Trường Đại học Bách khoa TP.HCM
Khoa Khoa học Máy tính & Kỹ thuật

Khóa học: Hệ điều hành
Bài tập - Hệ điều hành đơn giản

Ngày 1 tháng 4 năm 2024

Mục tiêu: Mục tiêu của bài tập này là mô phỏng các thành phần chính trong một hệ điều hành đơn giản, ví dụ như trình lập lịch, đồng bộ hóa, các hoạt động liên quan đến bộ nhớ vật lý và bộ nhớ ảo.

Nội dung: Học viên sẽ được thực hành chi tiết với 3 module chính: lập lịch, đồng bộ hóa, cơ chế phân bổ bộ nhớ từ bộ nhớ ảo sang bộ nhớ vật lý.

- người lập lịch
- đồng bộ hóa
- các hoạt động phân bổ bộ nhớ từ ảo sang vật lý

Kết quả: Sau bài tập này, học sinh có thể hiểu được một phần nguyên lý của một hệ điều hành đơn giản. Học sinh có thể hiểu và vẽ được vai trò của các module chính của hệ điều hành.

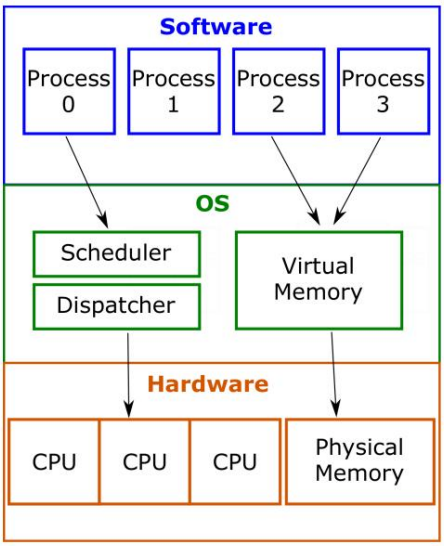
Nội dung

1 Giới thiệu

1.1 Tổng quan

Bài tập này là về việc mô phỏng một hệ điều hành đơn giản để giúp học sinh hiểu các khái niệm cơ bản về lập lịch, đồng bộ hóa và quản lý bộ nhớ. Hình 1 cho thấy kiến trúc tổng thể của hệ điều hành mà chúng ta sẽ triển khai. Nhìn chung, hệ điều hành phải quản lý hai tài nguyên ảo: CPU(s) và RAM bằng hai thành phần cốt lõi:

- Bộ lập lịch (và Bộ điều phối): xác định quy trình nào được phép chạy trên CPU nào.
- Công cụ bộ nhớ ảo (VME): cô lập không gian bộ nhớ của mỗi tiến trình khỏi các tiến trình khác. RAM vật lý được chia sẻ bởi nhiều tiến trình nhưng mỗi tiến trình không biết đến sự tồn tại của tiến trình khác. Điều này được thực hiện bằng cách để mỗi tiến trình có không gian bộ nhớ ảo riêng và Công cụ bộ nhớ ảo sẽ ánh xạ và dịch các địa chỉ ảo do các tiến trình cung cấp sang các địa chỉ vật lý tương ứng.



Hình 1: Tổng quan về các mô-đun chính trong bài tập này

Thông qua các mô-đun đó, hệ điều hành cho phép nhiều quy trình do người dùng tạo ra chia sẻ và sử dụng tài nguyên điện toán ảo. Do đó, trong bài tập này, chúng tôi tập trung vào việc triển khai bộ lập lịch/bộ điều phối và công cụ bộ nhớ ảo.

1.2 Mã nguồn

Sau khi tải xuống mã nguồn của bài tập trong phần Tài nguyên trên nền tảng cổng thông tin và giải nén nó, bạn sẽ thấy mã nguồn được sắp xếp như sau.

- Các tập tin tiêu đề

  - timer.h: Định nghĩa bộ đếm thời gian cho toàn bộ hệ thống. - cpu.h: Định nghĩa các hàm được sử dụng để triển khai CPU ảo
  - queue.h: Các hàm được sử dụng để triển khai hàng đợi chứa PCB của các tiến trình
  - sched.h: Xác định các hàm được sử dụng bởi trình lập lịch
  - mem.h: Các hàm được sử dụng bởi Virtual Memory Engine.

- loader.h: Các hàm được trình tải sử dụng để tải chương trình từ đĩa vào bộ nhớ
- common.h: Định nghĩa các cấu trúc và hàm được sử dụng ở mọi nơi trong hệ điều hành.
- bitopts.h: Xác định các hoạt động trên dữ liệu bit.
- os-mm.h, mm.h: Định nghĩa cấu trúc và dữ liệu cơ bản cho Quản lý bộ nhớ dựa trên phân trang.
- os-cfg.h: (Tùy chọn) Xác định hằng số được sử dụng để chuyển đổi cấu hình phần mềm.

• Các tập tin nguồn

- timer.c: Triển khai bộ đếm thời gian.
- cpu.c: Triển khai CPU ảo.
- queue.c: Triển khai các hoạt động trên hàng đợi (ưu tiên). -

paging.c: Sử dụng để kiểm tra chức năng của Virtual Memory Engine.

- os.c: Toàn bộ hệ điều hành bắt đầu chạy từ tệp này.
- loader.c: Triển khai trình tải
- sched.c: Triển khai trình lập lịch
- mem.c: Triển khai RAM và Virtual Memory Engine
- mm.c, mm-vm.c, mm-memphy.c: Triển khai Quản lý bộ nhớ dựa trên phân trang

• Tệp Make

• đầu vào Mẫu đầu vào được sử dụng để xác minh

• đầu ra Mẫu đầu ra của hệ điều hành.

1.3 Quy trình

Chúng ta sẽ xây dựng một hệ điều hành đa nhiệm cho phép nhiều tiến trình chạy đồng thời, do đó, đáng để dành một ít thời gian để giải thích về tổ chức các tiến trình. Hệ điều hành quản lý các tiến trình thông qua PCB của chúng được mô tả như sau:

```
// Từ include/common.h
cấu trúc pcb_t {
        uint32_t pid;
        uint32_t ưu tiên; uint32_t
        code_seg_t * mã; addr_t regs[10];
        uint32_t pc; #i fd ef
        MLQ_SCHED uint32_t
prio;

#kết thúcnếu
        struct page_table_t * page_table; /* lỗi thời không tương thích với MM_PAGING*/ uint32_t bp;

}
```

Ý nghĩa của các trường trong struct:

• PID: PID của tiến trình

• mức độ ưu tiên: Mức độ ưu tiên của quy trình, giá trị càng thấp thì mức độ ưu tiên của quy trình càng cao. Mức độ ưu tiên kế thừa này phụ thuộc vào thuộc tính của tiến trình và được cố định trong suốt phiên thực thi.

- mã: Đoạn văn bản của quy trình (Để đơn giản hóa mô phỏng, chúng tôi không đặt đoạn văn bản vào ĐẬP).
- regs: Thanh ghi, mỗi tiến trình có thể sử dụng tối đa 10 thanh ghi được đánh số từ 0 đến 9.
- pc: Vị trí hiện tại của bộ đếm chương trình.
- bảng trang: Bản dịch từ địa chỉ ảo sang địa chỉ vật lý (đã lỗi thời, không sử dụng) [1].
- bp: Con trỏ ngắt, dùng để quản lý phân đoạn heap.
- prio: Mức độ ưu tiên thực thi (nếu được hỗ trợ) và giá trị này ghi đè lên mức độ ưu tiên mặc định.

