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

OPERATING SYSTEM

Assignment

Scheduling - Memory Management and Translation Lookaside Buffer

---

Giảng viên hướng dẫn: ThS Hoàng Lê Hải Thanh
Nhóm thực hiện: Nguyễn Lê Hoàng Phúc - 2212629
Võ Nguyễn Đức Phát - 2212540
Nguyễn Bá Việt Quang - 2212741
Trần Thanh Phong - 2212571

HO CHI MINH CITY, MAY 2024

Mục lục

1 Danh sách thành viên & phân chia nhiệm vụ **2**

2 Scheduling **3**
2.1 Scheduling trong cơ sở lý thuyết . . . 3
2.1.1 Scheduling là gì . . . 3
2.1.2 Cách thức hoạt động của Scheduling . . . 4
2.2 Scheduling trong ngữ cảnh Bài tập lớn . . . 4
2.2.1 Sự khác biệt . . . 4
2.2.2 Mô tả giải thuật MLSQ . . . 5
2.2.3 Hiện thực . . . 8
2.2.3.a Xử lý sơ bộ . . . 9
2.2.3.b Hàm *enqueue*() . . . 10
2.2.3.c Hàm *dequeue*() . . . 10
2.2.3.d Hàm *get_proc*() , *put_proc*() và *add_proc*() . . . 11
2.3 Trả lời câu hỏi . . . 14

3 Memory Management **19**
3.1 Phân trang (Paging) . . . 19
3.1.1 Phân trang (paging) là gì . . . 19
3.1.2 Cách thức hoạt động của Paging . . . 19
3.1.3 Paging trong Bài tập lớn . . . 19
3.1.3.a Hàm alloc . . . 20
3.1.3.b Hàm free . . . 21
3.1.3.c Hàm pgread . . . 22
3.1.3.d Hàm pgwrite . . . 23
3.2 Translation Lookaside Buffer . . . 26
3.2.1 Translation Lookaside Buffer là gì? . . . 26
3.2.2 Translation Lookaside Buffer trong Bài tập lớn . . . 26
3.2.2.a Hàm tlb_cache_read . . . 27
3.2.2.b Hàm tlb_cache_write . . . 28
3.2.3 Hàm tlballoc . . . 29
3.2.4 Hàm tlbfree_data . . . 29
3.2.5 Hàm tlbread . . . 30
3.2.6 Hàm tlbwrite . . . 32

4 Put It All Together **36**

5 Chạy thử Testcase **39**
5.1 Tổng quan . . . 39
5.2 CPU Scheduling . . . 39
5.3 Memory . . . 46

1 Danh sách thành viên & phân chia nhiệm vụ

STT	Họ và tên	MSSV	Nhiệm vụ	Phần trăm đóng góp
1	Võ Nguyễn Đức Phát	2212540	- Memory management - Viết báo cáo	25%
2	Trần Thanh Phong	2212571	- Scheduling - Viết báo cáo	25%
3	Nguyễn Bá Việt Quang	2212741	- CPU TLB - Viết báo cáo	25%
4	Nguyễn Lê Hoàng Phúc	2212629	- Scheduling - Viết báo cáo	25%

2 Scheduling

2.1 Scheduling trong cơ sở lý thuyết

2.1.1 Scheduling là gì

Scheduling (Định thời – phiên âm tiếng Việt này sẽ được ưu tiên sử dụng hơn thuật ngữ tiếng Anh trong báo cáo Bài tập lớn, ngoại trừ các tiêu đề), là một trong rất nhiều những chức năng (operations) của hệ điều hành. Định thời đơn giản là quá trình lựa chọn và sắp xếp các công việc để hệ thống máy tính thực thi, nhưng ở cấp độ có tổ chức và chuyên môn hóa cao.

Định thời, về mặt lý thuyết, là tổng hòa của nhiều giai đoạn khác nhau, là sự phối hợp nhịp nhàng giữa các bộ định thời trong hệ thống. Có ba loại bộ định thời: bộ định thời dài hạn (long-term scheduling), bộ định thời trung hạn (medium-term scheduling) và bộ định thời ngắn hạn (short-term scheduling). Trong đó:

- *Bộ định thời dài và trung hạn:* gắn với giai đoạn Loading, thực hiện công việc xác định những chương trình nào được chấp nhận nạp vào hệ thống và trở thành tiến trình. (disk, memory -> ready_queue).
- *Bộ định thời ngắn hạn:* gắn với giai đoạn Excecuting, thực hiện công việc xác định những tiến trình nào được quyền sử dụng CPU (ready_queue -> run_queue).

