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

BÁO CÁO BÀI TẬP LỚN
Môn học: Hệ Điều Hành

SIMPLE OPERATING SYSTEM

Giảng viên hướng dẫn: T.A. Huỳnh Ngọc Như

Nhóm sinh viên thực hiện:

Trần Lê Xuân Ánh	2012628
Nguyễn Phúc Gia Khiêm	2211573
Lò Nhật Tân	1713067
Nguyễn Khánh Vi	2015036

Thành phố Hồ Chí Minh, tháng 05/2024.

MỤC LỤC

1 Giới thiệu chung **2**

2 Bộ lập lịch - Scheduler **4**

2.1 Lý thuyết 4

2.2 Trả lời các câu hỏi 4

2.3 Hiện thực 4

3 Quản lý bộ nhớ - Memory Management **9**

3.1 Lý thuyết 9

3.2 Trả lời các câu hỏi 11

3.3 Hiện thực 13

4 Kết hợp các thành phần - Put It All Together **18**

4.1 Trả lời các câu hỏi 18

5 Các yêu cầu khác **19**

5.1 Quy cách mã nguồn (Coding style) 19

5.2 Tuân thủ quy tắc đạo đức (Code of Ethics) 19

6 Kết luận **20**

DANH MỤC HÌNH ẢNH

3.1 Cấu trúc TLB entry 10
3.2 Hàm tlballoc 16
3.3 Hoạt động alloc của TLB 16
3.4 Lệnh read thông qua TLB 17

DANH MỤC BẢNG BIỂU

1 Bảng phân công nhiệm vụ 1

PHÂN CÔNG NHIỆM VỤ

STT	Sinh viên	MSSV	Vai trò	Mức độ đóng góp
1	Lò Nhật Tân	1713067	• Chuẩn bị workspace[1] • Nghiên cứu, tìm hiểu để bài • Hiện thực phần memory • Viết báo cáo	25%
2	Nguyễn Phúc Gia Khiêm	2211573	• Nghiên cứu, tìm hiểu để bài • Hiện thực phần TLB • Viết báo cáo	25%
3	Trần Lê Xuân Ánh	2012628	• Nghiên cứu, tìm hiểu để bài • Hiện thực phần memory • Viết báo cáo	25%
4	Nguyễn Khánh Vi	2015036	• Nghiên cứu, tìm hiểu để bài • Hiện thực phần scheduler • Viết báo cáo	25%

Bảng 1: Bảng phân công nhiệm vụ

Chương 1: GIỚI THIỆU CHUNG

Trong bài tập lớn này, nhóm sẽ thiết kế và hiện thực một hệ điều hành đa nhiệm đơn giản. Hệ điều hành này cho phép nhiều tiến trình được thực hiện đồng thời.

Các tiến trình (process) chạy trên hệ điều hành này có thể thực hiện một danh sách các lệnh được định nghĩa sẵn trong file mã nguồn của chúng. Có 5 loại lệnh mà một tiến trình có thể thực hiện:

- **CALC**: Thực hiện một số tính toán với CPU. Lệnh này không yêu cầu tham số. Cú pháp của lệnh này là:

  calc

- **ALLOC**: cấp phát bộ nhớ cho một register được chỉ định. Cú pháp của lệnh này là:

  alloc *[size] [reg]*

  Trong đó:
  size: số byte ta muốn cấp phát
  reg: register được cấp phát RAM

- **FREE**: giải phóng vùng nhớ cảu một register đã được cấp phát vùng nhớ trước đó. Cú pháp của lệnh này là:

  free *[reg]*

  Trong đó:
  reg: register sẽ được giải phóng vùng nhớ

- **READ**: đọc giá trị từ một byte và gán nó vào register được chỉ định. Cú pháp của lệnh này là:

  read *[source] [offset] [destination]*

  Trong đó:
  source: register giữ byte đầu của vùng nhớ chứa byte được đọc
  offset: độ dời so với **source**, địa chỉ của byte được đọc sẽ được tính bằng **source** +
  offset destination: register mà giá trị byte được đọc sẽ lưu vào

- **WRITE**: ghi giá trị vào một byte được chỉ định. Cú pháp của lệnh này là:

write *[data] [destination] [offset]*

Trong đó:
data: giá trị cần ghi **destination**: register giữ byte đầu của vùng nhớ chứa byte sẽ được ghi
offset: độ dời so với **source**, địa chỉ của byte được ghi được tính bằng **destination + offset**

Để hệ điều hành mô phỏng có thể chạy các tiến trình, ta cần hiện thực hai thành phần của nó là **Bộ lập lịch** và **Bộ quản lý bộ nhớ**.

Chương 2: BỘ LẬP LỊCH - SCHEDULER

2.1 Lý thuyết

Hệ điều hành mô phỏng này được thiết kế để có thể hoạt động được với nhiều processor. Giải thuật được sử dụng cho bộ lập lịch là **Multi-level Queue - MLQ**. Với giải thuật này, chúng ta đưa các tiến trình vào những hàng đợi gọi là **ready_queue** với độ ưu tiên tương ứng.

2.2 Trả lời các câu hỏi

Câu hỏi: Ưu điểm của chiến lược lập lịch được dùng trong bài tập lớn này so với các giải thuật đã được học là gì?

Trả lời: Scheduler Multilevel Queue này cho phép ưu tiên những process có độ ưu tiên cao được thực hiện trước, điều này giúp bảo đảm hệ thống đáp ứng được các yêu cầu quan trọng trong thời gian ngắn nhất. Mặt khác, các process có cùng độ ưu tiên thì được thực hiện theo phương pháp Round Robin, điều này giúp các process được thay phiên nhau thực hiện, tránh tình trạng starvation, đảm bảo tất cả các task đều có tiến triển.

