

VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING



OPERATING SYSTEMS

Report for Assignment

Simple Operating System

Advisor: Trần Trương Tuấn Phát
Students: Trần Đình Tường - 2213892
Lê Hoàng Khánh Vinh - 2213963
Nguyễn Minh Tú - 2213848
Trần Thành Tài - 2213001
Đoàn Viết Tường - 2213882

HO CHI MINH CITY, MAY 2024



Contents

1	Danh sách thành viên & Công việc	2
2	Giới Thiệu	3
2.1	Tổng quan	3
2.2	Source Code	5
2.3	Processes	7
2.4	How to Create a Process?	10
2.5	How to Run the Simulation	10
3	Scheduling	12
3.1	Questions	12
3.2	Implementation	14
3.2.1	Queue	14
3.2.2	Scheduler	15
3.3	Testcase and Gantt chart	18
4	Memory management	21
4.1	Questions	21
4.2	Translation Lookaside Buffer (TLB)	25
4.2.1	Thiết kế	25
4.2.2	Hiện thực	26
4.3	Hiện thực các Instruction	30
4.3.1	ALLOC	30
4.3.2	FREE	33
4.3.3	READ	35
4.3.4	WRITE	38
4.4	Testcase	41
5	Kết Luận	54
6	Tài liệu tham khảo	55



1 Danh sách thành viên & Công việc

No.	Họ và tên	MSSV	Công việc	%
1	Lê Hoàng Khánh Vinh	2213963	Code MMU và TLB Báo cáo Scheduling và MMU.	35%
2	Nguyễn Minh Tú	2213848	Chỉnh hình thức báo cáo	15%
3	Trần Thành Tài	2213001	Chỉnh hình thức báo cáo	15%
4	Đoàn Viết Tường	2213882	Chỉnh hình thức báo cáo	15%
5	Trần Đình Tường	2213892	Báo cáo phần Giới thiệu, Kết luận Code Scheduler, làm các questions	20%



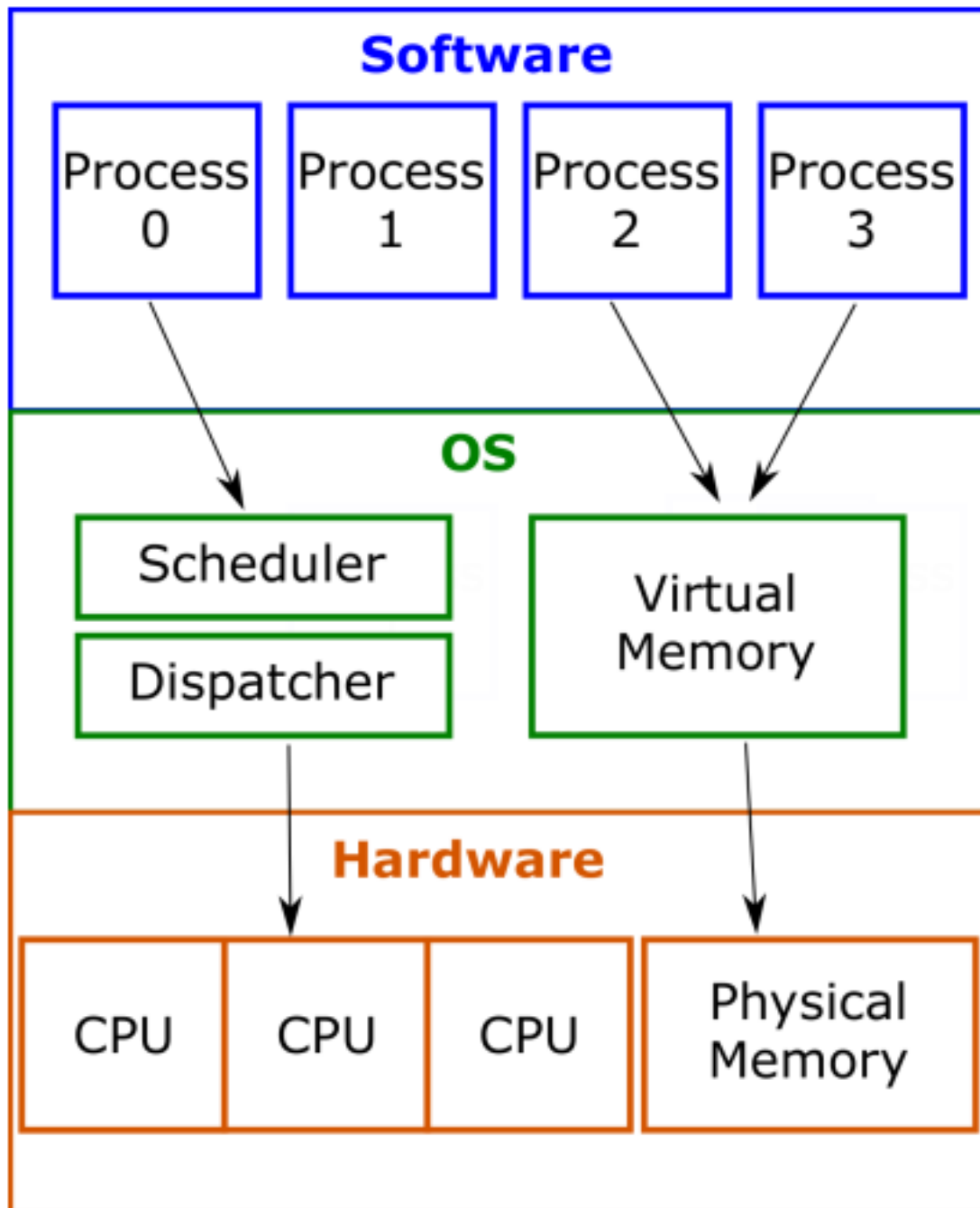
2 Giới Thiệu

2.1 Tổng quan

Bài tập này liên quan đến việc mô phỏng một hệ điều hành đơn giản để giúp sinh viên hiểu về các khái niệm cơ bản của lập lịch, đồng bộ hóa và quản lý bộ nhớ. Hình 1 mô tả kiến trúc tổng thể của hệ điều hành mà chúng ta sẽ triển khai. Nói 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 tiến trình nào được phép chạy trên CPU nào.
- Động cơ 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 sự tồn tại của các tiến trình khác. Điều này được thực hiện bằng cách cho mỗi tiến trình có không gian bộ nhớ ảo riêng của mình và Động cơ bộ nhớ ảo sẽ ánh xạ và dịch các địa chỉ ảo được cung cấp bởi các tiến trình thành các địa chỉ vật lý tương ứng.

Thực hiện thông qua các module này, hệ điều hành cho phép nhiều quy trình được tạo ra bởi người dùng chia sẻ và sử dụng các tài nguyên máy tính ảo. Do đó, trong bài tập này, chúng tôi tập trung vào việc triển khai bộ lập lịch/điều phối và động cơ bộ nhớ ảo.





2.2 Source Code

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

- **Header files**

- **timer.h**: Xác định bộ đếm thời gian cho toàn hệ thống
- **cpu.h**: Xác định các hàm dùng 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 chức năng được sử dụng bởi bộ lập lịch
- **mem.h**: Các chức năng được Virtual Memory Engine sử dụng
- **loader.h**: Các hàm được trình tải sử dụng để tải chương trình từ đĩa vào bộ nhớ
- **common.h**: Xác định các cấu trúc và hàm được sử dụng ở mọi nơi trong Hệ điều hành
- **bitopts.h**: Xác định các thao tác trên dữ liệu bit
- **os-mm.h, mm.h**: Xác định cấu trúc và dữ liệu cơ bản cho Quản lý bộ nhớ dựa trên phân trang
- **os-cfg.h**: (Tùy chọn) Xác định các hằng số sử dụng để chuyển đổi cấu hình phần mềm

- **Source files**

- **timer.c**: Triển khai bộ đếm thời gian
- **cpu.c**: Triển khai CPU ảo
- **queue.c**: Hiện thực các thao tác 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 file này
 - **loader.c**: Triển khai trình tải
 - **sched.c**: Triển khai bộ 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
-
- **Makefile**
 - **input** Mẫu đầu vào được sử dụng để xác minh
 - **output** Mẫu đầu ra của hệ điều hành

2.3 Processes

Chúng tôi sắp xây dựng một hệ điều hành đa nhiệm cho phép nhiều tiến trình chạy đồng thời, vì vậy rất đáng để dành chút thời gian để giải thích cách tổ chức các quy trình. Hệ điều hành quản lý các quy trình thông qua PCB của chúng được mô tả như sau:

```
1          // From include/common.h
2      struct pcb_t {
3          uint32_t pid;
4          uint32_t priority;
5          uint32_t code_seg_t * code;
6          addr_t regs[10];
7          uint32_t pc;
8          #ifdef MLQ_SCHED
9              uint32_t prio;
10         #endif
11         struct page_table_t * page_table; /* obsoleted
12             incompatible with MM_PAGING*/
13         uint32_t bp;
14     }
```

Dưới đây là mô tả ý nghĩa của các trường trong cấu trúc:

- **PID:** ID của quy trình (Process ID)
- **priority:** Ưu tiên của quy trình, giá trị càng thấp thì quy trình có ưu tiên càng cao. Ưu tiên này được xác định dựa trên các thuộc tính của quy trình và được cố định trong suốt phiên làm việc của quy trình
- **code:** Đoạn mã của quy trình (Text segment). Được sử dụng để lưu trữ mã máy của quy trình. Trong mô phỏng này, mã máy được đơn giản hóa và không được đặt trong RAM.