Tương tự như quy trình thực tế, mỗi quy trình trong mô phỏng này chỉ là một danh sách các lệnh được CPU thực hiện lần lượt từ đầu đến cuối (chúng tôi không triển khai lệnh nhảy ở đây). Có năm lệnh mà một quy trình có thể thực hiện:

- CALC: thực hiện một số phép tính bằng CPU. Lệnh này không có đối số.

  Chú thích vùng bộ nhớ: Một vùng lưu trữ nơi chúng ta phân bổ không gian lưu trữ cho một biến, thuật ngữ này thực sự được liên kết với một chỉ mục của BẢNG BIỂU TƯỢNG và thường hỗ trợ khả năng đọc được của con người thông qua tên biến và cơ chế ánh xạ. Thật không may, ánh xạ này nằm ngoài phạm vi của khóa học Hệ điều hành này. Nó có thể thuộc về một khóa học khác giải thích cách trình biên dịch thực hiện công việc của mình và ánh xạ nhãn vào chỉ mục liên kết của nó. Để đơn giản, chúng tôi tham chiếu ở đây một vùng bộ nhớ thông qua chỉ mục của nó và nó có giới hạn về số lượng biến trong mỗi chương trình/quy trình.

- ALLOC: Phân bổ một số khối bộ nhớ (RAM). Cú pháp lệnh:

```
phân bổ [kích thước] [reg]
```

  trong đó size là số byte mà tiến trình muốn phân bổ từ RAM và reg là số thanh ghi sẽ lưu địa chỉ của byte đầu tiên của vùng bộ nhớ được phân bổ. Ví dụ, lệnh alloc 124 7 sẽ phân bổ 124 byte từ OS và địa chỉ của byte đầu tiên trong số 124 byte đó sẽ được lưu trữ tại thanh ghi #7.

- FREE Giải phóng bộ nhớ được phân bổ. Cú pháp:

```
miễn phí [reg]
```

  trong đó reg là số thanh ghi lưu giữ địa chỉ của byte đầu tiên trong vùng bộ nhớ cần giải phóng.

- READ Đọc một byte từ bộ nhớ. Cú pháp:

```
đọc [nguồn] [bù trừ] [đích]
```

  Lệnh đọc một byte bộ nhớ tại địa chỉ bằng giá trị của nguồn thanh ghi + offset và lưu vào đích. Ví dụ, giả sử giá trị của thanh ghi #1 là 0x123 thì lệnh đọc 1 20 2 sẽ đọc một byte bộ nhớ tại địa chỉ 0x123 + 14 (14 là 20 trong hệ thập lục phân) và lưu vào thanh ghi #2.

- WRITE Ghi một thanh ghi giá trị vào bộ nhớ. Cú pháp:

```
ghi [dữ liệu] [đích] [bù trừ]
```

  Hướng dẫn ghi dữ liệu vào địa chỉ bằng giá trị của đích thanh ghi + độ lệch.
  Ví dụ, giả sử giá trị của thanh ghi số 1 là 0x123 thì lệnh write 10 1 20 sẽ ghi 10 vào bộ nhớ tại địa chỉ 0x123 + 14 (14 bằng 20 trong hệ thập lục phân).

[1] không phù hợp với MM PAGING, hãy tham khảo phần 2.2.5 đã đề cập

1.4 Làm thế nào để tạo ra một quy trình?

Nội dung của mỗi tiến trình thực chất là một bản sao của chương trình được lưu trữ trên đĩa. Do đó, để tạo một tiến trình, trước tiên chúng ta phải tạo chương trình mô tả nội dung của nó. Một chương trình được định nghĩa bằng một tệp duy nhất có định dạng sau:

```
[ưu tiên] [N = số hướng dẫn]
hướng dẫn 0
hướng dẫn 1
...
hướng dẫn N-1
```

trong đó priority là mức ưu tiên mặc định của quy trình được tạo từ chương trình này. Cần nhắc lại rằng hệ thống này sử dụng cơ chế ưu tiên kép.
Tiến trình có mức ưu tiên càng cao (với giá trị càng nhỏ), thì tiến trình đó càng có nhiều khả năng được CPU chọn từ hàng đợi (Xem phần 2.1 để biết thêm chi tiết). N là số lệnh và mỗi dòng tiếp theo trong N dòng là lệnh được biểu diễn theo định dạng đã đề cập ở phần trước. Bạn có thể mở các tệp trong thư mục input/proc để xem một số chương trình mẫu.

Cơ chế ưu tiên kép Xin hãy nhớ rằng giá trị mặc định này có thể bị ghi đè bởi mức ưu tiên trực tiếp trong quá trình gọi thực thi quy trình. Để giải quyết xung đột, khi nó có mức ưu tiên trong quá trình tải quy trình (tệp đầu vào này), nó sẽ ghi đè và thay thế mức ưu tiên mặc định trong tệp mô tả quy trình.

1.5 Cách chạy mô phỏng

Những gì chúng ta sẽ làm trong bài tập này là triển khai một hệ điều hành đơn giản và mô phỏng nó trên phần cứng ảo.
Để bắt đầu quá trình mô phỏng, chúng ta phải tạo một tệp mô tả trong thư mục đầu vào về phần cứng và môi trường mà chúng ta sẽ mô phỏng. Tệp mô tả được định nghĩa theo định dạng sau:

```
[khoảng thời gian] [N = Số CPU] [M = Số quy trình cần chạy] [thời gian 0] [đường dẫn 0] [mức độ ưu tiên 0] [thời
gian 1] [đường dẫn 1] [mức độ ưu tiên 1]

...
[thời gian M-1] [đường dẫn M-1] [ưu tiên M-1]
```

trong đó time slice là khoảng thời gian (tính bằng giây) mà một tiến trình được phép chạy. N là số CPU khả dụng và M là số tiến trình cần chạy. Tham số cuối cùng priority là mức ưu tiên trực tiếp khi tiến trình được gọi và điều này sẽ ghi đè lên mức ưu tiên mặc định trong tệp mô tả tiến trình (tham khảo phần 1.4).

Từ dòng thứ hai trở đi, mỗi dòng biểu diễn thời gian đến của tiến trình, nội dung , đường dẫn đến tập tin chứa của chương trình cần tải và mức độ ưu tiên của nó. Bạn có thể tìm thấy các tệp cấu hình tại thư mục đầu vào.

Cần nhắc lại rằng hệ thống này trang bị cơ chế ưu tiên kép. Nếu bạn không có mức ưu tiên mặc định thì chúng ta không có đủ tài liệu để giải quyết xung đột trong quá trình lập lịch. Nhưng nếu giá trị này cố định, nó sẽ hạn chế các thuật toán mà mô phỏng có thể minh họa cho lý thuyết. Xác minh với môi trường thực tế của bạn, có nhiều hệ thống ưu tiên khác nhau, một hệ thống liên quan đến chương trình hệ thống so với chương trình người dùng trong khi hệ thống còn lại cũng cho phép bạn thay đổi mức ưu tiên trực tiếp.

Để bắt đầu mô phỏng, trước tiên bạn phải biên dịch mã nguồn bằng lệnh Make all. Sau đó, chạy lệnh

```
./os [tệp cấu hình]
```

trong đó tệp cấu hình là đường dẫn đến tệp cấu hình cho môi trường mà bạn muốn chạy và nó phải được liên kết với tên của tệp mô tả được đặt trong thư mục đầu vào.