Trong phạm vi kiến thức của hệ điều hành, cơ sở lý thuyết dừng lại ở việc nghiên cứu định thời với bộ định thời ngắn hạn, tức chỉ quan tâm đến việc lựa chọn và sắp xếp các tài nguyên (tiến trình) trong quá trình thực thi – gọi chung là *CPU scheduling.*

Định thời tồn tại với hai mục tiêu chính: *một là,* đảm bảo hiệu suất CPU luôn ở mức cao (CPU ultilazation); *hai là,* đảm bảo tính tương tác cao với người dùng (interactive enhancement). Hai mục tiêu này phục vụ cho nhu cầu dung hòa giữa hai hình thức xử lý của hệ điều hành – multi-programming và multi-tasking. Có thể nói, định thời ra đời như một cách giải tỏa những mâu thuẫn mà trước đây, khi chí hướng của multi-programming và multi-tasking được gán với hai chiều định kiến trái ngược nhau, không thể nào hóa giải.

Để hoàn thành hai mục trên, hiện thực định thời cần đảm bảo các tiêu chí sau:

- Tiêu chí hướng người dùng (user – oriented):
  - Thời gian chờ đợi (waiting time): cực tiểu;
  - Thời gian đáp ứng (respond time): cực tiểu;
  - Thời gian hoàn thành (turn-around time): cực tiểu.
- Tiêu chí hướng hệ thống (system – oriented):
  - Hiệu suất CPU (CPU ultilization): cực đại;
  - Thông lượng (through-put): cực đại.

2.1.2 Cách thức hoạt động của Scheduling

Quá trình định thời được thực hiện bởi sự phối hợp liên tục của hai quá trình có thứ tự:

- *Bước 1:* lựa chọn một tiến trình phù hợp từ ready-queue và lấy ra khỏi ready-queue, quá trình này được thực hiện bởi CPU scheduler.
- *Bước 2:* cấp phát một CPU và trao quyền sử dụng CPU cho tiến trình chọn được ở *Bước* 1, quá trình này được thực hiện bởi CPU dispatcher, việc chuyển ngữ cảnh xảy ra tại bước này.

Nhìn chung, công việc ở *Bước 1* mang đậm hơi thở của logic hơn, vì việc hiện thực ở *Bước 1* mang trong mình nhiều sự lựa chọn, mỗi sự lựa chọn ứng với một giải thuật, và để có được sự lựa chọn tối ưu, cần xem xét đến nhiều vấn đề liên quan đến hiệu suất khi sử dụng giải thuật đó (trạng thái của tiến trình, khuynh hướng thực thi của tiến trình (CPU hay I/O burst),...).

Có 06 giải thuật lựa chọn tiến trình (giải thuật định thời) đã được tiếp cận:

1. First come – First serve (FCFS);
2. Shortest – Job – First (SJF);
3. Priority (Prio);
4. Round – Robin (RR);
5. Multilevel – Queue (MLQ);
6. Multilevel – Feedback – Queue (MLFQ).

2.2 Scheduling trong ngữ cảnh Bài tập lớn

Nhìn chung, bản chất của công việc định thời không thay đổi, vẫn mang hai mục tiêu chính: đa chương (tận dụng tối đa CPU) và chia thời (tối thiểu thời gian đáp ứng). Tuy nhiên, với cách mô phỏng một hệ điều hành thực bằng một hệ điều ảnh ảo, vẫn sinh ra một số sự thay đổi ...