So sánh với các giải thuật đã học: **Giải thuật FCFS:** MLQ nhờ có nhiều level priority khác nhau kết hợp cùng với giải thuật Round Robin giúp các process luân phiên được thực hiện, giảm thời gian chờ đối với những process có burst time ngắn. **Giải thuật SJF, SRTF:** giảm độ phức tạp, giảm tài nguyên và thời gian để tính toán Next CPU Burst mà vẫn đạt được hiệu quả.

2.3 Hiện thực

Giải thuật Chúng ta thực hiện cho dequeue các priority queue theo thứ tự ưu tiên của nó từ nhỏ đến lớn. Nếu các queue nào còn số timeslot thì mới được xuất process, còn không thì phải nhường cho process có độ ưu tiên thấp hơn.

Khi trong tất cả các queue, không còn queue nào có timeslot > 0, hoặc có timeslot > 0 nhưng queue rỗng, thì ta reset lại timeslot cho tất cả các queue.

Kết quả

Ở đây chúng em cho biến **MAX-PRIOR = 5**, để dễ dàng thấy sự thay đổi switching khi dùng giải thuật Round Robin giữa các process trong cùng một queue.

SƠ ĐỒ GANTT INPUT SCHED

Timeslot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
CPU 1														FIN			FIN	
CPU 0																		FIN

	Time Arrival	Priority
Process 1:	0	1
Process 2:	1	0
Process 3:	2	0

SƠ ĐỒ GANTT INPUT SCHED_0

Timeslot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
CPU 0																						FIN		FIN

	Time Arrival	Priority
Process 1:	0	1
Process 2:	4	2

SƠ ĐỒ GANTT INPUT SCHED_1

Timeslot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
CPU 0																					

	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
CPU 0		FIN																	FIN	FIN

	41	42	43	44	45
CPU 0					FIN

	Time Arrival	Priority
Process 1:	0	3
Process 2:	4	0
Process 3:	6	0
Process 4:	7	0

Output xuất ra màn hình

input sched

```
chip@DESKTOP-LPEVKGH:~/OS-assignment-232$ ./os sched
Time slot   0
ld_routine
        Loaded a process at input/proc/p1s, PID: 1 PRIO: 1
        CPU 1: Dispatched process  1
        Loaded a process at input/proc/p2s, PID: 2 PRIO: 0
Time slot   1
        Loaded a process at input/proc/p3s, PID: 3 PRIO: 0
Time slot   2
        CPU 0: Dispatched process  2
Time slot   3
        CPU 1: Put process   1 to run queue
        CPU 1: Dispatched process  3
Time slot   4
Time slot   5
Time slot   6
        CPU 0: Put process   2 to run queue
        CPU 0: Dispatched process  2
Time slot   7
        CPU 1: Put process   3 to run queue
        CPU 1: Dispatched process  3
Time slot   8
Time slot   9
Time slot  10
        CPU 0: Put process   2 to run queue
        CPU 0: Dispatched process  2
Time slot  11
        CPU 1: Put process   3 to run queue
        CPU 1: Dispatched process  1
Time slot  12
Time slot  13
Time slot  14
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  3
Time slot  15
        CPU 1: Put process   1 to run queue
        CPU 1: Dispatched process  1
Time slot  16
Time slot  17
        CPU 0: Processed  3 has finished
        CPU 0 stopped
Time slot  18
        CPU 1: Processed  1 has finished
        CPU 1 stopped
```

input sched-0

```
chip@DESKTOP-LPEVKGH:~/OS-assignment-232$ ./os sched_0
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
        Loaded a process at input/proc/s1, PID: 2 PRIO: 2
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  2
Time slot  10
Time slot  11
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot  12
Time slot  13
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot  14
Time slot  15
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  1
Time slot  16
Time slot  17
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  18
Time slot  19
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
```

```
Time slot  19
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  20
Time slot  21
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  22
        CPU 0: Processed  1 has finished
        CPU 0: Dispatched process  2
Time slot  23
        CPU 0: Processed  2 has finished
        CPU 0 stopped
```