2 Thực hiện

2.1 Lập lịch

Đầu tiên, chúng tôi triển khai trình lập lịch. Hình 2 cho thấy cách hệ điều hành lập lịch các tiến trình. Hệ điều hành được thiết kế để hoạt động trên nhiều bộ xử lý. Hệ điều hành sử dụng nhiều hàng đợi được gọi là hàng đợi sẵn sàng để xác định tiến trình nào sẽ được thực thi khi một CPU khả dụng. Mỗi hàng đợi được liên kết với một giá trị ưu tiên cố định. Trình lập lịch được thiết kế dựa trên thuật toán "hàng đợi đa cấp" được sử dụng trong Linux kernel2 .

Theo Hình 2, bộ lập lịch hoạt động như sau. Đối với mỗi chương trình mới, bộ nạp sẽ tạo một quy trình mới và gán một PCB mới cho quy trình đó. Sau đó, bộ nạp sẽ đọc và sao chép nội dung của chương trình vào đoạn văn bản của quy trình mới (được trỏ bằng con trỏ mã trong PCB của quy trình - mục 1.3). PCB của quy trình được đẩy đến hàng đợi sẵn sàng liên quan có cùng mức ưu tiên với giá trị prio của quy trình này. Sau đó, nó đợi CPU. CPU chạy các quy trình theo kiểu luân phiên. Mỗi quy trình được phép chạy theo lát cắt thời gian. Sau đó, CPU buộc phải đưa quy trình trở lại hàng đợi sẵn sàng có mức ưu tiên liên quan. Sau đó, CPU chọn một quy trình khác từ hàng đợi sẵn sàng và tiếp tục chạy.

Trong hệ thống này, chúng tôi triển khai chính sách Multi-Level Queue (MLQ). Hệ thống chứa các mức ưu tiên MAX PRIO. Mặc dù hệ thống thực, tức là hạt nhân Linux, có thể nhóm các mức này thành các tập hợp con, chúng tôi vẫn giữ nguyên thiết kế trong đó mỗi mức ưu tiên được giữ bởi một hàng đợi sẵn sàng để đơn giản hóa. Chúng tôi đơn giản hóa hàng đợi thêm và đặt proc bằng cách đặt quy trình proc vào hàng đợi sẵn sàng được phân bổ bằng cách khớp mức ưu tiên. Thiết kế chính thuộc về chính sách MLQ được triển khai bởi get proc để lấy một proc rồi phân phối CPU.

Mô tả về chính sách MLQ: bước được duyệt của danh sách hàng đợi sẵn sàng là một số cố định được xây dựng dựa trên mức độ ưu tiên, tức là slot = (MAX PRIO - prio), mỗi hàng đợi chỉ có slot cố định để sử dụng CPU và khi nó được sử dụng hết, hệ thống phải thay đổi tài nguyên sang quy trình khác trong hàng đợi tiếp theo và để lại công việc còn lại cho slot trong tương lai mặc dù nó cần một vòng hoàn thành của hàng đợi sẵn sàng.
Một ví dụ trong Linux MAX PRIO=140, prio=0..(MAX PRIO - 1)

```
prio = 0              |            1              | ....        | MAX_PRIO - 1 |
khe = MAX_PRIO | MAX_PRIO - 1                     | ....                1
```

Chính sách MLQ chỉ thực hiện bước cố định để duyệt qua toàn bộ hàng đợi trong danh sách hàng đợi sẵn sàng ưu tiên.
Công việc của bạn trong phần này là triển khai thuật toán này bằng cách hoàn thành các chức năng sau

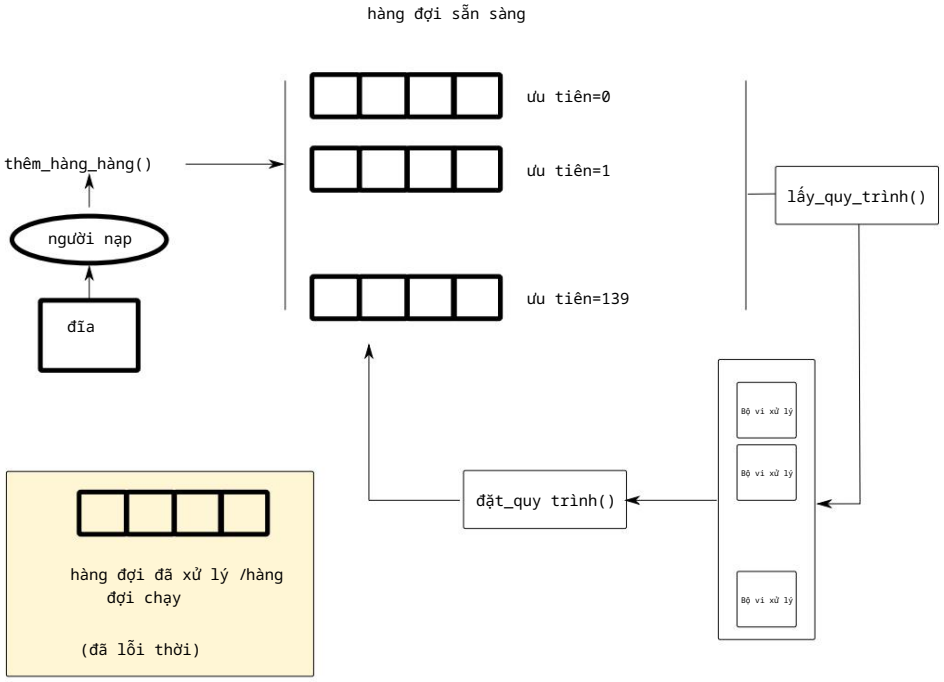
- enqueue() và dequeue() (trong queue.c): Chúng tôi đã định nghĩa một struct (queue t) cho một hàng đợi ưu tiên tại queue.h. Nhiệm vụ của bạn là triển khai các hàm đó để đưa một PCB mới vào hàng đợi và lấy PCB 'lần lượt' tiếp theo ra khỏi hàng đợi.

- get proc() (trong sched.c): lấy PCB của một tiến trình đang chờ từ hệ thống hàng đợi sẵn sàng.
  hàng đợi sẵn sàng được chọn 'lần lượt' đã được mô tả trong chính sách trên.

Bạn có thể so sánh kết quả của mình với các câu trả lời mẫu trong thư mục đầu ra. Lưu ý rằng vì trình tải và trình lập lịch chạy đồng thời nên có thể có nhiều hơn một câu trả lời đúng cho mỗi bài kiểm tra.

Lưu ý: hàng đợi chạy là thứ không tương thích với lý thuyết và đã lỗi thời trong một thời gian. Chúng ta không cần nó trong cả mô hình lý thuyết và triển khai mã, nó là một mã lỗi thời/cũ nhưng chúng ta

2Trên thực tế, Linux hỗ trợ cơ chế phản hồi cho phép di chuyển tiến trình giữa các hàng đợi ưu tiên nhưng chúng tôi không triển khai cơ chế phản hồi ment ở đây



Hình 2: Hoạt động của trình lập lịch trong nhiệm vụ

vẫn giữ lại để tránh phát hiện lỗi sau này.

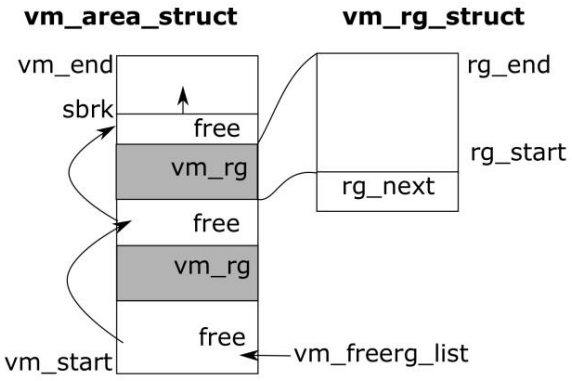
Câu hỏi: Ưu điểm của chiến lược lập lịch được sử dụng trong bài tập này so với các thuật toán lập lịch khác mà bạn đã học là gì?