- **regs:** Các thanh ghi (Registers). Mỗi quy 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 (Program Counter). Biểu diễn vị trí của lệnh đang được thực thi trong mã máy của quy trình
- **page_table:** Bảng dịch từ địa chỉ ảo sang địa chỉ vật lý (obsolete, không sử dụng). Trong phiên bản hiện tại của mô phỏng, không sử dụng bảng trang
- **bp:** Con trỏ Break (Break pointer), được sử dụng để quản lý phân đoạn heap của tiến trình (Process ID)
- **prio:** Ưu tiên thực thi (nếu được hỗ trợ), và giá trị này ghi đè lên ưu tiên mặc định
- **CALC:** Thực hiện một số tính toán bằng CPU. Lệnh này không có đối số.

Chú thích về vùng nhớ: Một khu vực lưu trữ nơi chúng ta cấp phát không gian lưu trữ cho một biến, thuật ngữ này thực sự liên quan đến một chỉ số trong BẢNG KÝ HIỆU và thường hỗ trợ đọc được bằng con người thông qua tên biến và một cơ chế ánh xạ. Thật không may, cơ chế ánh xạ này nằm ngoài phạm vi của môn học Hệ điều hành này. Nó có thể thuộc một môn học khác mà giải thích cách trình biên dịch thực hiện công việc của mình và ánh xạ nhân đến chỉ số liên quan. Để đơn giản, chúng tôi chỉ đề cập đến một vùng nhớ thông qua chỉ số của nó và nó có một giới hạn về số lượng biến trong mỗi chương trình/tiến trình.

- **ALLOC:** Cấp phát một số phần của bộ nhớ (RAM). Cú pháp của lệnh:

`alloc [size] [reg]`

Trong đó, **size** là số byte mà quá trình muốn cấp phát từ RAM và **reg** là số của thanh ghi sẽ lưu địa chỉ của byte đầu tiên của khu vực bộ nhớ được cấp



phát. Ví dụ, lệnh **alloc 124 7** sẽ cấp phát 124 byte từ hệ điều hành và địa chỉ của byte đầu tiên trong số 124 byte đó sẽ được lưu tại thanh ghi số #7

- **FREE:** Cấp phát một số phần của bộ nhớ (RAM). Cú pháp của lệnh:

`free [reg]`

Trong đó, reg là số thanh ghi chứa địa chỉ bắt đầu của khu vực bộ nhớ đã được cấp phát và cần được giải phóng

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

`free [reg]`

Lệnh đọc một byte từ bộ nhớ tại địa chỉ bằng giá trị của thanh ghi **source + offset** và lưu nó vào **đích (destination)**. Ví dụ, giả sử giá trị của thanh ghi #1 là **0x123**, sau đó lệnh **read 1 20 2** sẽ đọc một byte từ bộ nhớ tại địa chỉ **0x123 + 14** (14 là 20 trong hệ số hexa) và lưu nó vào thanh ghi #2

- **WRITE:** Lệnh ghi một giá trị từ thanh ghi vào bộ nhớ. Cú pháp của lệnh:

`free [reg]`

Lệnh này ghi dữ liệu vào địa chỉ bằng giá trị của thanh ghi **đích(destination) + offset**. Ví dụ, giả sử giá trị của thanh ghi #1 là **0x123**, sau đó lệnh **write 10 1 20** sẽ ghi giá trị 10 vào bộ nhớ tại địa chỉ **0x123 + 14** (14 là 20 trong hệ hexa)

2.4 How to Create a Process?

Nội dung của mỗi tiến trình thực sự là một bản sao của một chương trình được lưu trữ trên đĩa. Vì vậy, để tạo ra một quy trình, chúng ta trước tiên phải tạo chương trình mô tả nội dung của nó. Một chương trình được xác định bởi một tệp duy nhất có định dạng sau:

```
[priority] [N = number of      instructions]
instruction 0
instruction 1
...
instruction N-1
```

Trong đó **priority** là ưu tiên **mặc định** của tiến trình được tạo từ chương trình này. Cần nhớ rằng hệ thống này sử dụng cơ chế ưu tiên kép. Tiến trình có ưu tiên cao hơn (với giá trị nhỏ hơn) sẽ có cơ hội cao hơn để được chọn bởi CPU từ hàng đợi. **N** là số lệnh và mỗi dòng tiếp theo của **N** dòng sau đó là các lệnh được biểu diễn trong định dạng đã đề cập trong phần trước. Bạn có thể mở các tệp trong thư mục **input/proc** để xem các chương trình mẫu.

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

2.5 How to Run the Simulation

Chúng ta sẽ thực hiện 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 tập tin mô tả trong thư mục đầu vào về phần cứng và môi trường mà chúng ta sẽ mô phỏng. Tập tin mô tả được định nghĩa theo định dạng sau:

```
[time slice] [N = Number of CPU] [M = Number of Processes to be run]
[time 0] [path 0] [priority 0]
[time 1] [path 1] [priority 1]
...
[time M-1] [path M-1] [priority M-1]
```

Trong tập tin mô tả, **slice** thời gian là thời gian (tính bằng giây) mà một tiến trình được phép chạy. **N** là số lượng CPU có sẵn và **M** là số lượng tiến trình cần chạy. Tham số cuối cùng là **priority**, đó là ưu tiên thực thi khi tiến trình được gọi và điều này sẽ ghi đè lên ưu tiên mặc định trong tập mô tả tiến trình (xem phần 2.4).

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

Đáng lưu ý rằng hệ thống này trang bị cơ chế ưu tiên kép. Nếu bạn không có ưu tiên mặc định, thì chúng tôi không có đủ tài liệu để giải quyết xung đột trong quá trình lập lịch. Tuy nhiên, nếu giá trị này được cố định, nó giới hạn các thuật toán mà mô phỏng có thể minh họa lý thuyết. Xác minh với môi trường thực tế của bạn, có các hệ thống ưu tiên khác nhau, một là về chương trình hệ thống so với chương trình người dùng trong khi hệ thống khác cũng cho phép bạn thay đổi ưu tiên thực thi.

Để bắt đầu mô phỏng, bạn phải biên dịch mã nguồn trước bằng cách sử dụng lệnh **Make all**. Sau đó, chạy lệnh:

```
./os [configure_file]
```

là **tập cấu hình**(**configure_file** là đường dẫn đến tập cấu hình cho môi trường bạn muốn chạy và nó nên được liên kết với tên của một tập mô tả được đặt trong thư mục đầu vào.



3 Scheduling

3.1 Questions

Question 1: What is the advantage of the scheduling strategy used in this assignment in comparison with other scheduling algorithms you have learned ?

First Come First Served (FCFS):

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

Shortest Job First (SJF):

- MLQ dễ hiện thực hơn vì SJF yêu cầu phải biết trước (thực tế là tiên đoán) độ dài thời gian sử dụng CPU của process điều đó rất khó.
- SJC các process có thời gian sử dụng CPU dài phải chờ đợi lâu hơn hoặc chờ đợi vô thời hạn khi nhiều process có thời gian sử dụng CPU ngắn đồng thời nhập vào hàng đợi -> starvation (đói bụng) .



Round Robin (RR):

- nếu quantum lớn : RR->FCFS.
- Nếu quantum bé : context switch của CPU sẽ tăng đáng kể gây ra chi phí cố hữu, giảm sử dụng CPU.

Priority Scheduling (PS):

- Các process có ưu tiên thấp có thể không có cơ hội thực thi -> starvation (đói bụng).

