoặc MEMSWP hoặc là có thể có giá trị null, chức năng của từng bit dữ liệu của mục bảng trang được minh họa trong Hình 13.Trong phần nổi bật đã chọn của chúng tôi cài đặt trong Hình 7, chúng tôi có bảng 16.000 mục nhập, mỗi bảng có dung lượng lưu trữ 64 KB.

Trong phần 4.1 , quá trình có thể truy cập vào không gian bộ nhớ ảo theo cách liền kề của cấu trúc vùng vm. Công việc còn lại xử lý việc ánh xạ giữa trang và khung để cung cấp không gian bộ nhớ liền kề qua cơ chế lưu trữ khung rời rạc. Nó rơi vào hai cách tiếp cận chính là trao đổi bộ nhớ

và cơ bản các hoạt động bộ nhớ, tức là cấp phát/miễn phí/đọc/ghi, hầu hết giữ liên lạc với cấu trúc bảng trang pgd.

Chúng tôi đã được thông báo rằng một vùng bộ nhớ (phân đoạn) có thể không được sử dụng hết mức hạn chế không gian lưu trữ. Điều đó có nghĩa là có những không gian lưu trữ không được ánh xạ tới MEMRAM. Việc hoán đổi có thể giúp di chuyển nội dung của khung vật lý giữa MEMRAM và MEMSWAP. Các swapping là cơ chế thực hiện sao chép nội dung của frame từ bên ngoài vào ram bộ nhớ chính. Ngược lại, hoán đổi sẽ cố gắng di chuyển nội dung của khung trong MEMRAM sang MEMSWAP. Điển hình như trong bối cảnh này, việc hoán đổi giúp chúng ta có được khung RAM trống do kích thước của thiết bị SWAP thường đủ lớn.




Hình 6: Mô-đun hệ thống bộ nhớ

### Các thao tác cơ bản với bộ nhớ trong hệ thống phân trang

- Phân bổ trong hầu hết các trường hợp, nó phù hợp với khu vực có sẵn. Nếu không có không gian phù hợp như vậy, chúng ta cần nâng lên rào cản sbrk và vì nó chưa bao giờ được chạm vào nên nó có thể cần cung cấp một số frames vật lý và sau đó ánh xạ chúng bằng cách sử dụng Mục nhập bảng trang.
- Miễn phí không gian lưu trữ được liên kết với id khu vực. Vì chúng tôi không thể thu hồi lại vật chất đã lấy trong frame vật lý có thể gây ra lỗ hổng bộ nhớ, chúng tôi chỉ giữ không gian lưu trữ đã thu thập trong danh sách trống để sử dụng thêm yêu cầu phân bổ
- ĐỌC/ VIẾT yêu cầu trang được hiển thị trong bộ nhớ chính. Tài nguyên nhất bước tiêu tốn là hoán đổi trang. Nếu trang nằm trong thiết bị MEMSWAP, nó cần mang theo trang đó trở lại thiết bị MEMRAM (đổi vào) và nếu thiếu dung lượng, chúng tôi cần trả lại một số trang sang thiết bị MEMSWAP (đổi ra) để tạo thêm phòng.

Để thực hiện các thao tác này, nó cần có sự cộng tác giữa các mô-đun của mm như minh họa trong Hình 15.