2.2 Quản lý bộ nhớ

2.2.1 Ánh xạ bộ nhớ ảo trong mỗi tiến trình

Không gian bộ nhớ ảo được tổ chức như một ánh xạ bộ nhớ cho mỗi PCB tiến trình. Theo quan điểm của tiến trình, địa chỉ ảo bao gồm nhiều vùng vm (liên tiếp). Trong thế giới thực, mỗi vùng có thể hoạt động như đoạn mã, ngăn xếp hoặc heap. Do đó, tiến trình giữ trong pcb của nó một con trỏ của nhiều vùng liên tiếp
vùng nhớ.

Vùng nhớ Mỗi vùng nhớ có phạm vi liên tục trong [vm start, vm end]. Mặc dù không gian trải dài toàn bộ phạm vi, nhưng vùng có thể sử dụng thực tế bị giới hạn bởi đỉnh trỏ đến sbrk. Trong vùng giữa vm start và sbrk, có nhiều vùng được struct vm rg struct nắm bắt và các khe trống được danh sách vm freerg theo dõi. Thông qua thiết kế này, chúng ta sẽ thực hiện phân bổ bộ nhớ vật lý thực tế chỉ trong vùng có thể sử dụng, như trong Hình 3.



Hình 3: Cấu trúc của vùng và miền vm

```
//Từ include/os-mm.h /*

  * Cấu trúc vùng nhớ
  */
5 struct vm_rg_struct { rg_start
      dài không dấu ; rg_end dài không
      dấu ;

      cấu trúc vm_rg_struct *rg_next;
10 };

/*
  * Cấu trúc vùng nhớ */ 15 struct

vm_area_struct { vm_id dài không dấu ;
      vm_start dài không dấu ; vm_end
      dài không dấu ;



20      sbrk dài không dấu ;
/*
  * Trường phái sinh
  * không dấu dài vm_limit = vm_end - vm_start
  */
25      cấu trúc mm_struct *vm_mm; cấu trúc
      vm_rg_struct *vm_freerg_list; cấu trúc vm_area_struct
      *vm_next;
};
```

Vùng nhớ Như chúng tôi đã lưu ý trong phần trước 1.3, các vùng này thực sự được đóng vai trò là các biến trong mã nguồn của chương trình mà con người có thể đọc được. Do thực tế hiện tại nằm ngoài phạm vi, chúng tôi chỉ đề cập đến khái niệm không gian tên theo thuật ngữ lập chỉ mục. Chúng tôi chưa được trang bị đủ nguyên tắc của trình biên dịch. Một lần nữa, thật quá sức khi sử dụng một bảng ký hiệu phức tạp như vậy trong khóa học hệ điều hành này. Chúng tôi tạm thời hình dung các vùng này như một tập hợp số lượng vùng giới hạn. Chúng tôi quản lý chúng bằng cách sử dụng một mảng symrgtbl[PAGING MAX SYMTBL SZ]. Kích thước mảng được cố định bởi một hằng số, PAGING MAX SYMTBL SZ, biểu thị số lượng biến được phép trong mỗi chương trình. Để kết thúc, chúng tôi sử dụng struct vm rg struct symrgtbl để giữ điểm bắt đầu và điểm kết thúc của vùng và con trỏ rg next được dành riêng cho tương lai

thiết lập theo dõi.

```
//Từ include/os-mm.h /*

  * Cấu trúc ánh xạ bộ nhớ */

cấu trúc mm_struct {
    uint32_t *pgd;

    cấu trúc vm_area_struct *mmap;

    /* Hiện tại chúng tôi hỗ trợ một số lượng ký hiệu cố định */ struct vm_rg_struct
    symrgtbl[PAGING_MAX_SYMTBL_SZ];

    cấu trúc pgn_t *fifo_pgn;
};
```

Ánh xạ bộ nhớ được biểu diễn bằng struct mm struct, giữ theo dõi tất cả các vùng bộ nhớ đã đề cập trong một vùng bộ nhớ liền kề tách biệt. Trong mỗi struct ánh xạ bộ nhớ, nhiều vùng bộ nhớ được chỉ ra bởi struct vm area struct *mmap list. Trường quan trọng tiếp theo là pgd, là thư mục bảng trang, chứa tất cả các mục bảng trang. Mỗi mục là một bản đồ giữa số trang và số khung trong hệ thống quản lý bộ nhớ phân trang. Chúng tôi giữ ánh xạ khung trang chi tiết cho phần sau 2.2.3. Symrgtbl là một triển khai 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, tức là người gọi, trang fifo (để tham chiếu), vì vậy chúng tôi để nó ở đó và bạn có thể sử dụng riêng hoặc chỉ cần loại bỏ nó.

CPU định địa chỉ địa chỉ do CPU tạo ra để truy cập vào một vị trí bộ nhớ cụ thể. Trong hệ thống dựa trên phân trang, nó được chia thành:

- Số trang (p): được sử dụng như một chỉ mục vào bảng trang lưu trữ địa chỉ cơ sở cho mỗi trang trong bộ nhớ vật lý.
- Độ lệch trang (d): kết hợp với địa chỉ cơ sở để xác định địa chỉ bộ nhớ vật lý được gửi đến Đơn vị quản lý bộ nhớ

Hình 4: Địa chỉ CPU

Không gian địa chỉ vật lý của một tiến trình có thể không liền kề. Chúng tôi chia bộ nhớ vật lý thành các khối có kích thước cố định (khung) với hai kích thước 256B hoặc 512B. Chúng tôi đề xuất nhiều kết hợp cài đặt khác nhau trong Bảng 1 và kết thúc bằng cấu hình được tô sáng. Đây là cài đặt được 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 4.

Trong bản tóm tắt VM, tất cả các cấu trúc hỗ trợ VM đều được đặt trong mô-đun mm-vm.c.

Câu hỏi: Trong hệ điều hành đơn giản này, chúng tôi triển khai thiết kế nhiều phân đoạn bộ nhớ hoặc vùng bộ nhớ trong khai báo mã nguồn. Ưu điểm của thiết kế đề xuất gồm nhiều phân đoạn là gì?

CPU bus	PAGE kích thước	PAGE bit	Không có mục nhập pg	PAGE mục nhập sz	PAGE TBL bit OFFSET	PGT mem	MEMPHY bit khung	
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

Bảng 1: Giá trị cấu hình bus địa chỉ CPU khác nhau

2.2.2 Bộ nhớ vật lý của hệ thống

Hình 1 cho thấy phần cứng bộ nhớ được cài đặt theo toàn bộ hệ thống. Tất cả các quy trình sở hữu ánh xạ bộ nhớ tách biệt của chúng, nhưng tất cả các ánh xạ đều nhắm đến một thiết bị vật lý đơn lẻ. Có hai các loại thiết bị là RAM và SWAP. Cả hai đều có thể được thực hiện bởi cùng một thiết bị vật lý như trong mm-memphy.c với các thiết lập khác nhau. Các thiết lập được hỗ trợ là truy cập bộ nhớ ngẫu nhiên, truy cập bộ nhớ tuần tự/nối tiếp và dung lượng lưu trữ.
Bất chấp các cấu hình khác nhau có thể có, việc sử dụng hợp lý các thiết bị này có thể được phân biệt. Thiết bị RAM, thuộc hệ thống bộ nhớ chính, có thể được truy cập trực tiếp từ địa chỉ CPU bus, tức là có thể đọc/ghi bằng lệnh CPU. Trong khi đó, SWAP chỉ là một thiết bị bộ nhớ thứ cấp, và tất cả các thao tác dữ liệu được lưu trữ của nó phải được thực hiện bằng cách di chuyển chúng vào bộ nhớ chính. Vì nó thiếu quyền truy cập trực tiếp từ CPU, hệ thống thường trang bị một SWAP lớn với chi phí nhỏ và thậm chí có nhiều hơn một trường hợp. Trong cài đặt của chúng tôi, chúng tôi hỗ trợ phần cứng được cài đặt với một thiết bị RAM và lên đến 4 Thiết bị SWAP.