Ưu điểm của việc sử dụng các thuật toán **Multi-Level Queue (MLQ)**:

- **Response time**: Do các process có cùng ưu tiên được đặt trong cùng một hàng đợi, các tiến trình ưu tiên cao được xử lý trước.
- **Fairly**: Đảm bảo sự công bằng cho tất cả các process trong cùng một hàng đợi được thực thi bằng cách áp dụng Round Robin “style”.
- **Not starvation**: Mỗi hàng đợi chỉ có một số khe cố định để sử dụng CPU (MAX_PRIO - prio), và khi nó được sử dụng hết, hệ thống phải chuyển tài nguyên sang process khác trong hàng đợi tiếp theo.

3.2 Implementation

3.2.1 Queue

Trong bài tập lớn lần này chúng ta sẽ sử dụng Multi-level Queue để hiện thực hàng đợi cho các process. Hệ thống sẽ bao gồm 140 queue tương ứng với số mức độ ưu tiên từ 0 đến 139, độ ưu tiên bằng 0 chính là độ ưu tiên cao nhất. Ngoài ra trong mỗi queue cũng có một số lượng nhất định số lần queue đó được phép sử dụng CPU. Số slot trong một queue được quy định bằng MAX_PRIO - prio, với MAX_PRIO trong trường hợp này bằng 140 và prio là mức độ ưu tiên của queue. Ví dụ, queue có độ ưu tiên là 5 sẽ có slot là 135.

Cụ thể, cấu trúc của queue được biểu diễn trong struct queue_t như sau:

C Code

```
1 struct queue_t {
2     struct pcb_t * proc[MAX_QUEUE_SIZE];
3     int size;
4     int slot;
5     pthread_mutex_t queue_lock;
6 };
```

Ở trong file "queue.c", ta thực hiện các hàm enqueue() và dequeue() để đưa một process vào trong hàng đợi cũng như chọn process phù hợp nhất để lấy ra khỏi ready_queue.

- enqueue(): Thêm process vào cuối hàng đợi. Vì các process trong hàng đợi đều có mức độ ưu tiên giống nhau nên ta xác định thứ tự trong mỗi hàng đợi theo FIFO.
- dequeue(): Lấy process ở đầu hàng đợi và cập nhật lại ready_queue.



C Code

```
1 void enqueue(struct queue_t* q, struct pcb_t* proc) {
2     /* TODO: put a new process to queue [q] */
3     if (q == NULL || proc == NULL || q->size >=
4         MAX_QUEUE_SIZE) return;
5     q->proc[q->size++] = proc;
6 }
7 struct pcb_t* dequeue(struct queue_t* q) {
8     /* TODO: return a pcb whose priority is the highest
9      * in the queue [q] and remember to remove it from q
10     */
11     if (q == NULL || q->size == 0) return NULL;
12
13     struct pcb_t* res = q->proc[0];
14
15     for (int i = 0; i < q->size - 1; i++) {
16         q->proc[i] = q->proc[i + 1];
17     }
18     q->proc[q->size - 1] = NULL;
19     q->size--;
20     return res;
21 }
```

3.2.2 Scheduler

Đầu tiên, nhóm nghiên cứu sẽ chỉnh sửa một chút ở hàm `init_scheduler()` nhằm tạo `mutex_lock` và khởi tạo số lượng slot cho từng queue.

C Code

```
1 void init_scheduler(void) {
2 #ifdef MLQ_SCHED
3     int i ;
4     pthread_mutex_init(&mlq_ready_queue->queue_lock ,
5         NULL);
6     for (i = 0; i < MAX_PRIO; i ++) {
7         mlq_ready_queue[i].size = 0;
8         mlq_ready_queue[i].slot = MAX_PRIO - i;
9     }
10 #endif
11     ready_queue.size = 0;
12     run_queue.size = 0;
13 }
```

Nhóm nghiên cứu sau đó thực hiện hàm `get_mlq_proc()` để lấy process phù hợp ra khỏi hàng đợi.

- Bước 1: Chúng ta sẽ sử dụng một `mutex_lock` để đảm bảo sự đồng bộ của chương trình khi có nhiều process muốn truy cập vào vùng dữ liệu sử dụng chung.
- Bước 2: Ta sử dụng một biến `check` để kiểm tra liệu hàng đợi còn slot hay không. Nếu một hàng đợi không rỗng và còn slot thì ta sẽ lấy một process từ hàng đợi đó theo FIFO. Gán `check = 1` và thoát khỏi vòng lặp.
- Bước 3: Nếu mọi hàng đợi trong MLQ đều rỗng hoặc hết slot. Lúc này, biến `check` sẽ cho giá trị 0. Ta phải reset toàn bộ slot cho tất cả hàng đợi và thực hiện lại việc tìm process thích hợp trong hàng đợi.
- Bước 4: Mở khóa cho `mutex_lock` và trả về process nếu có.



C Code

```
1 struct pcb_t * get_mlq_proc(void) {
2     struct pcb_t * proc = NULL;
3     /*TODO: get a process from PRIORITY [ready_queue].
4     * Remember to use lock to protect the queue.
5     */
6     pthread_mutex_lock(&mlq_ready_queue->queue_lock);
7     int check = 0;
8     for (int i = 0; i < MAX_PRIO; i++) {
9         if (!empty(&mlq_ready_queue[i]) && mlq_ready_queue[
10             i].slot > 0) {
11             proc = dequeue(&mlq_ready_queue[i]);
12             mlq_ready_queue[i].slot--;
13             check = 1;
14             break;
15         }
16     }
17     if (check == 0) {
18         for (int i = 0; i < MAX_PRIO; i++) {
19             mlq_ready_queue[i].slot = MAX_PRIO - i;
20         }
21         for (int i = 0; i < MAX_PRIO; i++) {
22             if (!empty(&mlq_ready_queue[i]) &&
23                 mlq_ready_queue[i].slot > 0) {
24                 proc = dequeue(&mlq_ready_queue
25                     [i]);
26                 mlq_ready_queue[i].
27                     slot--;
28                 break;
29             }
30         }
31     }
32     pthread_mutex_unlock(&mlq_ready_queue->queue_lock);
33     return proc;
34 }
```

3.3 Testcase and Gantt chart

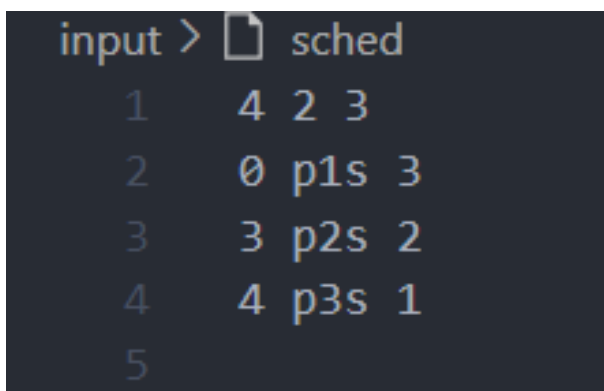
Để chạy được các testcase liên quan đến scheduler. Ta phải vào file "os-cfg.h" và comment các dòng #define như sau:

C Code

```
1 // #define CPU_TLB
2 // #define CPUTLB_FIXED_TLBSZ
3 // #define MM_PAGING 1
4 // #define MM_FIXED_MEMSZ 1
5 #define VMDBG 1
6 #define MMDBG 1
7 #define IODUMP 1
8 #define PAGETBL_DUMP 1
```

Sau đó mở Terminal và gõ lệnh make sched. Từ đó ta sẽ có thể bắt đầu test với những input liên quan đến scheduler.

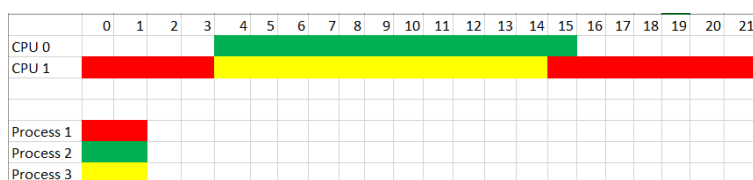
Ví dụ: Để sử dụng Multi-level queue ta cần thêm độ ưu tiên của process trong input như sau:



```
input > sched
1 4 2 3
2 0 p1s 3
3 3 p2s 2
4 4 p3s 1
5
```

Hình 1: *Input sched cho Scheduler.*

- Trong testcase này chúng ta sẽ có time slice cho thuật toán Round-Robin là 4.
Có 2 CPU và 3 Process.
- Process 1 có Arrival time là 0 và Priority là 3.
- Process 2 có Arrival time là 3 và Priority là 2.
- Process 3 có Arrival time là 4 và Priority là 1.