2.2.1 Sự khác biệt

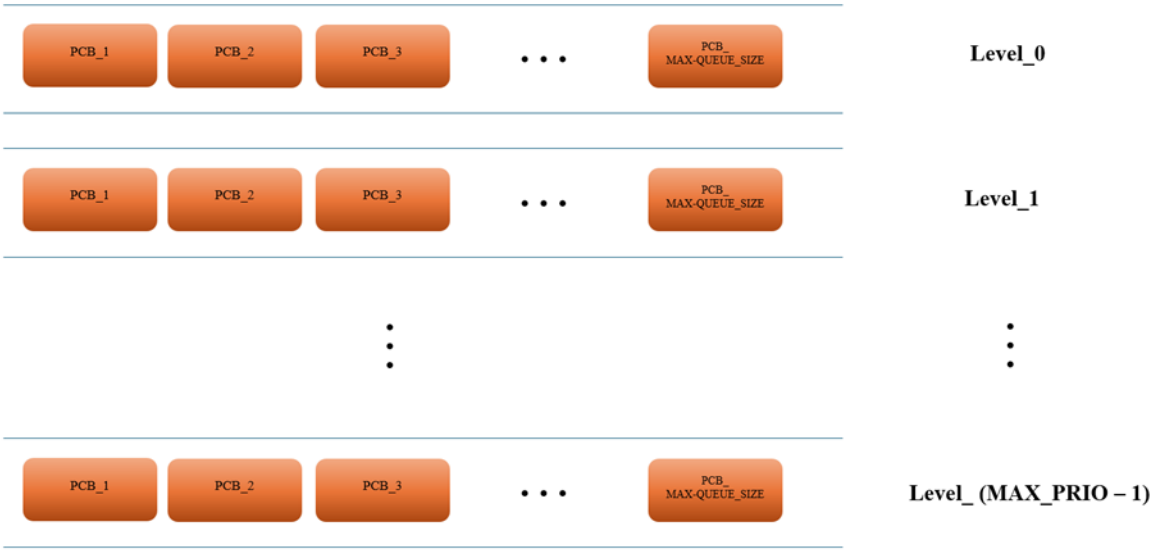
Sự khác biệt này xuất phát từ *Bước 1* trong cách thức hoạt động của bộ định thời ngắn hạn. Chính sách định thời mà hệ điều hành sử dụng là Multi Level Queue (MLQ) kết hợp Round Robin, thay vì các giải thuật thuần túy như lý thuyết đã nhắc trước đó (trong thực tế, rất ít hệ điều hành hiện thực định thời đơn giải thuật). Hơn nữa, giải thuật này bổ sung cơ chế slot như là một thay thế cho cơ chế thay đổi độ ưu tiên giữa các hàng đợi trong danh sách hàng đợi (sẽ được phân tích rõ hơn với question chung bên dưới).

Kể từ đây, quy ước *Multilevel – Slot – Queue (MLSQ)* là tên chính thức của giải thuật được sử dụng trong báo cáo Bài tập lớn này.

2.2.2 Mô tả giải thuật MLSQ

Kế thừa trực tiếp từ giải thuật MLQ (MLFQ), các tiến trình vẫn được lưu trữ trong một danh sách các queue dưới dạng mảng động (có MAX_PRIO mảng, kích thước tối đa của mỗi mảng là MAX_QUEUE _SIZE), mỗi mảng lưu giữ các con trỏ đại diện cho PCB của nhiều tiến trình khác nhau (xem Hình 1). Trong hệ thống đĩa hoặc bộ nhớ, mỗi tiến trình sở hữu một độ ưu tiên cố định (lưu trữ trong PCB) và là độ ưu tiên mặc định (default prioriy), độ ưu tiên này được dùng để xác định ready-queue nào sẽ được dùng để chứa tiến trình đó. Tuy nhiên, độ ưu tiên mặc định này sẽ bị thay thế nếu cấu hình file đầu vào (input) định nghĩa lại một độ ưu tiên mới (prio), và giá trị này ghi đè lên giá trị mặc định trước đó.

Trong nội dung chương trình ta học, quy ước prio có độ lớn càng thấp thì độ ưu tiên càng cao, và cụm từ "hàng đợi" được ngầm hiểu đang ám chỉ đến chỉ duy nhất một hàng đợi trong danh sách có tổng MAX_PRIO hàng đợi.



Hình 1: Mô phỏng cấu trúc hàng đợi trong giải thuật MLSQ

Ban đầu các hàng đợi có độ ưu tiên khác nhau sẽ có số lần sử dụng CPU khác nhau (thể hiện qua giá trị **slot**, tính bằng công thức $MAX\_PRIO - prio$), hàng đợi có độ ưu tiên càng cao thì càng có nhiều lượt sử dụng CPU. Hàng đợi có độ ưu tiên cao nhất sẽ được chọn để thi thực trước cho đến khi đã sử dụng hết số slot cho phép. Lúc này, CPU sẽ chuyển sang thực thi các process trong hàng đợi có độ ưu tiên thấp hơn. Tất cả hàng đợi sẽ được áp dụng cơ chế Round Robin, tức là việc thực thi một hàng đợi chính xác là thay phiên nhau thực thi các process trong hàng đợi ấy theo time slice cho trước, một lần thay phiên được tính là một lần sử dụng CPU.

Đến một thời điểm nhất định, tất cả các hàng đợi đều sử dụng hết số slot ban đầu, khi đó hệ thống sẽ cấp phát lại số slot cho từng hàng đợi theo công thức ban đầu và tiếp tục thực hiện công việc định thời.