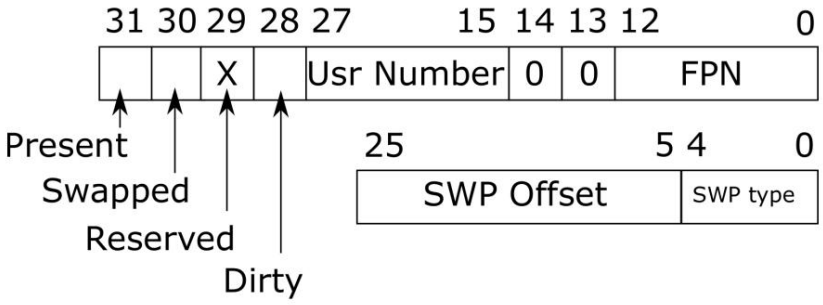
Cấu trúc framephy chủ yếu được sử dụng để lưu trữ số khung hình.
Cấu trúc memphy struct có các trường cơ bản là lưu trữ và kích thước. Trường rdmflg định nghĩa quyền truy cập bộ nhớ được truy cập ngẫu nhiên hoặc tuần tự. Các trường danh sách fp miễn phí và danh sách fp đã sử dụng được dành riêng để giữ lại các khung bộ nhớ chưa sử dụng và đã sử dụng.

```
//Từ include/os-mm.h
/*
  * Cấu trúc VẬT LÝ FRAME/MEM
  */
cấu trúc framephy_struct {
    int fpn;
    cấu trúc framephy_struct *fp_next;
};

cấu trúc memphy_struct {
    /* Trường dữ liệu cơ bản và kích thước */
    BYTE *lưu trữ;
    số nguyên maxsz;

    /* Các trường thiết bị tuần tự */
    số nguyên rdmflg;
    int con trỏ;

    /* Cấu trúc quản lý */
    cấu trúc framephy_struct *free_fp_list;
    cấu trúc framephy_struct *used_fp_list;
};
```

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

2.2.3 Sơ đồ dịch địa chỉ dựa trên phân trang

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



Hình 5: Định dạng mục nhập bảng trang.

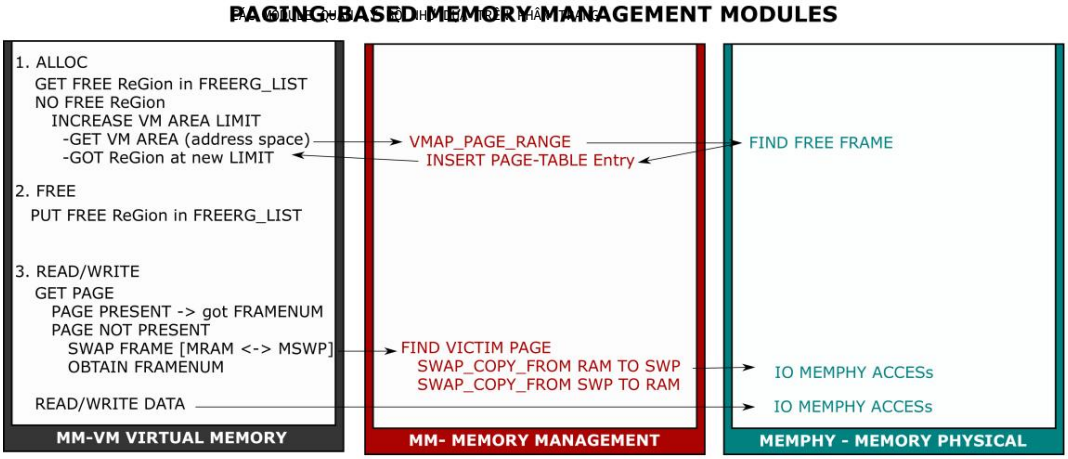
Bảng trang Cấu trúc này cho phép một quy trình không gian người dùng tìm ra khung vật lý mà mỗi trang ảo được ánh xạ tới. Nó chứa một giá trị 32 bit cho mỗi trang ảo, chứa dữ liệu sau:

```
    * Bit 0-12 số khung trang (FPN) nếu có
    * Bit 13-14 bằng 0 nếu có
    * Bit 15-27 được đánh số do người dùng xác định nếu có
    * Bit 0-4 hoán đổi loại nếu hoán đổi
5   * Bit 5-25 hoán đổi bù trừ nếu hoán đổi bẩn
    * Câu 28
    * Bit 29              kín đáo
    * Câu 30              trao đổi
    * Câu 31              trình bày
```

Không gian ảo được cô lập với từng thực thể sau đó mỗi struct pcb t có bảng riêng. Để làm việc trong hệ thống bộ nhớ dựa trên phân trang, chúng ta cần cập nhật struct này và phần sau sẽ thảo luận về sửa đổi cần thiết.
Trong mọi trường hợp, mỗi tiến trình đều có một không gian hoàn toàn biệt lập và duy nhất, N tiến trình trong thiết lập của chúng tôi tạo ra N bảng trang và ngược lại, mỗi trang phải có tất cả các mục nhập cho toàn bộ không gian địa chỉ CPU. Đối với mỗi mục nhập, số phân trang có thể có một khung liên kết trong MEMRAM hoặc MEMSWP hoặc có thể có giá trị null, chức năng của từng bit dữ liệu của mục nhập bảng trang được minh họa trong Hình 5. Trong thiết lập được tô sáng đã chọn của chúng tôi trong Bảng 1, chúng tôi có bảng 16.000 mục nhập, mỗi bảng tốn 64 KB dung lượng lưu trữ.

Trong phần 2.2.1, tiến trình có thể truy cập không gian bộ nhớ ảo theo cách liên tiếp của cấu trúc vùng vm.
Công việc còn lại liên quan đến việc ánh xạ giữa trang và khung để cung cấp không gian bộ nhớ liền kề trên cơ chế lưu trữ khung rời rạc. Nó thuộc về hai cách tiếp cận chính là hoán đổi bộ nhớ và các hoạt động bộ nhớ cơ bản, tức là phân bổ/giải phóng/đọc/ghi, chủ yếu giữ liên lạc với cấu trúc bảng trang pgd.

Hoán đổi bộ nhớ o Chúng tôi đã được thông báo rằng một vùng bộ nhớ (/segment) có thể không được sử dụng hết không gian lưu trữ giới hạn của nó. Điều đó có nghĩa là có những không gian lưu trữ không được ánh xạ tới MEMRAM.



Hình 6: Các mô-đun hệ thống bộ nhớ

swapping có thể giúp di chuyển nội dung của khung vật lý giữa MEMRAM và MEMSWAP. Swapping là cơ chế thực hiện sao chép nội dung của khung từ bên ngoài vào ram bộ nhớ chính. Swapping out, ngược lại, cố gắng di chuyển nội dung của khung trong MEMRAM sang MEMSWAP. Trong bối cảnh thông thường, swapping giúp chúng ta có được khung RAM trống vì kích thước của thiết bị SWAP thường đủ lớn.