Hình 2: *Gantt chart cho input sched.*

Mặc dù đến đầu tiên nhưng vì có độ ưu tiên thấp hơn hai process còn lại nên sau khi hết time slice thì Process 1 phải đợi 2 process kia hoàn thành mới có thể tiếp tục thực hiện công việc.



```
Time slot 0
ld_routine
  Loaded a process at input/proc/p1s, PID: 1 PRIO: 3
  CPU 1: Dispatched process 1
Time slot 1
Time slot 2
  Loaded a process at input/proc/p2s, PID: 2 PRIO: 2
Time slot 3
Time slot 4
  CPU 0: Dispatched process 2
  Loaded a process at input/proc/p3s, PID: 3 PRIO: 1
  CPU 1: Put process 1 to run queue
  CPU 1: Dispatched process 3
Time slot 5
Time slot 6
Time slot 7
Time slot 8
  CPU 0: Put process 2 to run queue
  CPU 0: Dispatched process 2
  CPU 1: Put process 3 to run queue
  CPU 1: Dispatched process 3
Time slot 9
Time slot 10
Time slot 11
Time slot 12
  CPU 0: Put process 2 to run queue
  CPU 0: Dispatched process 2
  CPU 1: Put process 3 to run queue
  CPU 1: Dispatched process 3
Time slot 13
Time slot 14
Time slot 15
  CPU 1: Processed 3 has finished
  CPU 1: Dispatched process 1
```

Hình 3: *Output của input sched trên Terminal (1).*

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

Hình 4: *Output của input sched trên Terminal (2).*

4 Memory management

4.1 Questions

Question 1: In this simple OS, we implement a design of multiple memory segments or memory areas in source code declaration. What is the advantage of the proposed design of multiple segments?

Ưu điểm của thiết kế đề xuất về nhiều đoạn:

- **Tổ chức bộ nhớ :** Bằng cách chia bộ nhớ thành nhiều đoạn, bạn có thể có một cấu trúc bộ nhớ được tổ chức và có trật tự hơn. Mỗi đoạn có thể được dành riêng cho một mục đích cụ thể hoặc loại dữ liệu (data segment, code segment, stack segment, heap segment,...), làm cho việc quản lý và truy cập thông tin trở nên dễ dàng hơn.
- **Phân bổ Bộ nhớ Hiệu quả :** Bằng cách chia bộ nhớ thành các đoạn, bạn có thể tối ưu hóa phân bổ bộ nhớ. Các segments khác nhau có thể được phân bổ dựa trên yêu cầu cụ thể, chẳng hạn như stack segment để lưu trữ biến cục bộ và cuộc gọi hàm, heap segment cho phân bổ bộ nhớ động và đoạn mã cho các chỉ thị có thể thực thi. Việc phân đoạn này có thể cải thiện việc sử dụng bộ nhớ và giảm thiểu sự phân mảnh.
- **Context Switching :** Khi thực hiện Context Switching giữa các process hoặc thread, việc có bộ nhớ được phân đoạn có thể làm đơn giản process. Chỉ cần các đoạn cần thiết được trao đổi, thay vì toàn bộ không gian bộ nhớ, điều này có thể hiệu quả và nhanh chóng hơn.

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

- **Giảm Thiểu Chi Phí Bộ Nhớ:** Với nhiều cấp độ hơn, mỗi bảng phân trang trở nên nhỏ hơn, giảm thiểu chi phí bộ nhớ cần thiết để lưu trữ các bảng trang. Điều này đặc biệt có lợi trong các hệ thống có không gian địa chỉ lớn.



- **Cải Thiện Thời Gian Truy Cập Bảng Phân Trang:** Trong một kế hoạch phân trang nhiều cấp, bảng phân trang được chia thành nhiều thành bảng phân trang nhỏ hơn. Điều này giảm kích thước của mỗi bảng, làm cho việc truy cập và duyệt qua nhanh hơn trong quá trình dịch địa chỉ. Nó có thể giúp giảm thiểu sự gia tăng thời gian truy cập mà sẽ xảy ra với một lớn.
- **Phân Bỏ Linh Hoạt Bảng Trang:** Với nhiều cấp độ, việc phân bổ và quản lý bảng phân trang động trở nên dễ dàng hơn. Bạn có thể phân bổ các bảng trang theo yêu cầu của các tiến trình, giảm thiểu lãng phí bộ nhớ.
- **Sử Dụng Bộ Nhớ Hiệu Quả:** Một kế hoạch phân trang đa cấp cho phép sử dụng hiệu quả các tài nguyên bộ nhớ. Thay vì có một bảng trang lớn cần phải được phân bổ liên tục, bạn có thể phân bổ các bảng trang nhỏ hơn theo cách phân mảnh, sử dụng bộ nhớ có sẵn một cách hiệu quả hơn.

Question 3: What is the advantage and disadvantage of segmentation with paging?
Ưu điểm:

- Tối ưu hóa bộ nhớ, sử dụng bộ nhớ một cách hiệu quả.
- Phân bổ đơn giản của bộ nhớ không liên tục.
- kích thước trang không cố định lớn bằng cách phân trang trong từng đoạn
- Không xảy ra phân mảnh ngoại.

Nhược điểm:

- Vẫn có sự phân mảnh nội xảy ra.
- Kết hợp segmentation và paging tăng độ phức tạp của hệ thống quản lý bộ nhớ.

- Khi sử dụng segmentation với paging, mỗi đoạn có thể chứa nhiều trang. Do đó, số lượng entries trong các bảng trang có thể tăng, dẫn đến bảng phân trang lớn hơn.

Question 4: What will happen if the multi-core system has each CPU core can be run in a different context, and each core has its own MMU and its part of the core (the TLB)? In modern CPU, 2-level TLBs are common now, what is the impact of these new memory hardware configurations to our translation schemes?

Nếu mỗi core chạy trên một phần khác nhau MMU và TLB riêng thì:

Ưu điểm:

- **Cải thiện hiệu suất xử lý** : Mỗi nhân CPU có ngữ cảnh thực thi và cơ chế dịch trang riêng biệt, cho phép nhiều chương trình hoặc luồng thực thi đồng thời mà không ảnh hưởng lẫn nhau khi truy cập bộ nhớ. Điều này giúp cải thiện hiệu suất chung của hệ thống, đặc biệt đối với các khối lượng công việc có thể song song hóa.
- **Giảm thiểu lỗi memory** : MMU của mỗi nhân CPU cung cấp khả năng cách ly giữa các tiến trình chạy trên các nhân khác nhau. Nghĩa là chương trình chạy trên một nhân không thể truy cập bộ nhớ được phân bổ cho nhân khác.
- **Giảm thiểu lỗi TLB** : Việc có TLB riêng cho mỗi nhân giúp giảm thiểu khả năng nhiều nhân cạnh tranh cho cùng một chỗ.

Nhược điểm:

- **Độ phức tạp gia tăng** : Quản lý nhiều MMU và TLB làm tăng độ phức tạp cho thiết kế hệ thống. Và việc thiết kế cũng trở nên khó hơn.

Kiến trúc bộ nhớ hiện đại , TLB có 2 level tác động tới chương trình dịch bộ nhớ của chúng ta



- TLB 2 level làm tăng tỉ lệ hit đến làm giảm thời gian truy cập bộ nhớ , điều đó làm hiệu suất hệ thống tốt hơn.

Question 5: What will happen if the synchronization is not handled in your simple OS? Illustrate the problem of your simple OS by example if you have any?

Nếu đồng bộ hóa không được xử lý trong một hệ điều hành đơn giản, nó có thể dẫn đến nhiều vấn đề khác nhau, bao gồm race conditions, data inconsistency, and deadlock.

Hãy xem xét một ví dụ đơn giản để minh họa những 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 đồng thời mà không có cơ chế đồng bộ hóa nào được áp dụng:

Tiến trình A và Tiến trình B là hai tiến trình đồng thời mà tăng và giảm một biến chia sẻ là "count".

Cả hai tiến trình bắt đầu với "count" được khởi tạo là 0.

Tiến trình A thực hiện code sau: $\text{count} = \text{count} + 1$

Tiến trình B thực hiện code sau: $\text{count} = \text{count} - 1$

nếu không có đồng bộ hóa, sự kiện sau có thể xảy ra:

- Tiến trình A đọc giá trị hiện tại của "count" (0) vào một thanh ghi.
- Tiến trình B đọc giá trị hiện tại của "count" (0) vào một thanh ghi.
- Tiến trình A tăng giá trị trong thanh ghi cục bộ của nó lên 1 ($0 + 1 = 1$).
- Tiến trình B giảm giá trị trong thanh ghi cục bộ của nó đi 1 ($0 - 1 = -1$).
- Tiến trình A ghi giá trị đã cập nhật của "count" = 1.
- Tiến trình B ghi giá trị đã cập nhật của "count" = -1.