## Câu hỏi:Ưu điểm và nhược điểm của việc phân đoạn bằng phân trang là gì??

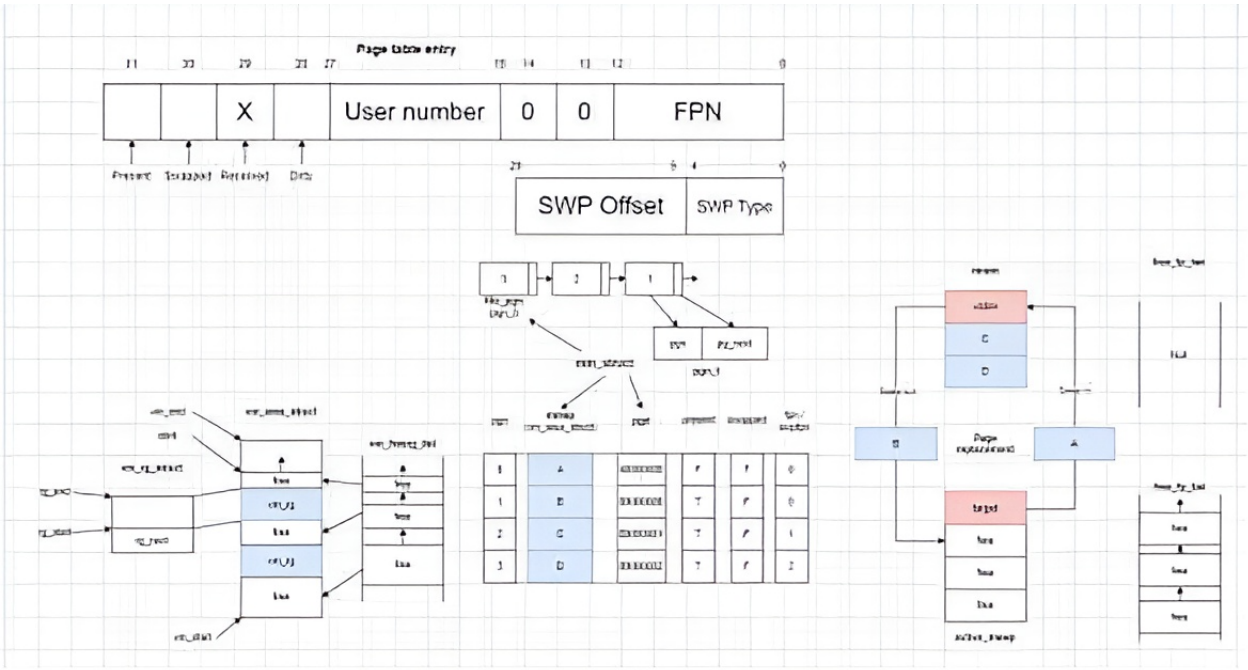
1. Ưu điểm

   - Phân trang trong các đoạn giảm cả phân mảnh bên trong và bên ngoài. Phân đoạn chia chương trình theo cách logic, trong khi phân trang cấp phát các khối bộ nhớ có kích thước cố định, sử dụng bộ nhớ hiệu quả.
   - Phân đoạn chia chương trình thành các đơn vị logic (ví dụ: mã, dữ liệu, ngăn xếp), cho phép kiểm soát truy cập chi tiết. Mỗi đoạn có thể có các quyền truy cập khác nhau (đọc, ghi, thực thi), tăng cường bảo mật.
   - Kết hợp phân đoạn và phân trang đơn giản hóa quá trình dịch địa chỉ. Bảng đoạn cung cấp địa chỉ cơ sở của bảng trang, và bảng trang ánh xạ các trang tới khung, giúp quá trình dịch địa chỉ trở nên mạch lạc hơn.
   - Hỗ trợ không gian địa chỉ lớn bằng cách chia chương trình thành các đoạn có thể phân trang riêng lẻ. Điều này giúp sử dụng bộ nhớ lớn một cách hiệu quả, hỗ trợ nhu cầu của các ứng dụng hiện đại.
   - Phân đoạn phù hợp với cấu trúc chương trình, cho phép tổ chức module. Điều này hỗ trợ trong việc phát triển, gỡ lỗi và bảo trì, vì dữ liệu và lệnh liên quan được nhóm lại một cách logic.

2. Nhược điểm

   - Quản lý cả đoạn và trang tạo ra sự phức tạp. Hệ điều hành phải duy trì và xử lý nhiều bảng (bảng đoạn và bảng trang), tăng chi phí quản lý.
   - Trong khi phân trang giảm phân mảnh trong đoạn, phân mảnh ở cấp độ đoạn vẫn có thể xảy ra. Quản lý các đoạn có kích thước khác nhau có thể dẫn đến sử dụng bộ nhớ không hiệu quả.
   - Phân đoạn kết hợp phân trang yêu cầu thêm bộ nhớ để lưu trữ bảng đoạn và bảng trang. Chi phí này có thể đáng kể, đặc biệt với số lượng đoạn và trang lớn.
   - Quá trình dịch địa chỉ hai cấp (đoạn đến trang đến khung) có thể gây ra độ trễ. Mỗi lần truy cập bộ nhớ có thể yêu cầu nhiều lần tra cứu, ảnh hưởng đến hiệu suất.
   - Việc tải và hoán đổi các đoạn và trang vào và ra khỏi bộ nhớ có thể phức tạp. Hệ điều hành phải xử lý các hoạt động này một cách hiệu quả để tránh suy giảm hiệu suất.