Các hoạt động bộ nhớ cơ bản trong hệ thống dựa trên phân trang

- ALLOC trong hầu hết các trường hợp, nó phù hợp với vùng có sẵn. Nếu không có không gian phù hợp như vậy, chúng ta cần nâng rào cản sbrk lên và vì nó chưa bao giờ được chạm vào, nó có thể cần cung cấp một số khung vật lý và sau đó ánh xạ chúng bằng cách sử dụng Page Table Entry.
- GIẢI PHÓNG không gian lưu trữ liên quan đến id vùng. Vì chúng tôi không thể thu thập lại khung vật lý đã lấy có thể gây ra lỗ hổng bộ nhớ, chúng tôi chỉ giữ không gian lưu trữ đã thu thập trong danh sách giải phóng để yêu cầu phân bổ thêm.
- READ/WRITE yêu cầu phải đưa trang vào bộ nhớ chính. Bước tốn nhiều tài nguyên nhất là hoán đổi trang. Nếu trang nằm trong thiết bị MEMSWAP, cần đưa trang đó trở lại thiết bị MEMRAM (hoán đổi vào) và nếu thiếu không gian, chúng ta cần trả lại một số trang cho thiết bị MEMSWAP (hoán đổi ra) để tạo thêm chỗ trống.

Để thực hiện các hoạt động này, cần có sự hợp tác giữa các mô-đun của mm như minh họa trong Hình 6.

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

2.2.4 Bộ đệm tra cứu dịch thuật (TLB)

Trong phần trước, hệ điều hành và hệ thống con quản lý bộ nhớ triển khai rằng mỗi tiến trình có bảng trang riêng. Bảng này chứa mục nhập bảng trang cung cấp số khung. Thách thức nằm ở việc tối ưu hóa thời gian truy cập cho các mục nhập này, thời gian truy cập gấp 2 lần khi đọc bảng trang (thực tế được đặt trên bộ nhớ chính hoặc thực tế là... MEMPHY) và truy cập dữ liệu bộ nhớ trong MEMPHY.

Với quy mô tiến trình lớn dẫn đến chi phí cao, TLB được đề xuất để tận dụng khả năng bộ nhớ đệm do bản chất tốc độ cao của nó. TLB thường là MEMPHY nhưng hoạt động như bộ nhớ đệm tốc độ cao cho các mục bảng trang. Tuy nhiên, bộ nhớ đệm là thành phần có chi phí cao; do đó, nó có dung lượng hạn chế. Đây là những công việc cơ bản của TLB:

- Phương pháp truy cập TLB: TLB là MEMPHY hỗ trợ cơ chế ánh xạ để xác định cách nội dung được liên kết với thông tin định danh. Trong công trình này, chúng tôi tận dụng kiến thức được trang bị của khóa học trước về phần cứng máy tính, trong đó sử dụng các kỹ thuật ánh xạ bộ nhớ đệm: ánh xạ trực tiếp/liên kết tập hợp/liên kết hoàn toàn.

- Thiết lập TLB: TLB chứa các mục bảng trang được sử dụng gần đây. Khi CPU tạo ra một địa chỉ ảo, nó kiểm tra TLB:

  – Nếu có mục bảng trang (một lần trúng TLB), số khung tương ứng sẽ được truy xuất, – Nếu không tìm thấy mục bảng trang (một lần lỡ TLB), số trang sẽ được sử dụng làm chỉ mục để truy cập bảng trang trong bộ nhớ chính. Nếu trang không có trong bộ nhớ chính, lỗi trang sẽ xảy ra và TLB sẽ được cập nhật với mục trang mới.

- TLB trúng đích:

  – CPU tạo ra một địa chỉ ảo.

  – TLB được kiểm tra (mục nhập hiện tại).

  – Lấy lại số khung hình tương ứng.

- TLB Miss:

  – CPU tạo ra một địa chỉ ảo.

  – TLB được kiểm tra (mục nhập không có).

  – Số trang được khớp với bảng trang trong bộ nhớ chính.

  – Số khung tương ứng được lấy lại

Vì TLB chỉ là một thiết bị lưu trữ bộ nhớ, bạn có thể thiết kế một phương pháp hiệu quả để xác định ánh xạ bộ nhớ đệm. Chúng tôi không sửa thiết kế, Chúng tôi chỉ cung cấp một đề xuất về đề xuất ánh xạ dựa trên pid và số trang, lưu ý rằng vì bộ nhớ đệm TLB được chia sẻ bởi tất cả các quy trình hệ thống và được dùng ở cấp CPU; do đó, pid là cần thiết.

```
/*
  * tlb_cache_read đọc thiết bị bộ nhớ đệm TLB
  * @mp: cấu trúc memphy * @pid:
  id tiến trình * @pgnum: số
5 trang
  * @value: giá trị thu được
  */
int tlb_cache_read(struct memphy_struct * mp, int pid, int pgnum, giá trị BYTE)(...)

10 /*
  * tlb_cache_write ghi thiết bị bộ nhớ đệm TLB * @mp: cấu trúc memphy *
  @pid: id tiến trình * @pgnum: số
  trang * @value: giá trị thu được
  */
15


int tlb_cache_write(struct memphy_struct *mp, int pid, int pgnum, giá trị BYTE)(...)
```

Các hoạt động TLB (tlballoc/tlbfree/tlbread/tlbwrite) diễn ra trước khi phân trang bộ nhớ hoạt động (alloc/ free/ read/ write). Sơ đồ dịch chuyển địa chỉ qua các lớp này theo thứ tự cụ thể khi người dùng gửi yêu cầu bộ nhớ và sau đó theo thứ tự ngược lại khi nhận được địa chỉ. Do mô hình lập trình phân cấp gốc của nó, nếu bạn nghĩ rằng nó sẽ giúp mọi thứ dễ dàng hơn, bạn có thể tự do triển khai các bản cập nhật TLB CACHED

trong bất kỳ mô-đun nào (tlb hoặc mm-vm) thông qua việc bỏ qua giữa các mô-đun. Chúng tôi đã dành riêng một bit được thiết kế để tạo điều kiện cho mục đích thao tác trang, chúng tôi hy vọng rằng phần được dành riêng sẽ hữu ích.

Đối với một hoạt động chung (thay thế xxx bằng alloc/free/read/write)

```
int tlbxxx(struct pcb_t *proc, kích thước uint32_t, chỉ số reg_index uint32_t)
{
   int địa chỉ, giá trị;

   /* TODO trước khi cập nhật TLB ĐÃ ĐƯỢC ĐỆM LẠI (nếu cần) */ /* bằng cách sử dụng
   tlb_cache_read()/tlb_cache_write()*/

   /* Thực hiện các hoạt động phân trang bằng cách sử dụng vmaid = 0 theo mặc định */ val =
   __xxx(proc, 0, reg_index, size, &addr);

   /* TODO cập nhật hậu tố TLB ĐÃ ĐƯỢC LƯU TRỮ (nếu cần) )*/

   trả về giá trị;
}
```

Đối với phát triển nâng cao, trong cộng đồng nhà phát triển thực sự, nó không liên kết với TLB nhưng là phương pháp bộ nhớ đệm gốc nơi các nhà sản xuất cạnh tranh về hiệu suất và tỷ lệ trúng đích. Để cung cấp hiệu suất bộ nhớ đệm trúng đích/bỏ lỡ tốt hơn, do đó họ kiểm tra kỹ lưỡng từng nội dung được lưu trong bộ nhớ đệm.