input sched-1

```
chip@DESKTOP-LPEVKGH:~/OS-assignment-232$ ./os sched_1
Time slot   0
ld_routine
        Loaded a process at input/proc/s0, PID: 1 PRIO: 3
Time slot   1
        CPU 0: Dispatched process  1
Time slot   2
Time slot   3
Time slot   4
        Loaded a process at input/proc/s1, PID: 2 PRIO: 0
Time slot   5
Time slot   6
        Loaded a process at input/proc/s2, PID: 3 PRIO: 0
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
Time slot   7
        Loaded a process at input/proc/s3, PID: 4 PRIO: 0
Time slot   8
Time slot   9
Time slot  10
Time slot  11
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  3
Time slot  12
Time slot  13
Time slot  14
Time slot  15
Time slot  16
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
Time slot  17
Time slot  18
Time slot  19
Time slot  20
```

```
Time slot  21
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  2
Time slot  22
Time slot  23
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  3
Time slot  24
Time slot  25
Time slot  26
Time slot  27
Time slot  28
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  1
Time slot  29
Time slot  30
Time slot  31
Time slot  32
Time slot  33
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  4
Time slot  34
Time slot  35
Time slot  36
Time slot  37
Time slot  38
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  3
Time slot  39
Time slot  40
        CPU 0: Processed  3 has finished
        CPU 0: Dispatched process  4
Time slot  41
        CPU 0: Processed  4 has finished
        CPU 0: Dispatched process  1
Time slot  42
Time slot  43
Time slot  44
Time slot  45
Time slot  46
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Chương 3: QUẢN LÝ BỘ NHỚ - MEMORY MANAGEMENT

3.1 Lý thuyết

3.1.1 Phân trang (Paging)

Phân trang (Paging) là một trong những phương pháp phổ biến được các hệ điều hành dùng để quản lý bộ nhớ. Ý tưởng của nó là chia bộ nhớ thành các vùng nhớ có kích thước bằng nhau, được gọi là các trang (page) trên bộ nhớ ảo và các khung (frame) trên bộ nhớ vật lý. Các trang sẽ được ánh xạ tới các khung dựa trên bảng phân trang (paging table). Đối với các tiến trình, vùng nhớ ảo chúng sử dụng sẽ là liên tục, trong khi trên thực tế các khung tương ứng với các trang đó trên bộ nhớ vật lý là không liên tục. Các ưu điểm của phương pháp phân trang sẽ được để cập ở phần trả lời câu hỏi.

3.1.2 Swapping

Để tăng hiệu suất sử dụng bộ nhớ, hệ điều hành có thể ghi các trang hiện đang không được sử dụng ra một thiết bị nhớ sao lưu (ví dụ như ổ cứng SSD, HDD…) Khi các vùng nhớ này cần được truy cập, nó sẽ được nạp trở lại bộ nhớ chính (RAM). Quá trình này được gọi là swapping (hoán đổi). Quá trình này giúp tối ưu không gian bộ nhớ trống cho các tiến trình.

3.1.3 Translation Lookaside Buffer (TLB)

3.1.2.1 Khái niệm

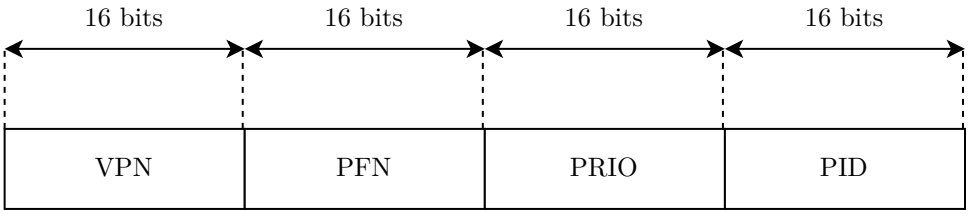
Translation Lookaside Buffer là một bộ nhớ vật lý có khả năng truy cập với tốc độ cao, được sử dụng như một bộ nhớ đệm (cache) dùng để truy xuất frame dựa vào page được cung cấp (chức năng giống như pages talbe).

3.1.2.2 Cấu trúc TLB trong bài tập lớn

Đối với BTL này, nhóm sẽ mô phỏng lại cách hoạt động của một TLB thật sự (TLB là một bộ nhớ phần cứng nhưng ta sẽ mô phỏng lại nó bằng chương trình C).

Theo thiết kế của nhóm, TLB sẽ có tối đa 512 entry, mỗi entry là một số 64 bits được phân vùng như

sau: Trong đó:



Hình 3.1: Cấu trúc TLB entry

1. **VPN**: Virtual page number là số hiệu của page trong process.
2. **PFN**: Physical frame number là số hiệu của frame trong bộ nhớ vật lý.
3. **PRIO**: Là số thể hiện độ ưu tiên trong thuật toán LRU, PRIO càng lớn cho thấy page đó được truy xuất càng gần thời điểm hiện tại.
4. **PID**: Process ID là ID của process sở hữu page, PID luôn khác 0, nếu PID có giá trị 0 nghĩa là entry rỗng.

3.1.2.3 Chính sách lưu trữ của TLB

TLB được thiết kế là một bộ nhớ cache lưu trữ theo cơ chế LRU, các page được truy cập gần nhất sẽ có giá trị PRIO cao nhất, khi có sự tương tác trong TLB (hit hoặc miss), thay đổi đối với page của process thì entry chứa pid và page number tương ứng sẽ được cập nhật lại frame number của entry đó và đặt PRIO ở giá trị cao nhất.

3.1.2.4 Chính sách truy xuất của TLB

Khi CPU tạo ra một địa chỉ ảo, nó kiểm tra TLB:

1. Nếu một bản ghi bảng trang có mặt (một hit TLB), số frame tương ứng được truy xuất.
2. Nếu một bản ghi bảng trang không được tìm thấy (một miss TLB), số trang 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 bản ghi trang mới.
3. Hit TLB:
   - CPU tạo ra một địa chỉ ảo.
   - TLB được kiểm tra (bản ghi có mặt).
   - Số frame tương ứng được truy xuất.
4. Miss TLB:
   - CPU tạo ra một địa chỉ ảo.
   - TLB được kiểm tra (bản ghi không có mặt).
   - Số trang được khớp với bảng trang trong bộ nhớ chính.
   - Số frame tương ứng được truy xuất.

3.2 Trả lời các câu hỏi

Câu hỏi: Trong hệ điều hành mô phỏng này, chúng ta hiện thực thiết kế nhiều phân đoạn bộ nhớ (memory segment). Ưu điểm của thiết kế nói trên là gì?

Trả lời: Thiết kế nhiều phân đoạn bộ nhớ trong mã nguồn mang lại nhiều lợi ích quan trọng:

- Tăng cường hiệu suất: Bằng cách phân chia bộ nhớ thành các đoạn có thể được cấp phát cho từng mục đích cụ thể riêng như lưu trữ chương trình hay bộ nhớ đệm..., hệ điều hành có thể phân bổ tài nguyên bộ nhớ một cách linh hoạt và hiệu quả hơn. Điều này giúp tối ưu hóa việc sử dụng bộ nhớ và tránh tình trạng lãng phí tài nguyên.
- Quản lý bộ nhớ linh hoạt: Thiết kế nhiều đoạn bộ nhớ cho phép hệ điều hành dễ dàng quản lý và theo dõi việc sử dụng bộ nhớ cũng như nhanh chóng phát hiện các vấn đề như rò rỉ bộ nhớ.
- Tăng cường bảo mật: Các phần khác nhau của hệ thống có thể được độc lập với nhau, ngăn chặn các lỗ hổng bảo mật trong một phần của hệ thống từ việc ảnh hưởng đến các phần khác.
- Hỗ trợ đa nhiệm: Mỗi tiến trình hoặc luồng thực thi có thể được gán một vùng bộ nhớ riêng biệt, giúp chúng hoạt động độc lập với nhau mà không ảnh hưởng đến bộ nhớ của nhau.

Câu hỏi: Điều gì sẽ xảy ra nếu ta chia địa chỉ nhiều hơn 2 mức trong hệ thống quản lý bộ nhớ theo cơ chế phân trang?

Trả lời: Trong hệ thống quản lý bộ nhớ theo cơ chế phân trang, việc ta chia nhiều tầng có thể mang lại một số ưu điểm cũng như hạn chế:

Ưu điểm:

- Giảm kích thước của các bảng phân trang

Hạn chế:

- Thời gian lookup sẽ tăng lên vì phải duyệt qua nhiều tầng
- Độ phức tạp trong việc quản lý cũng sẽ tăng lên

Câu hỏi: Ưu điểm và nhược điểm của phân đoạn kết hợp với phân trang (segmentation with paging) là gì?

Trả lời: Segmentation with paging (phân đoạn kết hợp với phân trang) có những ưu điểm và nhược điểm sau trong việc quản lý bộ nhớ trong hệ điều hành:

- Ưu điểm:

- Quản lý bộ nhớ linh hoạt: Segmentation with paging cho phép quản lý bộ nhớ linh hoạt bằng cách chia không gian địa chỉ logic thành các đoạn, sau đó chia nhỏ các đoạn đó thành các trang. Sự linh hoạt này cho phép việc phân bổ và quản lý tài nguyên bộ nhớ hiệu quả cho các tiến trình khác nhau.

- Bảo vệ và tách biệt: Segmentation cung cấp một mức độ bảo vệ và tách biệt giữa các đoạn khác nhau, ngăn chặn việc truy cập hoặc sửa đổi dữ liệu ngoài quyền hạn. Trong khi đó, Paging củng cố điều này bằng cách tăng cường bảo vệ tại cấp độ trang, cho phép kiểm soát chi tiết hơn về quyền truy cập bộ nhớ.

- Hỗ trợ cho không gian địa chỉ lớn: Segmentation xử lý việc chia nhỏ logic của bộ nhớ thành các đơn vị quản lý được, trong khi Paging xử lý việc ánh xạ vật lý của những đơn vị này tới các vị trí bộ nhớ thực tế, cho phép sử dụng hiệu quả cả bộ nhớ ảo và bộ nhớ vật lý.

- Nhược điểm:

- Phức tạp: Việc triển khai Segmentation with Paging làm tăng sự phức tạp cho hệ thống quản lý bộ nhớ của hệ điều hành. Điều này đòi hỏi sự phối hợp giữa các cơ chế Segmentation và Paging, đồng nghĩa với việc có thể làm tăng sự phức tạp của các thuật toán quản lý bộ nhớ và các cấu trúc dữ liệu.

- Phân mảnh: Segmentation có thể dẫn đến tình trạng phân mảnh nội và ngoại. Phân mảnh nội xảy ra khi bộ nhớ được phân bổ trong một đoạn lớn hơn cần thiết, làm lãng phí không gian. Phân mảnh ngoại xảy ra khi có đủ không gian bộ nhớ tồn tại, nhưng chúng không liên tục, làm khó khăn trong việc phân bổ các đoạn bộ nhớ lớn hơn.

- Tiêu tốn tài nguyên: Segmentation with Paging có thể tạo ra chi phí hoạt động bổ sung liên quan đến việc truy cập bộ nhớ và các hoạt động quản lý. Paging bao gồm việc duy trì bảng trang để ánh xạ địa chỉ logic sang địa chỉ vật lý, yêu cầu thêm bộ nhớ và tài nguyên xử lý. Ngoài ra, chuyển đổi giữa các đoạn và trang có thể tạo ra chi phí hoạt động bổ sung do việc chuyển đổi ngữ cảnh và các hoạt động ánh xạ bộ nhớ.

Câu hỏi: Điều gì sẽ xảy ra nếu hệ thống đa nhân với mỗi nhân CPU có thể chạy trên content khác nhau, mỗi nhân có đơn vị quản lý bộ nhớ riêng và một phần của nhân (TLB)? TLB 2 tầng được sử dụng phổ biến trên các CPU hiện đại, tác động của những cấu hình bộ nhớ tới các phương pháp dịch mã là gì?

Trả lời: Nếu mỗi nhân CPU trong hệ thống đa nhân có thể chạy trong các ngữ cảnh khác nhau và mỗi nhân có một Đơn vị Quản lý Bộ nhớ (MMU) riêng và một phần của lõi (TLB) của riêng nó, điều này ngụ ý rằng mỗi nhân có cơ chế dịch mã bộ nhớ và cấu trúc caching riêng. Dưới đây là cách cấu hình này và việc giới thiệu TLB 2 tầng trong CPU hiện đại ảnh hưởng đến các phương pháp dịch mã bộ nhớ của chúng ta:

- Tách biệt và Song song hóa: Với mỗi nhân có MMU và TLB riêng, các hoạt động dịch mã bộ nhớ và caching có thể được thực hiện độc lập trên mỗi nhân. Điều này khiến các nhân tách biệt hơn và cho phép thực thi song song các nhiệm vụ mà không cần tranh chấp về tài nguyên dịch mã bộ nhớ.

- Giảm chi phí nhất quán: Các chi phí nhất quán xảy ra khi nhiều nhân truy cập vào dữ liệu được chia sẻ và có thể giảm đi. TLB của mỗi nhân chứa các ánh xạ cụ thể cho ngữ cảnh thực thi riêng của nó, giảm sự cần thiết phải xóa TLB thường xuyên hoặc cơ chế thông điệp nhất quán để duy trì tính nhất quán giữa các TLB.

- Tăng hiệu suất: Việc sử dụng TLB 2 tầng trong CPU hiện đại cải thiện hiệu suất bằng cách giảm tỷ lệ TLB miss. Với khả năng chứa nhiều ánh xạ hơn và cấu trúc phân tầng, TLB 2 tầng có thể lưu trữ nhiều mã dịch hơn và cung cấp truy cập nhanh hơn vào các mã dịch được truy cập thường xuyên, giảm độ trễ của việc truy cập bộ nhớ.

- Phức tạp trong quản lý bộ nhớ: Tuy nhiên, quản lý dịch mã bộ nhớ trở nên phức tạp hơn do sự hiện diện của nhiều MMU và TLB. Hệ điều hành cần phối hợp dễ dàng các cập nhật dịch mã và tính nhất quán giữa các TLB để đảm bảo tính nhất quán trong các ánh xạ bộ nhớ trên tất cả các nhân.
- Tối ưu hóa phần mềm: Nhà phát triển phần mềm có thể cần tối ưu hóa mẫu truy cập bộ nhớ để tận dụng tối đa các lợi ích của hệ thống đa nhân với MMU và TLB riêng từng nhân. Điều này có thể bao gồm các chiến lược như sự kết hợp luồng, nơi các luồng được gắn với các nhân cụ thể để tận dụng các TLB cục bộ một cách hiệu quả.

3.3 Hiện thực

3.3.1 Cấu trúc dữ liệu tlbmemphy_struct

Để mô phỏng lại hoạt động của TLB bằng ngôn ngữ C, nhóm xây dựng một cấu trúc dữ liệu tlbmemphy_struct như sau:

```
struct tlbmemphy_struct
{
  /* Basic field of data and size */
  BYTE *storage;
  int maxsz;
  uint32_t num_of_entry;
  unsigned long long tlb_max_prio;
  unsigned long long tlb_min_prio;
  uint32_t tlb_min_prio_idx;
  uint32_t tlb_max_prio_idx;
};
```

Trong đó:

- **storage** là bộ nhớ chính của TLB, là bộ nhớ thô được tổ chức thành từng byte.
- **maxsz** là kích thước (tính bằng byte) của bộ nhớ TLB.
- **num_of_entry** là tổng số entry của TLB (bao gồm cả những entry trống).
- **tlb_max_prio** và **tlb_max_prio_idx** là giá trị độ ưu LRU lớn nhất của các entry và chỉ số của entry đó.
- **tlb_min_prio** và **tlb_min_prio_idx** là giá trị độ ưu LRU nhỏ nhất của các entry và chỉ số của entry đó.

3.3.2 Các chỉ thị tiền xử lí liên quan đến TLB

Như đã biết đối với TLB trong nội dung bài tập lớn này, chúng ta sẽ tương tác với từng entry có độ dài là 64 bits tức 8 bytes thay vì từng byte, do đó ta cần các chỉ thị tiền xử lí để tương tác với các entry

này tốt hơn. Lưu ý rằng các entry được xem như một số **unsigned long long**.