### Mô hình hoạt động của hệ thống



## 4.4 Translation Lookaside Buffer(TLB)

Trong phần trước, hệ điều hành và hệ thống quản lý bộ nhớ đã triển khai rằng mỗi tiến trình có bảng trang của riêng mình. Bảng này chứa các mục nhập bảng trang cung cấp số khung trang. Thách thức nằm ở việc tối ưu hóa thời gian truy cập cho các mục nhập này, với thời gian truy cập gấp đôi của việc đọc bảng trang (thực chất được đặt trong bộ nhớ chính hoặc MEMPHY) và truy cập dữ liệu bộ nhớ trong MEMPHY. Với kích thước tiến trình lớn dẫn đến chi phí cao, TLB được đề xuất để tận dụng khả năng của bộ nhớ cache nhờ vào tính chất tốc độ cao của nó. TLB thường là MEMPHY nhưng hoạt động như một bộ nhớ cache tốc độ cao cho các mục nhập bảng trang. Tuy nhiên, bộ nhớ cache là thành phần có chi phí cao; do đó, nó có dung lượng hạn chế. Đây là các công việc cơ bản của TLB:

- Phương pháp truy cập TLB: TLB là MEMPHY với cơ chế ánh xạ để xác định cách nội dung được liên kết với thông tin định danh. Trong công việc này, chúng tôi tận dụng kiến thức đã học từ các khóa học trước về phần cứng máy tính, nơi áp dụng các kỹ thuật ánh xạ bộ nhớ cache: ánh xạ trực tiếp / ánh xạ tập hợp / ánh xạ hoàn toàn.
- Cấu hình TLB: TLB chứa các mục nhập bảng trang được sử dụng gần đây. Khi CPU tạo ra một địa chỉ ảo, nó kiểm tra TLB:
  - Nếu một mục nhập bảng trang có sẵn (một TLB hit), số khung trang tương ứng sẽ được lấy ra.
  - Nếu một mục nhập bảng trang không được tìm thấy (một TLB miss), số trang sẽ được sử dụng như một chỉ mục để truy cập bảng trang trong bộ nhớ chính. Nếu trang không có trong bộ nhớ chính, một lỗi trang xảy ra và TLB được cập nhật với mục nhập trang mới.
- TLB Hit:
  - CPU tạo ra một địa chỉ ảo.
  - TLB được kiểm tra (mục nhập hiện có).
  - Số khung tương ứng được truy xuất.
- TLB Miss:
  - CPU tạo ra một địa chỉ ảo.
  - TLB được kiểm tra (mục nhập không tồn tại).
  - Số trang được so khớp với bảng trang trong bộ nhớ chính.
  - Số khung tương ứng được truy xuất.

Vì TLB hoàn toàn là một thiết bị lưu trữ bộ nhớ, bạn có thể thiết kế một phương pháp hiệu quả để xác định ánh xạ cache. Chúng tôi không cố định một thiết kế, chúng tôi chỉ đưa ra một đề xuất về phương án ánh xạ dựa trên pid và số trang, lưu ý rằng vì bộ nhớ cache TLB được chia sẻ bởi tất cả các tiến trình hệ thống và được dự định sử dụng tại cấp CPU; do đó, pid là cần thiết.

Các hoạt động của TLB (tlballoc/tlbfree/tlbread/tlbwrite) xảy ra trước khi hoạt động phân trang bộ nhớ diễn ra (alloc/free/read/write). Cơ chế dịch địa chỉ chuyển qua các tầng này theo một thứ tự cụ thể khi người dùng gửi các yêu cầu bộ nhớ và sau đó lại qua lại theo thứ tự ngược lại khi

địa chỉ được nhận. Do mô hình lập trình phân cấp tự nhiên của nó, nếu bạn nghĩ rằng điều này sẽ làm mọi thứ dễ dàng hơn, bạn có thể tự do triển khai cập nhật TLB CACHE trong bất kỳ mô-đun nào (tlb hoặc mm-vm) thông qua việc bỏ qua giữa các mô-đun. Chúng tôi đã dành một bit được thiết kế để thuận tiện cho mục đích điều khiển trang, chúng tôi hy vọng phần dành riêng này sẽ hữu ích. Đối với một hoạt động chung (thay thế xxx bằng alloc/free/read/write)

Đối với việc phát triển tiên tiến, trong một cộng đồng phát triển thực sự, nó không liên quan đến TLB mà là một phương pháp bộ nhớ cache cơ bản, nơi các nhà sản xuất cạnh tranh về hiệu suất và tỷ lệ trúng. Để cung cấp hiệu suất cache trúng/miss tốt hơn, họ nghiên cứu kỹ lưỡng mỗi nội dung được bảo tồn trong cache.