Trong tình huống này, cả hai tiến trình đều thực thi các hoạt động của mình đồng thời mà không có bất kỳ đồng bộ hóa nào. Kết quả là, giá trị cuối cùng của "count" là không chính xác -1 có thể -1 hoặc 0. Vấn đề này được biết đến là race

conditions, nơi kết quả của chương trình phụ thuộc vào thời gian và xen kẽ của các hoạt động đồng thời.

Ngoài ra, nếu thiếu các cơ chế đồng bộ hóa thích hợp, có nguy cơ xảy ra Deadlock. Deadlock có thể xảy ra khi hai hoặc nhiều tiến trình đang chờ đợi vô hạn để nhau giải phóng tài nguyên. Nếu có các tài nguyên chia sẻ mà các tiến trình cần truy cập vào, mà không có đồng bộ hóa thích hợp và có thể xảy ra một tình huống mà nhiều tiến trình đang chờ đợi một tài nguyên mà sẽ không bao giờ được giải phóng.

4.2 Translation Lookaside Buffer (TLB)

4.2.1 Thiết kế

Trong bài tập lớn lần này, nhóm nghiên cứu quyết định sẽ hiện thực một TLB fully associative với cơ chế thay các entry trong TLB bằng giải thuật LRU hiện thực bằng phương pháp Counter.

C Code

```
1 struct tlb_entry {
2     uint32_t pid;
3     int validBit;
4     int pgn;
5     int fpn;
6     int lruCount;
7 };
8
9 struct tlb_struct {
10     struct tlb_entry *tlb_entry;
11     int maxsz;
12 };
```

- Một TLB bao gồm một mảng các entry của nó và maxsz để cho biết kích thước của TLB.

- Mỗi entry sẽ có pid để biểu diễn id của process thuộc trang đó. Điều này là cần thiết vì TLB sẽ được dùng chung bởi toàn bộ process.
- Valid bit dùng để xác nhận rằng frame của entry đó có đang nằm trong main memory hay không. Valid bit bằng 1 thì frame đó đang nằm trong RAM, ngược lại frame đó có thể nằm trong SWAP hoặc không tồn tại.
- Các biến pgn và fpn để chứa Page Number và Frame Number tương ứng.
- lruCount là biến để thực hiện giải thuật LRU bằng Counter. Với những entry mới được truy cập gần đây, TLB entry sẽ có lruCount lớn và ngược lại lruCount sẽ bằng 0 đối với những entry đã lâu chưa được sử dụng. Có thể sẽ có nhiều entry có lruCount bằng 0.

4.2.2 Hiện thực

Đầu tiên nhóm nghiên cứu sẽ khởi tạo một TLB bằng hàm `init_tlbmemphy()`:

C Code

```
1 int init_tlbmemphy(struct tlb_struct *mp, int max_size)
2 {
3     mp->tlb_entry = (struct tlb_entry *)malloc(max_size *
4         sizeof(struct tlb_entry));
5     for (int i = 0; i < max_size; i++)
6     {
7         mp->tlb_entry[i].pid = -1;
8         mp->tlb_entry[i].validBit = 0;
9         mp->tlb_entry[i].pgn = -1;
10        mp->tlb_entry[i].fpn = -1;
11        mp->tlb_entry[i].lruCount = 0;
12    }
13    mp->maxsz = max_size;
14    return 0;
15 }
```

Tiếp theo, nhóm nghiên cứu sẽ hiện thực một số hàm liên quan đến việc cập nhật và tương tác với TLB:

- `tlb_updatelru()`: Hàm này sẽ nhận vào một index của TLB. Vì đang hiện thực TLB bằng phương pháp Counter, ta duyệt qua toàn bộ TLB, trừ `lruCount` đi 1 đối với những entry có `lruCount` lớn hơn `lruCount` của entry hiện tại. Sau đó, chúng ta cập nhật `lruCount` của entry hiện tại để thể hiện rằng entry này mới được truy cập gần đây. Cuối cùng, hàm trả về số lượng entry có `lruCount` lớn hơn entry hiện tại.

C Code

```
1 int tlb_updatelru(struct tlb_struct *tlb, int index)
2 {
3     int currentCount = tlb->tlb_entry[index].lruCount;
4     int count = 0;
5     for (int i = 0; i < tlb->maxsz; i++)
6     {
7         if (tlb->tlb_entry[i].lruCount > currentCount)
8         {
9             // Counter implementation of LRU requires
10              // all LRU counts greater than the one being
11              // accessed now to be decremented.
12              tlb->tlb_entry[i].lruCount--;
13              count++;
14          }
15      }
16      tlb->tlb_entry[index].lruCount = tlb->maxsz - 1;
17      return count;
18 }
```

- `tlb_search()`: Hàm này sẽ nhận vào một Page Number và trả về Frame Number tương ứng nếu như valid bit của entry chứa frame đó bằng 1.

C Code

```
1 int tlb_search(struct pcb_t *proc, struct tlb_struct *  
    tlb, int pgnum)  
2 {  
3     /* TODO search TLB cached*/  
4     for (int i = 0; i < tlb->maxsz; i++)  
5     {  
6         if (tlb->tlb_entry[i].pid == proc->pid && tlb->  
            tlb_entry[i].validBit && tlb->tlb_entry[i].  
            pgn == pgnum)  
7         {  
8             tlb_updatelru(tlb, i);  
9             return tlb->tlb_entry[i].fpn;  
10        }  
11    }  
12    return -1;  
13 }
```

- `tlb_getlru()`: Hàm này sẽ trả về entry đầu tiên có `lruCount` bằng 0. Vì chúng ta đang hiện thực LRU bằng Counter, sẽ luôn có ít nhất 1 entry có `lruCount` bằng 0.

C Code

```
1 int tlb_getlru(struct tlb_struct *tlb)
2 {
3
4     for (int i = 0; i < tlb->maxsz; i++)
5     {
6         if (tlb->tlb_entry[i].lruCount == 0)
7         {
8             return i;
9         }
10    }
11 }
```

- `tlb_get_invalid()`: Hàm này sẽ trả về vị trí đầu tiên có valid bit bằng 0 trong TLB.

C Code

```
1 int tlb_get_invalid(struct tlb_struct *tlb)
2 {
3     for (int i = 0; i < tlb->maxsz; i++)
4     {
5         if (tlb->tlb_entry[i].validBit == 0)
6         {
7             return i;
8         }
9     }
10    return -1;
11 }
```

- `tlb_update()`: Đầu tiên, hàm `tlb_update()` sẽ tìm vị trí đầu tiên có valid bit bằng 0 để cập nhật TLB. Nếu không thể tìm thấy, chúng ta sẽ tìm entry ít được sử dụng nhất, hay least recently used (LRU) để thực hiện thay thế.

C Code

```
1 int tlb_update(struct tlb_struct *tlb, int pid, int pgn,
2               int fpn)
3 {
4     int index = tlb_get_invalid(tlb);
5     if (index < 0) // If there is no invalid entry, we
6                   // replace with the lru entry
7     {
8         index = tlb_getlru(tlb);
9     }
10    tlb->tlb_entry[index].pid = pid;
11    tlb->tlb_entry[index].pgn = pgn;
12    tlb->tlb_entry[index].fpn = fpn;
13    tlb->tlb_entry[index].validBit = 1;
14    tlb_updatelru(tlb, index);
15 }
```

4.3 Hiện thực các Instruction

4.3.1 ALLOC

- `tlballoc()`: Ta gọi hàm `__alloc()` để cấp phát vùng nhớ cho process. Sau đó ta cập nhật `fpn` và `pgn` của địa chỉ được trả về trên TLB.



C Code

```
1 int tlballoc(struct pcb_t *proc, uint32_t size, uint32_t
    reg_index)
2 {
3     int addr, val;
4
5     /* By default using vmaid = 0 */
6     val = __alloc(proc, 0, reg_index, size, &addr);
7
8     /* TODO update TLB CACHED frame num of the new
        allocated page(s)*/
9     /* by using tlb_cache_read()/tlb_cache_write()*/
10
11     int pgn = PAGING_PGN(addr);
12     int pte = proc->mm->pgd[pgn];
13     int fpn = PAGING_PTE_FPN(pte);
14     tlb_update(proc->tlb, proc->pid, pgn, fpn);
15
16     return val;
17 }
```

- `__alloc()`: Trong hàm `__alloc()`, đầu tiên ta kiểm tra vùng không gian đã được cấp phát cho process có đủ kích thước không. Nếu không, ta sẽ cấp phát thêm vùng nhớ cho process đó. Sau khi đã có đủ kích thước cần thiết cho việc cấp phát, ta cập nhật lại `vm_area` của process cho phù hợp.