- Tạo bộ nhớ đệm: trong một hệ thống xác thực, trước khi xóa hoặc cập nhật nhẹ tài liệu bộ nhớ đệm, chúng sẽ lưu trữ một phiên bản hoặc thế hệ được đặt tên trong bộ nhớ đệm (tlb gen nếu bạn cần tham chiếu) và xác minh rằng thế hệ đã trải qua sửa đổi bộ nhớ đáng kể. Đây là một lợi thế của bố cục tuyệt vời của bạn, nhưng chúng ta có thể loại trừ nó đối với yêu cầu tối thiểu.

- Ngưỡng địa phương: để duy trì các mục TLB được phân bổ khớp với địa phương hiện tại, chúng tôi thêm một mục nhập sau khi quá trình 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 vừa. TLB nên được sử dụng hiệu quả hơn với bảng trang đa cấp bằng cách giảm số lượng mục phải tìm kiếm. Tuy nhiên, bảng trang đa cấp khó triển khai hơn.

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

2.2.5 Tổng hợp tất cả các triển khai hướng phân trang

Giới thiệu về kiểm soát cấu hình sử dụng định nghĩa hằng số: xử lý sự can thiệp giữa các mô-đun [3] để làm ít nỗ lực hơn chương trình hướng tính năng, chúng tôi áp dụng cùng một cách tiếp cận trong cộng đồng nhà phát triển bằng cách cô lập từng tính năng thông qua một hệ thống cấu hình. Tận dụng cơ chế này, chúng tôi có thể duy trì nhiều hệ thống con riêng biệt, tất cả đều tồn tại trong một phiên bản mã duy nhất. Chúng tôi có thể kiểm soát cấu hình được sử dụng trong chương trình mô phỏng của mình trong tệp include/os-cfg.h

```
// Từ include/os-cfg.h #define
MLQ_SCHED 1 #define MAX_PRIO
140
```

[3] Phần này chủ yếu áp dụng cho quản lý bộ nhớ phân trang. Nếu bạn vẫn đang làm việc trong phần lập lịch, bạn nên giữ nguyên cài đặt mặc định và tránh chạm quá nhiều vào các giá trị này

```
#định nghĩa CPU_TLB
#định nghĩa CPUTLB_FIXED_TLBSZ
#định nghĩa MM_PAGING
#định nghĩa MM_FIXED_MEMSZ
```

Một ví dụ về thiết lập MM PAGING : Với các mô-đun mới này của bộ nhớ phân trang, chúng ta có một dẫn xuất của PCB struct được thêm một số trường quản lý bộ nhớ bổ sung và chúng được bao bọc bởi một định nghĩa hằng số. Nếu chúng ta muốn sử dụng mô-đun MM PAGING thì chúng ta kích hoạt dòng cấu hình #define liên quan trong include/os-cfg.h

```
// Từ include/common.h struct
pcb_t {
        ...
#i fd ef MM_PAGING
        cấu trúc mm_struct *mm; cấu trúc
        memphy_struct *mram; cấu trúc memphy_struct
        **mswp; cấu trúc memphy_struct *active_mswp;

#kết thúc nếu
        ...
};
```

Một ví dụ khác về thiết lập MM_FIXED MEMSZ : Liên quan đến phiên bản mới của PCB struct, tệp mô tả trong đầu vào có thể giữ nguyên thiết lập cũ với #define MM FIXED MEMSZ trong khi nó vẫn hoạt động trong chế độ quản lý bộ nhớ phân trang mới. Cấu hình chế độ này có lợi cho khả năng tương thích ngược với tệp đầu vào phiên bản cũ. Bật thiết lập này cho phép tương thích ngược.

Một ví dụ khác về thiết lập TLB của CPU : Với các mô-đun phân cấp bộ nhớ mới này, chúng ta có được một tập hợp các hoạt động trang khác nhau (cấp phát, giải phóng, đọc và ghi), được đóng gói hoặc mã hóa cứng bởi thiết lập hệ thống vì phần cứng thay đổi do chi phí sản xuất cắt giảm.

```
// Từ src/cpu.c int
run(struct pcb_t * proc) {
        ...
#i fd ef CPU_TLB
                stat = tlbfree_data(proc, ins.arg_0);
#elif đã định nghĩa(MM_PAGING)
                thống kê = pgfree_data(proc, ins.arg_0);
#khác
                thống kê = dữ liệu miễn phí(proc, ins.arg_0);
#kết thúcnếu
        ...
};
```

Cấu trúc hỗ trợ CPU TLB trong cấu trúc PCB được kích hoạt bằng cách sử dụng CPU TLB liên kết với dòng cấu hình xác định trong include/os-cfg.h

```
// Từ include/common.h
cấu trúc pcb_t {
        ...
#i fd ef CPU_TLB
        cấu trúc memphy_struct *tlb;
```

```
#kết thúc nếu
        ...
};
```

Cấu hình mới với khai báo rõ ràng về kích thước bộ nhớ (Hãy cẩn thận, chế độ này hỗ trợ kích thước bộ nhớ tùy chỉnh ngụ ý rằng chúng ta chú thích hoặc xóa hoặc vô hiệu hóa hằng số #define CPUTLB FIXED TLBSZ) và #define MM FIXED MEMSZ) Nếu chúng ta ở chế độ này, thì chương trình mô phỏng sẽ lấy thêm các dòng từ tệp đầu vào. Các dòng đầu vào này chứa kích thước TLB và kích thước bộ nhớ vật lý của hệ thống: MEM-RAM và tối đa 4 MEMSWP. Giá trị kích thước yêu cầu giá trị số nguyên không âm. Chúng ta có thể đặt kích thước bằng 0, nhưng điều đó có nghĩa là hoán đổi bị vô hiệu hóa. Để giữ tham số kích thước bộ nhớ hợp lệ, chúng ta phải có MEMRAM và ít nhất 1 MEMSWAP, các giá trị đó phải là số nguyên dương, các giá trị còn lại có thể được đặt thành 0.

```
[khoảng thời gian] [N = Số CPU] [M = Số quy trình cần chạy]
[CPU_TLB_SZ]
[MEM_RAM_SZ] [MEM_SWP_SZ_0] [MEM_SWP_SZ_1] [MEM_SWP_SZ_2] [MEM_SWP_SZ_3] [lần 0] [đường dẫn 0] [ưu tiên 0] [thời gian
1] [đường dẫn 1] [ưu tiên 1]

...
[thời gian M-1] [đường dẫn M-1] [ưu tiên M-1]
```

Dòng đầu vào được tô sáng được kiểm soát bởi định nghĩa hằng số. Kiểm tra lại tệp đầu vào và nội dung của include/os-cfg.h sẽ giúp chúng ta hiểu cách chương trình mô phỏng hoạt động khi có điều gì đó lạ.