Sự đổi mới trong logic chương trình:

Với giải thuật và cơ chế vừa trình bày trên, mặc dù có giảm thiểu tình trạng starvation, nhưng không thể giải quyết triệt để vấn đề đó. Để hiểu rõ hơn về starvation trong giải thuật hiện tại, ta quan sát ví dụ minh họa sau:

Ngữ cảnh đầu vào:

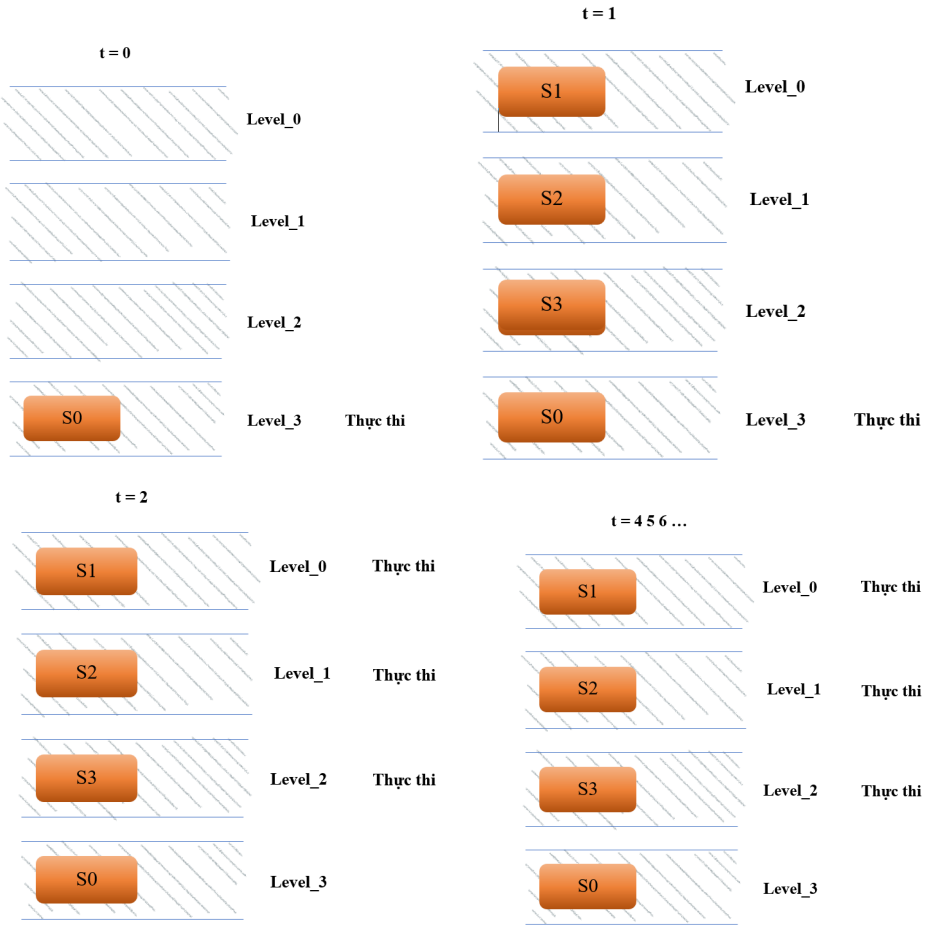
- $MAX\_PRIO = 4$ (có tất cả 4 hàng đợi trong danh sách hàng đợi)
- $MAX\_QUEUE\_SIZE = 4$ (mỗi hàng đợi có khả năng lưu trữ đồng thời 4 tiến trình)
- Có 4 tiến trình ngẫu nhiên được chọn quan sát, thuộc tính mỗi tiến trình được thể hiện trong bảng bên dưới:

File input (theo cấu trúc bài tập lớn) có nội dung như sau:

2 1 4	Time quantum = 2 – Có 1 CPU được sử dụng – Có 4 tiến trình được thử nghiệm
0 s0 3	Tiến trình s0 vào tại thời điểm t = 0, có prio = 3
1 s1 0	Tiến trình s1 vào tại thời điểm t = 1, có prio = 0
1 s2 1	Tiến trình s2 vào tại thời điểm t = 1, có prio = 1
1 s3 2	Tiến trình s3 vào tại thời điểm t = 1, có prio = 2