- Tạo cache: trong một hệ thống đáng tin cậy, trước khi xóa hoặc cập nhật một cách nhẹ nhàng nội dung của cache, họ bảo tồn một phiên bản hoặc thế hệ cache được đặt tên (tlb gen nếu bạn cần một tham chiếu) và xác minh rằng thế hệ đã trải qua sự sửa đổi bộ nhớ đáng kể. Điều này là một ưu điểm của bố cục tuyệt vời của bạn, nhưng chúng ta có thể loại bỏ nó cho yêu cầu tối thiểu cần thiết.
- Locality threshold: để duy trì các mục nhập TLB đã được phân bổ phù hợp với địa phương hiện tại, chúng tôi thêm một mục nhập sau khi một truy xuất bộ nhớ đạt đến một ngưỡng nhất định.
- Bảng trang đa cấp: bảng trang đa cấp có thể hiệu quả hơn khi không gian địa chỉ ảo nhỏ hoặc trung bình. TLB nên được sử dụng một cách hiệu quả hơn với bảng trang đa cấp bằng cách giảm số lượng mục nhập cần tìm kiếm. Tuy nhiên, việc triển khai bảng trang đa cấp lại khó hơn.

## Câu hỏi: Điều gì sẽ xảy ra nếu hệ thống đa lõi có mỗi lõi CPU có thể chạy trong một ngữ cảnh khác nhau, và mỗi lõi có MMU riêng và một phần của lõi (TLB)? Trong CPU hiện đại, TLB 2 cấp hiện đang phổ biến, tác động của cấu hình phần cứng bộ nhớ mới này đối với các phương pháp dịch của chúng ta là gì?

Nếu mỗi lõi CPU trong hệ thống đa lõi có thể chạy trong ngữ cảnh khác nhau và có MMU riêng cũng như một phần của lõi (TLB), điều quan trọng là mỗi lõi sẽ duy trì bản sao riêng của bảng trang và TLB của mình. Điều này có nghĩa là mỗi lõi có thể có một bản sao của TLB của riêng mình, làm cho nó có thể duy trì các bản ghi trang riêng biệt và cập nhật chúng mà không cần liên lạc với các lõi khác.

Trong các CPU hiện đại, TLB 2 cấp trở nên phổ biến. Với cấu hình phần cứng bộ nhớ mới này, có thể có tác động đến các phương pháp dịch của chúng ta. Ví dụ, việc sử dụng nhiều bộ TLB riêng biệt trên mỗi lõi CPU có thể đưa đến việc cần thiết phải cập nhật cơ chế dịch bảng trang để quản lý TLB đa cấp này hiệu quả hơn. Điều này có thể đưa đến việc phải xem xét lại chiến lược ánh xạ bộ nhớ và cách quản lý TLB để tối ưu hóa hiệu suất và sử dụng bộ nhớ.

## Câu hỏi: Điều gì sẽ xảy ra nếu đồng bộ hóa không được xử lý trong hệ điều hành đơn giản của bạn? Mô tả vấn đề của hệ điều hành đơn giản của bạn bằng ví dụ nếu có. Lưu ý: Bạn cần chạy hai phiên bản của hệ điều hành đơn giản của mình: chương trình có/ không có đồng bộ hóa, sau đó quan sát hiệu suất của chúng dựa trên kết quả thử nghiệm và giải thích sự khác biệt của chúng.

Nếu đồng bộ hóa không được xử lý trong một hệ điều hành đơn giản, điều này có thể dẫn đến các vấn đề khác nhau, bao gồm các tình huống cạnh tranh (race conditions), không nhất quán dữ liệu và tình trạng bế tắc (deadlock).

Hãy xem xét một ví dụ đơn giản để minh họa các vấn đề này. Giả sử hệ điều hành đơn giản của chúng ta cho phép nhiều tiến trình truy cập và sửa đổi một biến chia sẻ cùng một lúc mà không có bất kỳ cơ chế đồng bộ hóa nào:

Quá trình A và Quá trình B là hai quá trình đồng thời tăng và giảm một biến chia sẻ count. Cả hai quá trình bắt đầu với count được khởi tạo là 0.