```
/* TLB definition */
#define TLB_ENTRYSZ 64 /* 64 bits */
/* Virtual page number (VPN) */
#define TLB_VPN_HBIT 63
#define TLB_VPN_LBIT 48
/* Physical frame number (PFN) */
#define TLB_PFN_HBIT 47
#define TLB_PFN_LBIT 32
/* Priority value */
#define TLB_PRIO_HBIT 31
#define TLB_PRIO_LBIT 16
/* Process ID (PID) */
#define TLB_PID_HBIT 15
#define TLB_PID_LBIT 0

/* TLB Masks */
#define TLB_VPN_MASK GENMASK_ULL (TLB_VPN_HBIT, TLB_VPN_LBIT)
#define TLB_PFN_MASK GENMASK_ULL (TLB_PFN_HBIT, TLB_PFN_LBIT)
#define TLB_PRIO_MASK GENMASK_ULL (TLB_PRIO_HBIT, TLB_PRIO_LBIT)
#define TLB_PID_MASK GENMASK_ULL (TLB_PID_HBIT, TLB_PID_LBIT)
```

Ngoài ra còn các chỉ thị **TLB_GET_XXX** và **TLB_SET_XXX** (**XXX** có thể là **VPN, PFN, PID, PRIO hoặc ENTRY**) dùng để trích xuất dữ liệu cũng như gán dữ liệu cho từng entry.

3.3.3 Các hàm tlb_cache_xxx

Theo thiết kế của nhóm, các hàm **tlb_cache_read, tlb_cache_write** và **tlb_cache_flush** sẽ là các hàm tương tác chính với bộ nhớ của TLB, các khai báo như sau:

```
int tlb_cache_read(struct tlbmemphy_struct *mp,
                   unsigned int pid, unsigned int pgnum,
                   unsigned long long *value);
int tlb_cache_write(struct tlbmemphy_struct *mp,
                    unsigned int pid, unsigned int pgnum,
                    unsigned long long value);
int tlb_cache_flush(struct tlbmemphy_struct *mp,
                    unsigned int pid, unsigned int pgnum);
```

Trong đó giải thuật của từng hàm như sau:

1. **tlb_cache_read**:
    - Kiểm tra **pid** có khác 0 hay không, nếu có trả về 1 và kết thúc hàm.

- Duyệt qua các entry của TLB, kiểm tra **PID** của từng entry, nếu nó trùng với **pid** thì chuyển sang bước tiếp theo.
- Tiếp tục lấy **VPN** của entry, so sánh với **pgnum**, nếu trùng thì ta lấy **PFN** của entry đó và gán vào value. Tiếp đó ta tiến hành gán **PRIO** của enty thành **tlb_max_prio + 1** và cập nhận cả **tlb_max_prio_idx** thành chỉ số của entry hiện tại. Nếu sau khi cập nhật mà **tlb_max_prio** có giá trị bằng với **UINT16_MAX** thì ta cần tiến hành phân bố lại các giá trị ưu tiên của entry bằng hàm **tlb_prio_reset** sẽ được trình bày ở phần tiếp theo.

2. **tlb_cache_write**:

- Kiểm tra **pid** có khác 0 hay không, nếu có trả về 1 và kết thúc hàm.
- Duyệt qua các entry của TLB, kiểm tra **PID** của từng entry, nếu nó trùng với **pid** thì chuyển sang bước tiếp theo.
- Tiếp tục lấy **VPN** của entry, so sánh với **pgnum**, nếu trùng thì ta gán value vòa **PFN** của entry. Tiếp đó ta tiến hành gán **PRIO** của enty thành **tlb_max_prio + 1** và cập nhận cả **tlb_max_prio_idx** thành chỉ số của entry hiện tại. Nếu sau khi cập nhật mà **tlb_max_prio** có giá trị bằng với **UINT16_MAX** thì ta cần tiến hành phân bố lại các giá trị ưu tiên của entry bằng hàm **tlb_prio_reset** sẽ được trình bày ở phần tiếp theo.

3. **tlb_cache_flush**:

- Kiểm tra **pid** có khác 0 hay không, nếu có trả về 1 và kết thúc hàm.
- Duyệt qua các entry của TLB, kiểm tra **PID** của từng entry, nếu nó trùng với **pid** thì chuyển sang bước tiếp theo.
- Tiếp tục lấy **VPN** của entry, so sánh với **pgnum**, nếu trùng thì ta dùng chỉ thị tiền xử lý **TLB_SET_ENTRY** và gán cả entry bằng giá trị 0 (xóa bỏ nội dung entry).

3.3.4 Các hàm tlbxxx

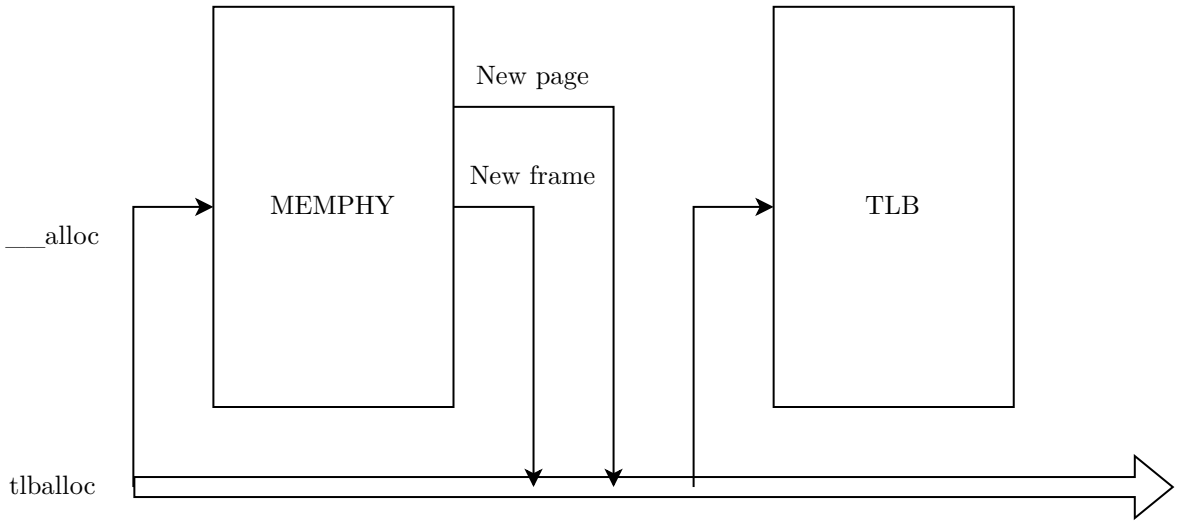
Các hàm **tlbxxx** (**xxx** là **alloc, free_data, read** hoặc **write**) luôn được gọi trước các tác vụ tương ứng (**___alloc, ___free, ___read** hoặc **___write**) nếu hệ điều hành hỗ trợ **CPU_TLB**. Các hàm này có vai trò tương tác chức năng của TLB với bộ nhớ vật lý chính của hệ điều hành.

Các hàm này có khai báo như sau:

```
int tlballoc (struct pcb_t *proc, unsigned int size,
              unsigned int reg_index);
int tlbfree_data (struct pcb_t *proc, unsigned int reg_index);
int tlbread (struct pcb_t *proc, unsigned int source,
             unsigned int offset, unsigned int *destination);
int tlbwrite (struct pcb_t *proc, BYTE data, unsigned int destination,
              unsigned int offset);
```

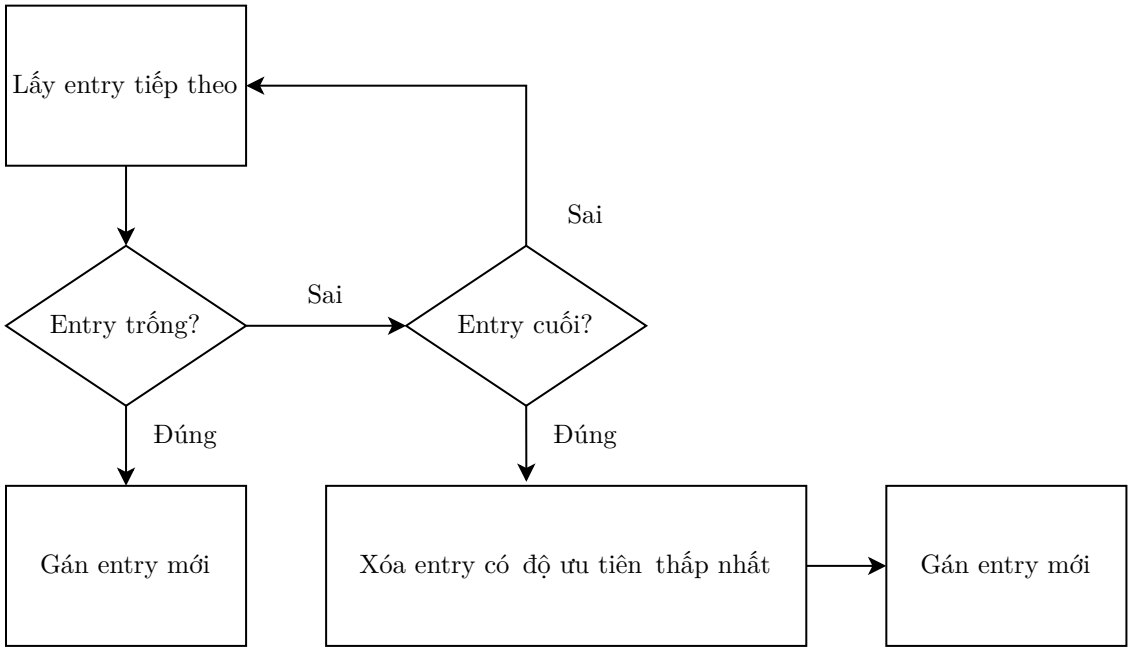
1. **tlballoc**

   Sơ đồ tương tác bộ nhớ vật lý (RAM và SWAP) với TLB.



Hình 3.2: Hàm tlballoc

Sau khi luồng xử lý đi vào TLB sẽ tiếp tục thực hiện các bước sau.



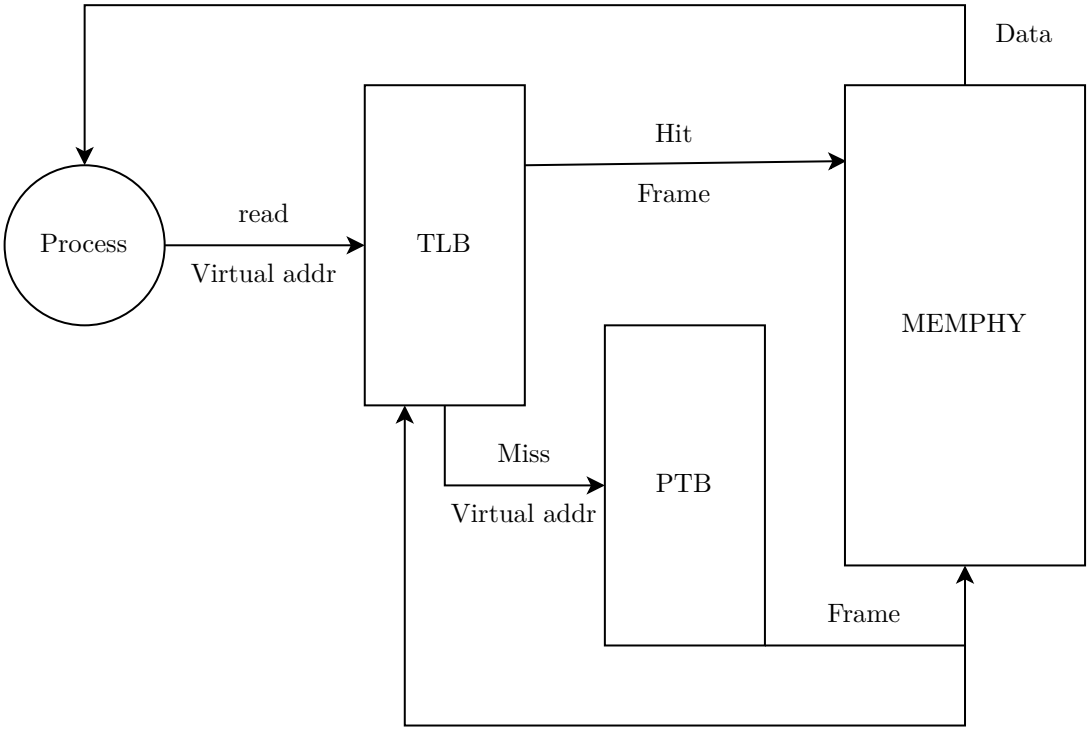
Hình 3.3: Hoạt động alloc của TLB

2. **tlbfree_data**

   Hàm nhận vào một giá trị **reg_index** để truy xuất trở lại page và frame tương ứng chúng ta cần giải phóng, sau đó ta tiến hành giải phóng bộ nhớ trong bộ nhớ vật lý. Cuối cùng ta xóa đi entry có page và frame tương ứng bằng hàm **tlb_cache_flush**.

3. **tlbread**

   Hàm thực hiện tác vụ đọc của process được thể hiện thông qua sơ đồ sau



Hình 3.4: Lệnh read thông qua TLB

4. **tlbwrite**

   Lệnh **tlbwrite** cũng thực hiện lệnh write của hệ thống tương tự như lệnh read.

3.3.5 Một số hàm khác