Tiến trình	**s0**	**s1**	**s2**	**s3**
Thuộc tính	3 13	0 8	1 10	2 15
	calc	calc	calc	calc
	calc	calc	calc	calc
	calc	calc	calc	calc
	calc	calc	calc	calc
	calc	calc	calc	calc
	calc	calc	calc	calc
	calc	calc	calc	calc
	calc	calc	calc	calc
	calc	calc	calc	calc
	calc		calc	calc
	calc		calc	calc
	calc			calc
	calc			calc
	calc			calc
				calc
				calc

Starvation sẽ xảy ra với tiến trình s3, khi tại thời điểm t = 2, s3 đã không còn quyền sử dụng CPU và slot cũng bị giảm bằng 0. Kể từ thời điểm t = 3 trở đi, các tiến trình s0, s1, s2 sẽ thực thi lần lượt cho đến khi không còn slot của mình. Khi đó, các prio sẽ được khởi tạo lại từ đầu, với giải thuật hiện tại,



điểm bắt đầu thực thi chính là ở queue level 0 (có độ ưu tiên cao nhất), lúc này tiến trình s3 đã đợi trong hàng chờ khoảng thời gian bằng tổng thời gian thực thi của s0 (4 slot), s1 (3 slot), s2 (2 slot) (giả sử khoảng thời gian chờ này là a, a = 4*2 + 3*2 + 2*2). Sau khi đã khởi tạo lại giá trị độ ưu tiên cho toàn bộ hàng đợi, tiến trình s3 lại phải tiếp tục chờ thêm một khoảng thời gian a, để có thể nhận được quyền sử dụng CPU. Như vậy, trong trường hợp này, thời gian chờ tổng thể giữa 2 lượt sử dụng CPU liên tiếp của tiến trình s3 là 2*a. Điều tồi tệ hơn sẽ xảy ra nếu số tiến trình tăng lên, có nhiều hàng đợi hơn, và hàng đợi ở các queue với level lớn (càng lớn bao nhiêu thì càng ít được ưu tiên và slot càng ít bấy nhiêu) có nhiều tiến trình phải chờ, số slot quá ít trong khi khối lượng tiến trình cần xử lý vượt quá khả năng đáp ứng, làm cho các tiến trình với độ ưu tiên thấp hầu như không thể tiếp cận được cơ hội sử dụng CPU trong một khoảng thời gian dài.

Để giảm thiểu thực trạng nêu trên, các thành viên trong nhóm đã thống nhất thực hiện đánh đổi giá trị sử dụng của prio trong giải thuật ban đầu để nâng cao khả năng tiếp cận CPU cho các tiến trình nhằm tối đa hóa tính công bằng giữa các tiến trình. Cụ thể, tại thời điểm bắt đầu khởi tạo lại giá trị slot cho toàn bộ hàng đợi (sau khi tất cả slot đã được dùng hết), điểm bắt đầu thực thi không nằm ở hàng đợi đầu tiên với độ ưu tiên cao nhất, mà là hàng đợi vừa sử dụng CPU trước đó.

Dưới đây là một vài phân tích về ưu - nhược điểm của giải thuật sửa đổi:

- Ưu điểm:
  - Starvation được giải quyết tốt hơn: Việc thực hiện vòng tròn trong công tác cấp phát CPU (lấy tiến trình tiếp theo sau tiến trình đã thực thi trước đó thay vì lấy tiến trình tại hàng đợi đầu tiên), sẽ nâng cao cơ hội có được quyền sử dụng CPU của các tiến trình với độ ưu tiên thấp. Đây là điều mà giải thuật trước đó không đề cập đến, tiềm ẩn nguy cơ starvation khi khối lượng xử lý trở nên lớn và phức tạp. Giải thuật mới giới hạn khoảng thời gian chờ đợi tối đa của một tiến trình là *một lần duyệt*.
  - Nâng cao tính công bằng giữa các tiến trình: Việc không tuân theo nguyên tắc "độ ưu tiên" vô hình trung làm tăng tính công bằng khi không quan trọng hóa quá mức về độ ưu tiên giữa các tiến trình, đánh đổi tính ưu tiên bằng tính đáp ứng.
- Nhược điểm:
  - Giảm tính thiết thực của thuật toán trong vài trường hợp: Đây cũng chính là nhược điểm lớn của giải thuật mới, để giảm thiểu starvation, buộc giải thuật phải đánh đổi mức độ ưu tiên của tiến trình này để tăng hiệu suất đáp ứng của các tiến trình khác. Trong thực tế, điều này không nên được xảy ra mà thiếu sự kiểm soát chặt chẽ, bởi khi đã gán giá trị ưu tiên cho tiến trình, đồng nghĩa với việc phải cấp phát CPU kịp thời để phản hồi đúng với giá trị độ ưu tiên đó. Trong giải thuật mới, nhóm đã không thay đổi số slot của các hàng đợi mà giữ nguyên với giá trị như giải thuật cũ, chính là để tăng khả năng kiểm soát hệ thống, đảm bảo rằng độ ưu tiên vẫn còn giá trị và vẫn được xem xét, dù giảm mức độ quan trọng của nó đi ít nhiều.