Quá trình A thực thi mã sau: count — count + 1

Quá trình B thực thi mã sau: count = count - 1

Trong trường hợp không có đồng bộ hóa, chuỗi sự kiện sau có thể xảy ra:

- Quá trình A đọc giá trị hiện tại của count (0) vào một thanh ghi.
- Quá trình B đọc giá trị hiện tại của count (0) vào một thanh ghi.
- Quá trình A tăng giá trị thanh ghi cục bộ của nó lên 1 (0 + 1 = 1).
- Quá trình B giảm giá trị thanh ghi cục bộ của nó xuống 1 (0 - 1 = -1).
- Quá trình A ghi giá trị đã cập nhật của nó (1) trở lại vào count.
- Quá trình B ghi giá trị đã cập nhật của nó (-1) trở lại vào count.

Trong kịch bản này, cả hai quá trình đã thực thi các thao tác của chúng đồng thời mà không có bất kỳ đồng bộ hóa nào. Kết quả là giá trị cuối cùng của count không chính xác (-1 thay vì 0). Vấn đề này được gọi là tình trạng cạnh tranh (race condition), nơi mà kết quả của chương trình phụ thuộc vào thời gian và sự xen kẽ của các thao tác đồng thời.

Ngoài ra, nếu không có cơ chế đồng bộ hóa thích hợp, có nguy cơ xảy ra tình trạng bế tắc (deadlock). Bế tắc có thể xảy ra khi hai hoặc nhiều quá trình chờ đợi vô thời hạn để nhau giải phóng tài nguyên. Nếu có các tài nguyên chia sẻ mà các quá trình cần truy cập độc quyền, mà không có đồng bộ hóa thích hợp, có thể xảy ra tình huống mà nhiều quá trình đang chờ một tài nguyên sẽ không bao giờ được giải phóng.

# 5 Implementation

## 5.1 Priority Queue

**Hàm enqueue và dequeue**

- **enqueue**: Đẩy 1 process
- **dequeue**: Lấy ra 1 process

```
12  void enqueue(struct queue_t * q, struct pcb_t * proc) {
13          if (q == NULL || proc == NULL) {
14                  return;
15          }
16
17          if (q->size == 0) {
18                  q->proc[0] = proc;
19          } else {
20                  q->proc[q->size] = proc;
21          }
22
23          q->size++;
24  }
25
26  struct pcb_t * dequeue(struct queue_t * q) {
27          if (q == NULL || q->size == 0) {
28                  return NULL;
29          }
30
31          struct pcb_t* proc = q->proc[0];
32
33          for(int i = 0; i < q->size - 1; i++) {
34                  q->proc[i] = q->proc[i+1];
35          }
36
37          q->size--;
38
39          return proc;
40  }
```

## 5.2 Scheduler

**get_mlq_proc:** Lấy 1 process từ PRIORITY[ready_queue]

```c
// Get a process from the MLQ
struct pcb_t * get_mlq_proc(void) {
    struct pcb_t * proc = NULL;
    pthread_mutex_lock(&queue_lock);
    label:
    if(flag){
        for(; MarkedPrior < MAX_PRIO; MarkedPrior++){
            if(mlq_ready_queue[MarkedPrior].size != 0 && prioSlot[MarkedPrior] < MAX_PRIO - MarkedPrior){
                proc = dequeue(&mlq_ready_queue[MarkedPrior]);
                prioSlot[MarkedPrior]++;
                flag = 0;
                count = 0;
                pthread_mutex_unlock(&queue_lock);
                return proc;
            }
            count++;
        }
        for(int j = 0; j < MAX_PRIO; j++){
            prioSlot[j] = 0;
        }
        MarkedPrior = 0;
        flag = 1;
        if(count == MAX_PRIO){
            pthread_mutex_unlock(&queue_lock);
            return proc;
        }else{
            count = 0;
        }
        goto label;
    }else{
        if(prioSlot[MarkedPrior] < MAX_PRIO - MarkedPrior && mlq_ready_queue[MarkedPrior].size != 0 ){
            proc = dequeue(&mlq_ready_queue[MarkedPrior]);
            prioSlot[MarkedPrior]++;
            pthread_mutex_unlock(&queue_lock);
            return proc;
        }else{
            flag = 1;
            goto label;
        }
    }
    pthread_mutex_unlock(&queue_lock);
    return proc;
}
```