C Code

```
1 int __alloc(struct pcb_t *caller, int vmaid, int rgid,
2             int size, int *alloc_addr)
3 {
4     /*Allocate at the top of free area */
5     struct vm_rg_struct rgnode;
6     if (get_free_vmrg_area(caller, vmaid, size, &rgnode)
7         == 0) {
8         caller->mm->symrgtbl[rgid].rg_start = rgnode.
9             rg_start;
10        caller->mm->symrgtbl[rgid].rg_end = rgnode.
11            rg_end;
12        *alloc_addr = rgnode.rg_start;
13    }
14    else{
15        /*Attempt to increase limit to get space */
16        struct vm_area_struct *cur_vma = get_vma_by_num(
17            caller->mm, vmaid);
18        int gap_size = cur_vma->vm_end - cur_vma->sbrk;
19        if (gap_size < size){
20            int inc_sz = PAGING_PAGE_ALIGNSZ(size -
21                gap_size);
22            if (inc_vma_limit(caller, vmaid, inc_sz) <
23                0) return -1;
24        }
25        /*Successful increase limit */
26        caller->mm->symrgtbl[rgid].rg_start = cur_vma->
27            sbrk;
28        caller->mm->symrgtbl[rgid].rg_end = cur_vma->
29            sbrk + size;
30        *alloc_addr = caller->mm->symrgtbl[rgid].
31            rg_start;
32        cur_vma->sbrk += size;
33    }
34    return 0;
35 }
```



4.3.2 FREE

- `tlbfree_data()`: Hàm này chỉ đơn giản là gọi hàm `__free()` để thực hiện giải phóng bộ nhớ đã được cấp phát cho process khi process đã kết thúc.

C Code

```
1 int tlbfree_data(struct pcb_t *proc, uint32_t reg_index)
2 {
3     return __free(proc, 0, reg_index);
4 }
```

- `__free()`: Ta chuyển toàn bộ những TLB entry có liên quan đến process này thành invalid, sau đó đưa những vùng nhớ đã được cấp phát cho process này vào danh sách những vùng nhớ tự do.



C Code

```
1 int __free(struct pcb_t *caller, int vmaid, int rgid)
2 {
3     struct vm_rg_struct *rgnode;
4
5     if (rgid < 0 || rgid > PAGING_MAX_SYMTBL_SZ)
6         return -1;
7
8     /* TODO: Manage the collect freed region to
9        freerg_list */
10
11    rgnode = get_symrg_byid(caller->mm, rgid);
12
13    for (int i = 0; i < caller->tlb->maxsz; i++) {
14        if (caller->tlb->tlb_entry[i].pid == caller->pid
15            && caller->tlb->tlb_entry[i].pgn ==
16                PAGING_PGN(rgnode->rg_start)) {
17            caller->tlb->tlb_entry[i].validBit = 0;
18        }
19    }
20
21    /*enlist the obsoleted memory region */
22    enlist_vm_freerg_list(caller->mm, rgnode);
23
24    #ifdef VMDBG
25        printf("Free regions after free:\n");
26        print_list_rg(caller->mm->mmap->vm_freerg_list);
27    #endif
28
29    #ifdef MMDBG
30        printf("Using frames after free:");
31        print_list_fp(caller->mram->fifo_fp_list);
32    #endif
33
34    return 0;
35 }
```



4.3.3 READ

- `tlbread()`: Trong hàm này, đầu tiên ta kiểm tra trong TLB. Nếu địa chỉ cần tìm đang ở trong TLB, ta sẽ lấy được địa chỉ của nó trong main memory và thực hiện đọc dữ liệu tại đó. Nếu không có trong TLB, ta sẽ tiếp tục tìm trong Page Table.

C Code

```
1 int tlbread(struct pcb_t *proc, uint32_t source,
2             uint32_t offset, uint32_t destination)
3 {
4     BYTE data;
5     struct vm_rg_struct *cur_rg = get_symrg_byid(proc->
6         mm, source);
7     if (cur_rg == NULL)
8         return -1;
9     if (cur_rg->rg_start + offset >= cur_rg->rg_end)
10        return -1;
11    int pgn = PAGING_PGN((cur_rg->rg_start + offset));
12    int fpn = tlb_search(proc, proc->tlb, pgn);
13    int val;
14    if (fpn >= 0)
15    {
16        printf("TLB hit at read region=%d offset=%d\n",
17            source, offset);
18        int phyaddr = (fpn << PAGING_ADDR_FPN_LOBIT) +
19            offset;
20        MEMPHY_read(proc->mram, phyaddr, &data);
21        val = 0;
22    }
23    else
24    {
25        printf("TLB miss at read region=%d offset=%d\n",
26            source, offset);
27        val = __read(proc, 0, source, offset, &data);
28    }
29    proc->regs[destination] = (uint32_t)data;
30    return val;
31 }
```

- `__read()`: Ta lấy được địa chỉ cần đọc dữ liệu bằng `symrgtbl`, sau đó gọi hàm `pg_getval()` để lấy data.

C Code

```
1 int __read(struct pcb_t *caller, int vmaid, int rgid,
  int offset, BYTE *data)
2 {
3     struct vm_rg_struct *curr_rg = get_symrg_byid(caller->
  mm, rgid);
4
5     struct vm_area_struct *cur_vma = get_vma_by_num(
  caller->mm, vmaid);
6
7     if (curr_rg == NULL || cur_vma == NULL) /* Invalid
  memory identify */
8         return -1;
9
10    pg_getval(caller->mm, curr_rg->rg_start + offset, data
  , caller);
11
12    return 0;
13 }
```

- `pg_getval()`: Ta lấy được fpn thông qua pgn sau đó thực hiện việc đọc dữ liệu tại vị trí đã tìm được. Cập nhật lại TLB với entry bao gồm pgn và fpn mới được truy cập.

C Code

```
1 int pg_getval(struct mm_struct *mm, int addr, BYTE *data
  , struct pcb_t *caller)
2 {
3     int pgn = PAGING_PGN(addr);
4     int off = PAGING_OFFST(addr);
5     int fpn;
6
7     /* Get the page to MEMRAM, swap from MEMSWAP if
      needed */
8     if (pg_getpage(mm, pgn, &fpn, caller) != 0)
9         return -1; /* invalid page access */
10
11     int phyaddr = (fpn << PAGING_ADDR_FPN_LOBIT) + off;
12
13     MEMPHY_read(caller->mram, phyaddr, data);
14
15 #ifdef CPU_TLB
16     /* Update TLB */
17     tlb_update(caller->tlb, caller->pid, pgn, fpn);
18 #endif
19
20     return 0;
21 }
```

4.3.4 WRITE

- `tlbwrite()`: Đầu tiên, ta kiểm tra TLB, nếu ta tìm thấy `fpn` tương ứng. Ta thực hiện việc ghi dữ liệu với địa chỉ tương ứng trong RAM và cập nhật lại TLB. Nếu TLB Miss, ta bắt đầu thực hiện việc ghi dữ liệu bằng hàm `__write()` với việc kiểm tra Page Table.



C Code

```
1 int tlbwrite(struct pcb_t *proc, BYTE data,
2             uint32_t destination, uint32_t offset)
3 {
4     int val;
5     struct vm_rg_struct *cur_rg = get_symrg_byid(proc->
6         mm, destination);
7     if (cur_rg == NULL)
8         return -1;
9     if (cur_rg->rg_start + offset >= cur_rg->rg_end)
10    {
11        printf("Write out of bound\n");
12        return -1;
13    }
14    int pgn = PAGING_PGN((cur_rg->rg_start + offset));
15    int fpn = tlb_search(proc, proc->tlb, pgn);
16    if (fpn >= 0)
17    {
18        printf("TLB hit at write region=%d offset=%d
19            value=%d\n",
20                destination, offset, data);
21        int phyaddr = (fpn << PAGING_ADDR_FPN_LOBIT) +
22            offset;
23        MEMPHY_write(proc->mram, phyaddr, data);
24        val = 0;
25    }
26    else
27    {
28        printf("TLB miss at write region=%d offset=%d
29            value=%d\n",
30                destination, offset, data);
31        val = __write(proc, 0, destination, offset, data
32            );
33    }
34    return val;
35 }
```


- `__write()`: Ta lấy địa chỉ thông qua `symrgtbl` và gọi hàm `pg_setval()` để ghi dữ liệu vào địa chỉ đó.