2.3 Kết hợp tất cả lại với nhau

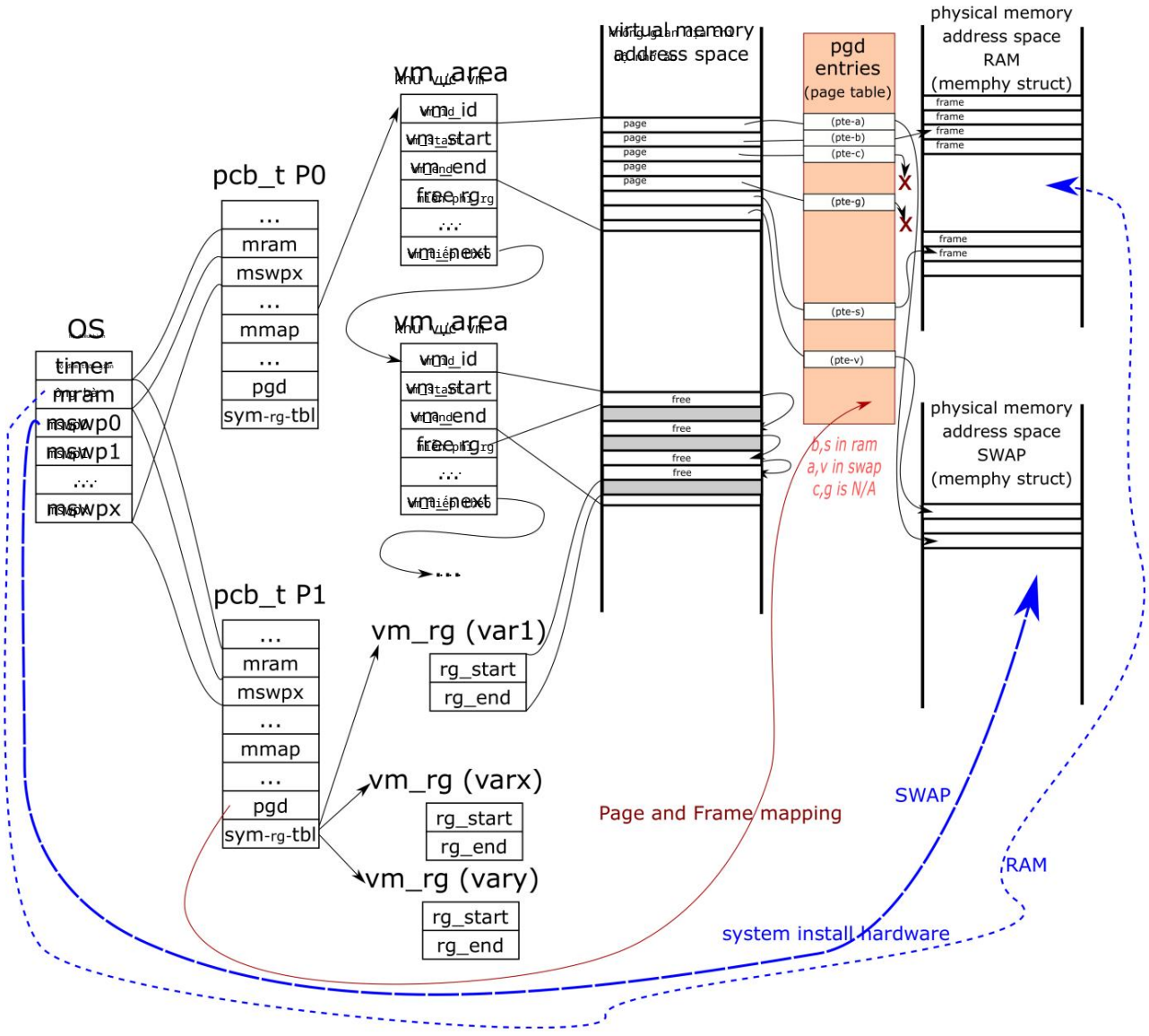
Cuối cùng, chúng ta kết hợp bộ lập lịch và quản lý bộ nhớ để tạo thành một hệ điều hành hoàn chỉnh. Hình 7 cho thấy tổ chức hoàn chỉnh của quản lý bộ nhớ hệ điều hành. Nhiệm vụ cuối cùng cần làm là đồng bộ hóa. Vì hệ điều hành chạy trên nhiều bộ xử lý, nên có khả năng các tài nguyên chia sẻ có thể được truy cập đồng thời bởi nhiều hơn một quy trình cùng một lúc. Nhiệm vụ của bạn trong phần này là tìm tài nguyên chia sẻ và sử dụng cơ chế khóa để bảo vệ chúng.

Kiểm tra công việc của bạn bằng cách biên dịch toàn bộ mã nguồn trước

```
làm tất cả
```

và so sánh kết quả của bạn với kết quả trong output. Hãy nhớ rằng khi chúng ta chạy nhiều quy trình, có thể có nhiều hơn một kết quả đúng. Tất cả các kết quả được sử dụng làm mẫu và không phải là kết quả bị hạn chế.
Kết quả của bạn cần được giải thích và so sánh với khuôn khổ lý thuyết.

Câu hỏi: Điều gì sẽ xảy ra nếu đồng bộ hóa không được xử lý trong hệ điều hành đơn giản của bạn? Minh họa vấn đề của hệ điều hành đơn giản của bạn bằng ví dụ nếu bạn có. Lưu ý: Bạn cần chạy hai phiên bản 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 demo và giải thích sự khác biệt của chúng.



Hình 7: Hoạt động liên quan đến bộ nhớ ảo trong bài tập

3 Đệ trình

3.1 Mã nguồn

Yêu cầu: bạn phải mã hóa lệnh gọi hệ thống theo sau là phong cách mã hóa. Tham khảo: https://
www.gnu.org/prep/standards/html node/Writing-C.html

3.2 Yêu cầu

Bộ lập lịch triển khai bộ lập lịch sử dụng chính sách MLQ như mô tả trong phần 2.1.

Quản lý bộ nhớ triển khai hệ thống phân trang và tập trung vào việc triển khai mô-đun bộ nhớ TLB. Việc xử lý TLB-miss được đề xuất và lựa chọn theo nhóm và bắt buộc phải tiến hành đánh giá hiệu suất tỷ lệ hit/miss.

Sinh viên phải trả lời tất cả các câu hỏi trong phần mô tả bài tập.

3.3 Báo cáo

Viết báo cáo trả lời các câu hỏi trong phần triển khai và diễn giải kết quả chạy thử nghiệm trong từng phần:

- Lên lịch: vẽ sơ đồ Gantt mô tả cách các tiến trình được CPU thực hiện.
- Bộ nhớ: Hiển thị trạng thái của trang được ánh xạ và trang chỉ mục liên quan đến quy trình TLB và xử lý trường hợp xảy ra lỗi mất trang.
- Nhìn chung: sinh viên tìm ra cách riêng để diễn giải kết quả mô phỏng.

Sau khi hoàn thành bài tập, hãy di chuyển báo cáo của bạn đến thư mục mã nguồn và nén toàn bộ thư mục thành một tệp duy nhất có tên là MSSV.zip và gửi cho BKEL.

3.4 Phân loại

Bạn phải thực hiện bài tập này theo nhóm gồm 4 hoặc 5 học sinh. Điểm tổng thể của nhóm bạn là sự kết hợp của hai phần:

- Trình bày (7 điểm)
  - Lên lịch: 3 điểm
  - CPU MMU: phân trang và ánh xạ trực tiếp TLB đơn giản 2,5 điểm
  - CPU TLB hoạt động đầy đủ 1,5 điểm
- Báo cáo (3 điểm)

3.5 Quy tắc đạo đức

Các thành viên khoa tham gia phát triển mã đã giữ toàn bộ bản quyền của mã nguồn dự án.

Cấp phép mã nguồn: Tác giả cấp cho Người được cấp phép quyền sử dụng và sửa đổi Mã nguồn được cấp phép với mục đích duy nhất là học tập trong quá trình tham gia khóa học CO2018.

Lịch sử sửa đổi

Ngày sửa đổi	Tác giả	Sự miêu tả
1.0	01.2019 pdnguyen, Minh Thành CHUNG, NGUYỄN HẢI ĐỨC	Tạo CPU, lập lịch, bộ nhớ
1.1	09.2022 pdnguyen	Thêm Lập lịch CPU cho Hàng đợi đa cấp (MLQ)
2.0	03.2023 pdnguyen	Khởi tạo MM Paging Framework
2.1	10.2023 pdnguyen	Thêm Thay thế Trang
2.2	03.2024 pdnguyen	Thêm Bộ đệm tra cứu dịch thuật CPU (TLB)