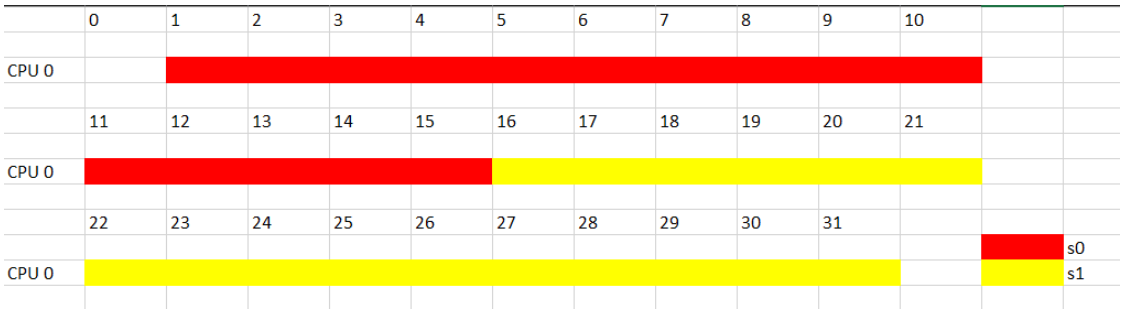
**Input:** /input/sched_0
**Output:** /output/sched_0:

```
Time slot   0
ld_routine
        Loaded a process at input/proc/s0, PID: 1 PRIO: 4

Time slot   1
        CPU 0: Dispatched process  1
        Loaded a process at input/proc/s0, PID: 2 PRIO: 0
```

```
Time slot   2

Time slot   3
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1

Time slot   4

Time slot   5
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1

Time slot   6

Time slot   7
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1

Time slot   8

Time slot   9
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1

Time slot  10

Time slot  11
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1

Time slot  12

Time slot  13
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1

Time slot  14

Time slot  15
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1

Time slot  16
        CPU 0: Processed  1 has finished
        CPU 0: Dispatched process  2
```

```
Time slot  17

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

Time slot  19

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

Time slot  21

Time slot  22
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2

Time slot  23

Time slot  24
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2

Time slot  25

Time slot  26
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2

Time slot  27

Time slot  28
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2

Time slot  29

Time slot  30
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2

Time slot  31
        CPU 0: Processed  2 has finished
        CPU 0 stopped
```

**Gantt Chart của sched_0:**




Hình 7: 1 CPU thực thi 2 process

## 5.3 Memory Management

**Hiển thị trạng thái của RAM** Dưới đây là kết quả của testcase quá trình logging sau mỗi lần cấp phát, giải phóng, đọc ghi lệnh trong chương trình