C Code

```
1 int __write(struct pcb_t *caller, int vmaid, int rgid,
2             int offset, BYTE value)
3 {
4     struct vm_rg_struct *curr_g = get_symrg_byid(caller->
5           mm, rgid);
6
7     struct vm_area_struct *cur_vma = get_vma_by_num(
8           caller->mm, vmaid);
9
10    if (curr_g == NULL || cur_vma == NULL) /* Invalid
11          memory identify */
12        return -1;
13
14    int exceed = (curr_g->rg_start + offset >= curr_g->
15          rg_end);
16    if (exceed)
17    {
18        printf("Write value failed\n");
19        return -1;
20    }
21
22    int set_result = pg_setval(caller->mm, curr_g->
23          rg_start + offset, value, caller);
24    if (set_result == -1)
25    {
26        printf("Write value failed\n");
27        return -1;
28    }
29
30    return 0;
31 }
```

- `pg_setval()`: Tại đây, chúng ta có được `pgn`, `fpn`. Từ đó có được địa chỉ cần được ghi dữ liệu trong main memory và thực hiện ghi dữ liệu vào địa chỉ tương ứng. Cuối cùng cập nhật lại TLB.

C Code

```
1 int pg_setval(struct mm_struct *mm, int addr, BYTE value
  , struct pcb_t *caller)
2 {
3     int pgn = PAGING_PGN(addr);
4     int off = PAGING_OFFST(addr);
5     int fpn;
6
7     /* Get the page to MEMRAM, swap from MEMSWAP if
       needed */
8     if (pg_getpage(mm, pgn, &fpn, caller) != 0)
9         return -1; /* invalid page access */
10
11     int phyaddr = (fpn << PAGING_ADDR_FPN_LOBIT) + off;
12
13     MEMPHY_write(caller->mram, phyaddr, value);
14
15 #ifdef CPU_TLB
16     /* Update TLB */
17     tlb_update(caller->tlb, caller->pid, pgn, fpn);
18 #endif
19
20     return 0;
21 }
```

4.4 Testcase

Để kiểm tra chương trình mô phỏng hệ điều hành, nhóm nghiên cứu chọn một input đơn giản là "os_1_mlq_paging" để thực thi, trong file "os-cfg.h" nhóm nghiên cứu để size của TLB là một giá trị cụ thể và trong input lần này, cụ thể sẽ là 2 entry.



Vì đây là một input nhỏ nên nhóm nghiên cứu chỉ để kích thước tối đa của TLB là 2 entry, nhưng ta có thể thấy tỉ lệ hit vẫn khá cao với 3 lần hit và 1 lần miss. Sở dĩ tỉ lệ hit cao là do trong việc hiện thực hệ điều hành đơn giản này, chúng ta sử dụng một Multi-level queue và việc định thời cho các process dựa trên độ ưu tiên của process đó. Điều đó khiến cho các process có độ ưu tiên cao luôn được tập trung thực hiện trước nên các entry trong TLB sẽ chứa những frame và page có liên quan đến nhau và liên quan đến process đang thực hiện (hay còn gọi là locality of references). Việc hiện thực TLB theo cơ chế LRU và mapping theo fully associative cũng rất phù hợp với phương pháp định thời theo priority này, giúp tăng tỉ lệ hit của TLB và làm cho hệ điều hành thực hiện các công việc nhanh hơn.

Kết quả đem lại trên Terminal như sau:



```
Time slot 0
ld_routine
    Loaded a process at input/proc/p0s, PID: 1 PRIO: 130
    CPU 2: Dispatched process 1
Time slot 1
print_pgtbl: 0 - 512 [PID=1]
00000000: 80000000
00000004: 80000001
Free regions after alloc:
print_list_rg: NULL list
Using frames after alloc:
print_list_fp:
fp[1]
fp[0]

Time slot 2
    Loaded a process at input/proc/s3, PID: 2 PRIO: 39
    CPU 3: Dispatched process 2
Time slot 3
    CPU 2: Put process 1 to run queue
    CPU 2: Dispatched process 1
print_pgtbl: 0 - 768 [PID=1]
00000000: 80000000
00000004: 80000001
00000008: 80000002
Free regions after alloc:
print_list_rg: NULL list
Using frames after alloc:
print_list_fp:
fp[2]
fp[1]
fp[0]

    Loaded a process at input/proc/m1s, PID: 3 PRIO: 15
Freeing region: rg_start=0 rg_end=300
Free regions after free:
print_list_rg:
rg[0->300]
```

Hình 5: *Output os_1_mlq_paging.*



```
Loaded a process at input/proc/m1s, PID: 3 PRIO: 15
Freeing region: rg_start=0 rg_end=300
Free regions after free:
print_list_rg:
rg[0->300]

Using frames after free:print_list_fp:
fp[2]
fp[1]
fp[0]

CPU 1: Dispatched process 3
Time slot 4

print_pgtbl: 0 - 512 [PID=3]
00000000: 80000003
00000004: 80000004
Free regions after alloc:
print_list_rg: NULL list
Using frames after alloc:
print_list_fp:
fp[4]
fp[3]
fp[2]
fp[1]
fp[0]

CPU 3: Put process 2 to run queue
CPU 3: Dispatched process 2
CPU 2: Put process 1 to run queue
CPU 2: Dispatched process 1
Free regions after alloc:
print_list_rg:
rg[100->300]

Using frames after alloc:
print_list_fp:
fp[4]
fp[3]
fp[2]
```

Hình 6: *Output os_1_mlq_paging.*



```
Using frames after alloc:
print_list_fp:
fp[4]
fp[3]
fp[2]
fp[1]
fp[0]

Free regions after alloc:
print_list_rg: NULL list
Using frames after alloc:
print_list_fp:
fp[4]
fp[3]
fp[2]
fp[1]
fp[0]

Time slot 5
    Loaded a process at input/proc/s2, PID: 4 PRIO: 120
TLB hit at write region=1 offset=20 value=100
[PID=1] write region=1 offset=20 value=100
print_pgtbl: 0 - 768 [PID=1]
00000000: 80000000
00000004: 80000001
00000008: 80000002
Memory dump:
20: 100
    CPU 1: Put process 3 to run queue
    CPU 1: Dispatched process 3
Freeing region: rg_start=0 rg_end=300
Time slot 6
    CPU 0: Dispatched process 4
Free regions after free:
print_list_rg:
rg[0->300]

Using frames after free:print_list_fp:
fp[4]
```

Hình 7: *Output os_1_mlq_paging.*



```
Using frames after free:print_list_fp:
fp[4]
fp[3]
fp[2]
fp[1]
fp[0]

TLB Memory dump:
TLB entry 0: pid=3, valid=1, pgn=1, fpn=4, lru=0
TLB entry 1: pid=1, valid=1, pgn=0, fpn=0, lru=1
CPU 3: Put process 2 to run queue
Free regions after alloc:
print_list_rg:
rg[100->300]

Using frames after alloc:
print_list_fp:
fp[4]
fp[3]
fp[2]
fp[1]
fp[0]

CPU 3: Dispatched process 2
CPU 2: Put process 1 to run queue
CPU 2: Dispatched process 1
TLB hit at read region=1 offset=20
[PID=1] read region=1 offset=20 value=100
print_pgtbl: 0 - 768 [PID=1]
00000000: 80000000
00000004: 80000001
00000008: 80000002
Memory dump:
20: 100
Time slot 7
TLB Memory dump:
TLB entry 0: pid=3, valid=1, pgn=0, fpn=3, lru=0
TLB entry 1: pid=1, valid=1, pgn=0, fpn=0, lru=1
Loaded a process at input/proc/m0s, PID: 5 PRIO: 120
```

Hình 8: *Output os_1_mlq_paging.*



```
Time slot 7
TLB Memory dump:
TLB entry 0: pid=3, valid=1, pgn=0, fpn=3, lru=0
TLB entry 1: pid=1, valid=1, pgn=0, fpn=0, lru=1
    Loaded a process at input/proc/m0s, PID: 5 PRIO: 120
Write out of bound
    CPU 0: Put process 4 to run queue
    CPU 0: Dispatched process 5
    CPU 1: Put process 3 to run queue
    CPU 1: Dispatched process 3
Freeing region: rg_start=0 rg_end=100
Free regions after free:
print_list_rg:
rg[0->300]

Using frames after free:print_list_fp:
fp[6]
fp[5]
fp[4]
fp[3]
fp[2]
fp[1]
fp[0]

print_pgtbl: 0 - 512 [PID=5] Time slot 8

00000000: 80000005
00000004: 80000006
Free regions after alloc:
print_list_rg: NULL list
Using frames after alloc:
print_list_fp:
fp[6]
fp[5]
fp[4]
fp[3]
fp[2]
fp[1]
fp[0]
```