```
int tlb_prio_reset (struct tlbmemphy_struct *mp);
int tlb_get_min_prio (unsigned int *idx, unsigned long long *minPrio,
                      struct tlbmemphy_struct *mp);
int tlb_get_max_prio (unsigned int *idx, unsigned long long *maxPrio,
                      struct tlbmemphy_struct *mp);
```

1. **tlb__get__max__prio**: Duyệt qua tất cả entry và gán giá trị cho **tlb__max__prio** và **tlb__max__prio__idx** trong cấu trúc của TLB.

2. **tlb__get__min__prio**: Duyệt qua tất cả entry và gán giá trị cho **tlb__min__prio** và **tlb__min__prio__idx** trong cấu trúc của TLB.

3. **tlb__prio__reset**: Sử dụng khi số **tlb__max__prio** đạt tới giới hạn **UINT16__MAX** (Do trong quá trình tương tác ta luôn tăng giá trị này lên mà không căn chỉnh nhằm tránh phải duyệt qua toàn bộ entry mỗi lần tương tác với TLB). Hàm **tlb__prio__reset** sẽ duyệt qua tất cả entry theo thứ tự chỉ số từ nhỏ đến lớn, với mỗi **PRIO** có tồn tại (giá trị entry lấy được từ **TLB__GET__ENTRY** khác 0) ta sẽ đánh số lại **PRIO** của entry đó. Việc đánh số bắt đầu từ 1 và tăng dần, từ bỏ các thứ tự đã có trước đó.

Chương 4: KẾT HỢP CÁC THÀNH PHẦN - PUT IT ALL TOGETHER

4.1 Trả lời các câu hỏi

Câu hỏi: Điều gì sẽ xảy ra nếu hệ điều hành mô phỏng này không được hiện thực việc đồng bộ hóa? Mô phỏng vấn đề đó nếu có thể.

Nếu hệ điều hành mô phỏng này không hiện thực được việc đồng hóa, sẽ có nhiều vấn đề xảy ra, như các tình huống cạnh tranh, làm hỏng dữ liệu và tình trạng kẹt.

- Tình huống cạnh tranh: Hãy tưởng tượng một tình huống trong đó hai tiến trình trong hệ điều hành đang truy cập và sửa đổi một tài nguyên chia sẻ đồng thời mà không có đồng bộ hóa đúng cách. Ví dụ, xem xét hai tiến trình, A và B, cả hai đều tăng biến chia sẻ counter. Nếu không có đồng bộ hóa đúng cách, cả hai tiến trình có thể đọc giá trị hiện tại của counter đồng thời, tăng nó và sau đó ghi lại. Điều này có thể dẫn đến hành vi không mong muốn, chẳng hạn như lỗi cập nhật hoặc giá trị cuối cùng của counter không chính xác.
- Làm hỏng dữ liệu: Xem xét một tình huống trong đó nhiều tiến trình truy cập và sửa đổi một cấu trúc dữ liệu chia sẻ, chẳng hạn như một danh sách liên kết hoặc một tập tin, mà không có đồng bộ hóa đúng cách. Nếu một tiến trình đang trong quá trình cập nhật cấu trúc dữ liệu khi một tiến trình khác cố gắng đọc hoặc sửa đổi nó, cấu trúc dữ liệu có thể rơi vào trạng thái không nhất quán, dẫn đến làm hỏng dữ liệu.
- Tình trạng kẹt luồng thực thi: Các tình trạng kẹt xảy ra khi hai hoặc nhiều tiến trình đang chờ vô thời hạn để có tài nguyên được giữ bởi nhau, dẫn đến trạng thái mà không có tiến trình nào có thể tiếp tục. Nếu không có cơ chế đồng bộ hóa đúng cách như khóa mutex hoặc semaphore, tình trạng kẹt có thể xảy ra dễ dàng.

Chương 5: CÁC YÊU CẦU KHÁC

5.1 Quy cách mã nguồn (Coding style)

Nhóm sử dụng *git* làm công cụ quản lý mã nguồn giữa các thành viên.

Các file header và mã nguồn đều tuân theo format mã nguồn C quy định bởi GNU theo yêu cầu của đề bài. Để đảm bảo điều đó, nhóm sử dụng công cụ *clang-format*[1], phiên bản 14.0.0-1ubuntu1.1 để format mã nguồn mỗi khi commit.

File cấu hình cho *clang-format* là *.clang-format* được đặt tại project root và tuân theo GNU coding style, với duy nhất một option được chỉnh sửa là *SortIncludes: Never* để đảm bảo mã nguồn có thể được biên dịch với các file header cho sẵn. Luật *format* được thêm vào *Makefile* để format mã nguồn.

Bên cạnh đó, nhóm sử dụng công cụ *Code Spell Checker*[2] để kiểm tra chính tả cho mã nguồn. File cấu hình *.cspell.json* tại project root chứa các từ khóa được bỏ qua vì các từ khóa này được sử làm tên khai báo cho nhiều đối tượng trong mã nguồn.

5.2 Tuân thủ quy tắc đạo đức (Code of Ethics)

Nhóm tự hiện thực các giải thuật được đề bài yêu cầu dựa trên nền mã nguồn được các giáo viên phụ trách môn học cung cấp. Nếu có sai phạm, nhóm xin chịu trách nhiệm trước Khoa và Nhà trường.

Mã nguồn này được sử dụng cho mục đích nghiên cứu và học tập. Github repository của nhóm được dùng để các thành viên trong nhóm làm việc chung hiệu quả hơn, và sẽ chỉ để quyền riêng tư cho các thành viên nhóm xem và chỉnh sửa. Mã nguồn này sẽ không được mở công khai.

[1] Công cụ giúp format mã nguồn C, C++ và một số ngôn ngữ họ C.
[2] Công cụ kiểm tra chính tả nhiều ngôn ngữ khác nhau, được tích hợp trong nhiều IDE và text editor.

Chương 6: KẾT LUẬN

Thông qua quá trình tìm hiểu và hiện thực bài tập lớn trên, nhóm đã hiểu hơn về lý thuyết và cách hiện thực hai thành phần quan trọng nhất của một hệ điều hành là bộ lập lịch và bộ quản lý bộ nhớ. Nhóm cũng được tìm hiểu sâu về các cơ chế *Multi Level Queue* cho bộ lập lịch và *TLB paging* cho bộ quản lý bộ nhớ. Bên cạnh đó, nhóm còn được trau dồi các kỹ năng lập trình với các API hệ điều hành theo chuẩn POSIX, ngôn ngữ C và các kỹ năng làm việc nhóm.

Nhóm cũng xin dành lời cảm ơn cho đội ngũ giảng viên phụ trách môn học, đặc biệt là thầy Trần Văn Hoài phụ trách lớp lý thuyết và T.A. Huỳnh Ngọc Như phụ trách lớp thực hành và bài tập lớn đã mang đến những kiến thức bổ ích và sự hỗ trợ kịp thời, hữu ích để nhóm có thể hoàn thành môn học một cách tốt đẹp.