**Input:** /input/os_1_singleCPU_mlq_paging
**Output:** /output/os_1_singleCPU_mlq_paging:

```
Time slot   0
ld_routine
        Loaded a process at input/proc/s4, PID: 1 PRIO: 4

Time slot   1

Time slot   2
        CPU 0: Dispatched process  1
        Loaded a process at input/proc/s3, PID: 2 PRIO: 3

Time slot   3
        Loaded a process at input/proc/m1s, PID: 3 PRIO: 2
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1

Time slot   4

Time slot   5
        Loaded a process at input/proc/s2, PID: 4 PRIO: 3
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1

Time slot   6

Time slot   7
        Loaded a process at input/proc/m0s, PID: 5 PRIO: 3

Time slot   8
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
        Loaded a process at input/proc/p1s, PID: 6 PRIO: 2
        CPU 0: Processed  1 has finished

Time slot   9
        CPU 0: Dispatched process  3

--------------------------------------
Process 3 trong tballoc, Kích thước: 300 , chỉ số vùng: 0
Địa chỉ bắt đầu và kết thúc của vùng nhớ: 0 300
```

```
Start: 0, End: 512, sbrk: 512 của vùng nhớ ảo hiện tại của tiến trình này

----------------------------------------
Địa chỉ: 0
print_pgtbl: 0 - 512
00000000: 80000000
00000004: 80000001

Time slot  10

----------------------------------------
Process 3 trong tballoc, Kích thước: 100 , chỉ số vùng: 1
Địa chỉ bắt đầu và kết thúc của vùng nhớ: 301 401
Start: 0, End: 512, sbrk: 512 của vùng nhớ ảo hiện tại của tiến trình này

----------------------------------------
Địa chỉ: 301
print_pgtbl: 0 - 512
00000000: 80000000
00000004: 80000001
        Loaded a process at input/proc/s0, PID: 7 PRIO: 1

Time slot  11
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  6

Time slot  12

Time slot  13
        CPU 0: Put process  6 to run queue
        CPU 0: Dispatched process  3

----------------------------------------
Process 3 trong hàm tlbfree với chỉ số vùng nhớ: 0
Địa chỉ bắt đầu: 0 và địa chỉ kết thúc: 300 của vùng nhớ được giải phóng
Start: 0, End: 512, sbrk: 512 của vùng nhớ hiện tại của tiến trình này

----------------------------------------
print_pgtbl: 0 - 512
00000000: 80000000
00000004: 80000001

Time slot  14

----------------------------------------
Process 3 trong tballoc, Kích thước: 100 , chỉ số vùng: 2
```

```
Địa chỉ bắt đầu và kết thúc của vùng nhớ: 0 100
Start: 0, End: 512, sbrk: 512 của vùng nhớ ảo hiện tại của tiến trình này

----------------------------------------
Địa chỉ: 0
print_pgtbl: 0 - 512
00000000: 80000000
00000004: 80000001

Time slot  15
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  6

Time slot  16
        Loaded a process at input/proc/s1, PID: 8 PRIO: 0

Time slot  17
        CPU 0: Put process  6 to run queue
        CPU 0: Dispatched process  3

----------------------------------------
Process 3 trong hàm tlbfree với chỉ số vùng nhớ: 2
Địa chỉ bắt đầu: 0 và địa chỉ kết thúc: 100 của vùng nhớ được giải phóng
Start: 0, End: 512, sbrk: 512 của vùng nhớ hiện tại của tiến trình này

----------------------------------------
print_pgtbl: 0 - 512
00000000: 80000000
00000004: 80000001

Time slot  18

----------------------------------------
Process 3 trong hàm tlbfree với chỉ số vùng nhớ: 1
Địa chỉ bắt đầu: 301 và địa chỉ kết thúc: 401 của vùng nhớ được giải phóng
Start: 0, End: 512, sbrk: 512 của vùng nhớ hiện tại của tiến trình này

----------------------------------------
print_pgtbl: 0 - 512
00000000: 80000000
00000004: 80000001

Time slot  19
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  6
```

```
Time slot  20

Time slot  21
        CPU 0: Put process  6 to run queue
        CPU 0: Dispatched process  3

---------------------------------------
Process 3 trong hàm tlbfree với chỉ số vùng nhớ: 0
Địa chỉ bắt đầu: 100 và địa chỉ kết thúc: 300 của vùng nhớ được giải phóng
Start: 0, End: 512, sbrk: 512 của vùng nhớ hiện tại của tiến trình này

---------------------------------------
print_pgtbl: 0 - 512
00000000: 80000000
00000004: 80000001

Time slot  22

---------------------------------------
Process 3 trong hàm tlbfree với chỉ số vùng nhớ: 0
Địa chỉ bắt đầu: 100 và địa chỉ kết thúc: 300 của vùng nhớ được giải phóng
Start: 0, End: 512, sbrk: 512 của vùng nhớ hiện tại của tiến trình này

---------------------------------------
print_pgtbl: 0 - 512
00000000: 80000000
00000004: 80000001

Time slot  23
        CPU 0: Processed  3 has finished
        CPU 0: Dispatched process  6

Time slot  24

Time slot  25
        CPU 0: Put process  6 to run queue
        CPU 0: Dispatched process  6

Time slot  26

Time slot  27
        CPU 0: Processed  6 has finished
        CPU 0: Dispatched process  2

Time slot  28
```

```
Time slot  29
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  4

Time slot  30

Time slot  31
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  5

---------------------------------------
Process 5 trong tballoc, Kích thước: 300 , chỉ số vùng: 0
Địa chỉ bắt đầu và kết thúc của vùng nhớ: 0 300