Hình 9: *Output os_1_mlq_paging.*



```
Loaded a process at input/proc/p1s, PID: 6 PRIO: 15
CPU 2: Put process 1 to run queue
CPU 2: Dispatched process 6
Time slot 9
CPU 3: Put process 2 to run queue
CPU 3: Dispatched process 2
Freeing region: rg_start=300 rg_end=400
Free regions after free:
print_list_rg:
rg[0->400]

Using frames after free:print_list_fp:
fp[6]
fp[5]
fp[4]
fp[3]
fp[2]
fp[1]
fp[0]

Free regions after alloc:
print_list_rg: NULL list
Using frames after alloc:
print_list_fp:
fp[6]
fp[5]
fp[4]
fp[3]
fp[2]
fp[1]
fp[0]

CPU 1: Put process 3 to run queue
CPU 1: Dispatched process 3
Freeing region: rg_start=0 rg_end=300
Free regions after free:
print_list_rg:
rg[0->300]
rg[0->400]
```

Hình 10: *Output os_1_mlq_paging.*



```
Using frames after free:print_list_fp:
fp[6]
fp[5]
fp[4]
fp[3]
fp[2]
fp[1]
Time slot 10
fp[0]

    CPU 0: Put process 5 to run queue
    CPU 0: Dispatched process 4
    Loaded a process at input/proc/s0, PID: 7 PRIO: 38
Freeing region: rg_start=0 rg_end=300
    CPU 3: Put process 2 to run queue
    CPU 3: Dispatched process 7
Free regions after free:
print_list_rg:
rg[0->300]
rg[0->300]
rg[0->400]

Using frames after free:print_list_fp:
fp[6]
fp[5]
fp[4]
fp[3]
fp[2]
fp[1]
fp[0]

    CPU 2: Put process 6 to run queue
    CPU 2: Dispatched process 6
Time slot 11
    CPU 1: Processed 3 has finished
Time slot 12
    CPU 1: Dispatched process 2
    CPU 0: Put process 4 to run queue
    CPU 0: Dispatched process 5
```

Hình 11: *Output os_1_mlq_paging.*



```
Time slot 12
    CPU 1: Dispatched process 2
    CPU 0: Put process 4 to run queue
    CPU 0: Dispatched process 5
Freeing region: rg_start=0 rg_end=300
Free regions after free:
print_list_rg:
rg[0->300]

Using frames after free:print_list_fp:
fp[6]
fp[5]
fp[2]
fp[1]
fp[0]

    CPU 3: Put process 7 to run queue
Time slot 13
    CPU 3: Dispatched process 7
Free regions after alloc:
print_list_rg:
rg[100->300]

Using frames after alloc:
print_list_fp:
fp[6]
fp[5]
fp[2]
fp[1]
fp[0]

    CPU 2: Put process 6 to run queue
    CPU 2: Dispatched process 6
    CPU 1: Put process 2 to run queue
    CPU 0: Put process 5 to run queue
    CPU 0: Dispatched process 4
    CPU 1: Dispatched process 2
Time slot 14
    CPU 3: Put process 7 to run queue
    CPU 3: Dispatched process 7
```

Hình 12: *Output os_1_mlq_paging.*



```
Time slot 14
    CPU 3: Put process 7 to run queue
    CPU 3: Dispatched process 7
    CPU 1: Processed 2 has finished
    CPU 1: Dispatched process 5
TLB hit at write region=1 offset=20 value=102
[PID=5] write region=1 offset=20 value=102
print_ptbl: 0 - 512 [PID=5]
00000000: 80000005
00000004: 80000006
Memory dump:
20: 100
1556: 102
    CPU 2: Put process 6 to run queue
    CPU 2: Dispatched process 6
Time slot 15
TLB Memory dump:
TLB entry 0: pid=5, valid=1, pgn=0, fpn=5, lru=0
TLB entry 1: pid=5, valid=1, pgn=1, fpn=6, lru=1
    Loaded a process at input/proc/s1, PID: 8 PRIO: 0
    CPU 0: Put process 4 to run queue
    CPU 0: Dispatched process 8
Time slot 16
print_ptbl: 0 - 512 [PID=8]
00000000: 80000004
00000004: 80000003
Free regions after alloc:
print_list_rg: NULL list
Using frames after alloc:
print_list_fp:
fp[3]
fp[4]
fp[6]
fp[5]
fp[2]
fp[1]
fp[0]

Write out of bound
    CPU 3: Put process 7 to run queue
```

Hình 13: *Output os_1_mlq_paging.*



```
Write out of bound
  CPU 3: Put process 7 to run queue
  CPU 1: Put process 5 to run queue
  CPU 1: Dispatched process 4
  CPU 3: Dispatched process 7
  CPU 2: Put process 6 to run queue
  CPU 2: Dispatched process 6
Time slot 17
Time slot 18
  CPU 0: Put process 8 to run queue
  CPU 0: Dispatched process 8
  CPU 3: Put process 7 to run queue
  CPU 2: Processed 6 has finished
  CPU 1: Put process 4 to run queue
  CPU 1: Dispatched process 4
  CPU 2: Dispatched process 5
TLB miss at write region=0 offset=0 value=0
[PID=5] write region=0 offset=0 value=0
print_pttbl: 0 - 512 [PID=5]
00000000: 80000005
00000004: 80000006
Memory dump:
20: 100
Time slot 19
1556: 102
  CPU 3: Dispatched process 7
TLB Memory dump:
TLB entry 0: pid=8, valid=1, pgn=0, fpn=4, lru=0
TLB entry 1: pid=5, valid=1, pgn=0, fpn=5, lru=1
  CPU 2: Processed 5 has finished
  CPU 2: Dispatched process 1
  CPU 0: Put process 8 to run queue
  CPU 0: Dispatched process 8
Time slot 20
  CPU 3: Put process 7 to run queue
Time slot 21
  CPU 3: Dispatched process 7
  CPU 1: Put process 4 to run queue
  CPU 1: Dispatched process 4
Freeing region: rg_start=300 rg_end=600
```

Hình 14: *Output os_1_mlq_paging.*



```
Time slot 21
    CPU 3: Dispatched process 7
    CPU 1: Put process 4 to run queue
    CPU 1: Dispatched process 4
Freeing region: rg_start=300 rg_end=600
Free regions after free:
print_list_rg:
rg[100->600]

Using frames after free:print_list_fp:
fp[3]
fp[4]
fp[2]
fp[1]
fp[0]

    CPU 2: Processed 1 has finished
    CPU 2 stopped
Time slot 22
    CPU 0: Put process 8 to run queue
    CPU 0: Dispatched process 8
    CPU 3: Put process 7 to run queue
    CPU 3: Dispatched process 7
    CPU 1: Processed 4 has finished
Time slot 23
    CPU 1 stopped
    CPU 0: Processed 8 has finished
    CPU 0 stopped
Time slot 24
Time slot 25
    CPU 3: Put process 7 to run queue
    CPU 3: Dispatched process 7
Time slot 26
    CPU 3: Processed 7 has finished
    CPU 3 stopped
```

Hình 15: *Output os_1_mlq_paging.*



5 Kết Luận

Bài tập lớn nói về việc hiện thực mô phỏng một hệ điều hành với các chức năng quản lý bộ nhớ, định thời CPU. Ngoài ra, thông qua hiện thực bộ quản lý bộ nhớ bằng phân trang cho BTL lần này, sinh viên sẽ nắm chắc các kiến thức liên quan đến vùng nhớ ảo, bộ nhớ RAM/SWAP và không thể thiếu giải thuật thay trang. Việc kết hợp các bộ phận trên đòi hỏi người hiện thực bài tập lớn phải có cơ chế đồng bộ hoá thích hợp để tránh xảy ra tranh chấp tài nguyên. Thông qua các testcase và đánh giá, ta sẽ hiểu được hệ điều hành hoạt động chính xác như thế nào từ đó có thể tối ưu hệ thống. Cuối cùng, cảm ơn thầy **Trần Trương Tuấn Phát** đã hỗ trợ rất nhiều



6 Tài liệu tham khảo

References

- [1] Operating System Concepts, Silberschatz, Galvin, and Gagne, 10th Ed., John-Wiley & Sons, Inc., 2018
- [2] <https://github.com/Akter8/memory-subsystem-simulator>