Start: 0, End: 512, sbrk: 512 của vùng nhớ ảo hiện tại của tiến trình này

---------------------------------------
Địa chỉ: 0
print_pgtbl: 0 - 512
00000000: 80000002
00000004: 80000003

Time slot  32

---------------------------------------
Process 5 trong tballoc, Kích thước: 100 , chỉ số vùng: 1
Địa chỉ bắt đầu và kết thúc của vùng nhớ: 301 401
Start: 0, End: 512, sbrk: 512 của vùng nhớ ảo hiện tại của tiến trình này

---------------------------------------
Địa chỉ: 301
print_pgtbl: 0 - 512
00000000: 80000002
00000004: 80000003

Time slot  33
        CPU 0: Put process  5 to run queue
        CPU 0: Dispatched process  2

Time slot  34

Time slot  35
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  4

Time slot  36
```

```
Time slot  37
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  5

---------------------------------------
Process 5 trong hàm tlbfree với chỉ số vùng nhớ: 0
Địa chỉ bắt đầu: 0 và địa chỉ kết thúc: 300 của vùng nhớ được giải phóng
Start: 0, End: 512, sbrk: 512 của vùng nhớ hiện tại của tiến trình này

---------------------------------------
print_pgtbl: 0 - 512
00000000: 80000002
00000004: 80000003

Time slot  38

---------------------------------------
Process 5 trong tballoc, Kích thước: 100 , chỉ số vùng: 2
Địa chỉ bắt đầu và kết thúc của vùng nhớ: 0 100
Start: 0, End: 512, sbrk: 512 của vùng nhớ ảo hiện tại của tiến trình này

---------------------------------------
Địa chỉ: 0
print_pgtbl: 0 - 512
00000000: 80000002
00000004: 80000003

Time slot  39
        CPU 0: Put process  5 to run queue
        CPU 0: Dispatched process  2

Time slot  40

Time slot  41
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  4

Time slot  42

Time slot  43
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  5
Process 5 đang thực hiện ghi vào TLB
fpn_retrieved từ TLB: 3
TLB hit at write region = 1 offset = 20 value = 102
print_pgtbl: 0 - 512
```

```
00000000: 80000002
00000004: 80000003
Register id: 1
Inside pg_setval, fpn: 3, phyaddr: 833
The value of RAM at address 833 is: 102

Time slot  44
Process 5 đang thực hiện ghi vào TLB
fpn_retrieved từ TLB: 0
TLB hit at write region = 2 offset = 1000 value = 1
print_pgtbl: 0 - 512
00000000: 80000002
00000004: 80000003
Register id: 2
Inside pg_setval, fpn: 0, phyaddr: 232
The value of RAM at address 232 is: 1

Time slot  45
        CPU 0: Put process  5 to run queue
        CPU 0: Dispatched process  2

Time slot  46

Time slot  47
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  4

Time slot  48

Time slot  49
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  5
Process 5 đang thực hiện ghi vào TLB
fpn_retrieved từ TLB: 2
TLB hit at write region = 0 offset = 0 value = 0
print_pgtbl: 0 - 512
00000000: c0000000
00000004: 80000003
Register id: 0
Inside pg_setval, fpn: 0, phyaddr: 100
The value of RAM at address 100 is: 0

Time slot  50
        CPU 0: Processed  5 has finished
        CPU 0: Dispatched process  2
```

```
Time slot  51

Time slot  52
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  4

Time slot  53

Time slot  54
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  2

Time slot  55
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  4

Time slot  56

Time slot  57
        CPU 0: Processed  4 has finished
        CPU 0: Dispatched process  8

---------------------------------------
Process 8 trong tballoc, Kích thước: 300 , chỉ số vùng: 0
Địa chỉ bắt đầu và kết thúc của vùng nhớ: 0 300
Start: 0, End: 512, sbrk: 512 của vùng nhớ ảo hiện tại của tiến trình này

---------------------------------------
Địa chỉ: 0
print_pgtbl: 0 - 512
00000000: 80000004
00000004: 80000005

Time slot  58

Time slot  59
        CPU 0: Put process  8 to run queue
        CPU 0: Dispatched process  8

Time slot  60

Time slot  61
        CPU 0: Put process  8 to run queue
        CPU 0: Dispatched process  8

Time slot  62
```

```
Time slot  63
        CPU 0: Put process  8 to run queue
        CPU 0: Dispatched process  8

Time slot  64
        CPU 0: Processed  8 has finished
        CPU 0: Dispatched process  7

Time slot  65

Time slot  66
        CPU 0: Put process  7 to run queue
        CPU 0: Dispatched process  7

Time slot  67

Time slot  68
        CPU 0: Put process  7 to run queue
        CPU 0: Dispatched process  7

Time slot  69

Time slot  70
        CPU 0: Put process  7 to run queue
        CPU 0: Dispatched process  7

Time slot  71

Time slot  72
        CPU 0: Put process  7 to run queue
        CPU 0: Dispatched process  7

Time slot  73

Time slot  74
        CPU 0: Put process  7 to run queue
        CPU 0: Dispatched process  7

Time slot  75

Time slot  76
        CPU 0: Put process  7 to run queue
        CPU 0: Dispatched process  7

Time slot  77
```

```
Time slot  78
        CPU 0: Put process  7 to run queue
        CPU 0: Dispatched process  7

Time slot  79
        CPU 0: Processed  7 has finished
        CPU 0 stopped
```