2.2.3 Hiện thực

Trước hết, để hiểu rõ luồng thực thi của chương trình mô phỏng, cần đọc và nắm được ý nghĩa cơ bản của các hiện thực trong file `os.c`. `os.c` trừu tượng hóa các thiết lập phức tạp, chỉ hiện thực các hàm chính thông qua việc tái sử dụng các định nghĩa đã được hiện thực ở các file khác, cụ thể:

- Hàm `cpu_routine(void * args)`:
  - Công dụng: Thực hiện việc xử lý các tiến trình trên một CPU ảo, quản lý thời gian cho mỗi tiến trình và điều chỉnh việc chuyển đổi giữa các tiến trình.
  - Các hàm chính được gọi bao gồm:
    - `get_proc()`: Lấy một tiến trình từ hàng đợi sẵn sàng.
    - `put_proc(struct pcb_t * proc)`: Đưa một tiến trình vào hàng đợi sẵn sàng.
    - `run(struct pcb_t * proc)`: Thực thi một tiến trình.
- Hàm `ld_routine(void * args)`:
  - Công dụng: Dọc các tiến trình từ tệp cấu hình và nạp chúng vào bộ nhớ, sẵn sàng cho việc thực thi.
  - Các hàm chính được gọi bao gồm:

* `add_proc(struct pcb_t * proc)`: Thêm một tiến trình vào hàng đợi sẵn sàng để thực thi.
* `load(char *path)`: nạp một tiến trình từ tệp cấu hình được chỉ định bởi path.

Như vậy, có ba hàm trong dung lượng phần định thời được sử dụng trong file os.c: `get_proc()`, `put_proc()` và `add_proc()`, điều này có nghĩa chúng ta cần quan sát và hiệu chỉnh nội dung các file này cho phần định thời.

2.2.3.a Xử lý sơ bộ

Trong cơ chế MLQ để giám sát hàng đợi nào vừa sử dụng CPU trước thì ta khai báo thêm biến `current_queue_index` để lưu lại vị trí của hàng đợi vừa sử dụng CPU. Quá trình này diễn ra tại `file sched.c`.

```
static int current_queue_index=0;
```

Để phát triển cơ chế slot (điểm đặc biệt của giải thuật MLSQ), ta cần bổ sung định nghĩa thuộc tính slot với kiểu số nguyên đặc trưng cho số lần được phép sử dụng CPU của hàng đợi. Công dụng của biến slot là theo dõi số vé được thực thi với CPU, mỗi lần CPU thực hiện xong một tiến trình trong khoảng thời gian cố định, giá trị của biến slot sẽ giảm một đơn vị. Quá trình này diễn ra tại struct `queue_t`, `file queue.h`, folder `src`:

```
struct queue_t {
    struct pcb_t * proc[MAX_QUEUE_SIZE];
    int size;
    #ifdef MLQ_SCHED
    int slot;
    #endif
};
```

Hàm `init_scheduler()` bổ sung công việc khởi tạo các giá trị cho kích thước và số slot của tất cả các hàng đợi. Điều này được thực hiện bằng một vòng lặp duyệt qua từng hàng đợi trong danh sách có MAX_PRIO hàng đợi, ở mỗi lần xét duyệt, gán giá trị `mlq_ready_queue[i].size = 0;` và `mlq_ready_queue[i].slot = MAX_PRIO-i;` (với i là biến đếm vòng lặp, cũng chính là giá trị prio chính thức của tiến trình).

Tuy nhiên, để đảm bảo không xảy ra các xung đột dữ liệu và luồng thực thi chương trình về sau, cần đặt khai báo trên trong một block `#indef` và `#endif`. Run-queue không được đề cập và xem xét trong cả lý thuyết lẫn dung lượng Bài tập lớn, tuy nhiên như vẫn được sử dụng để tránh trường hợp không tương thích xảy ra.

```
void init_scheduler(void) {
#ifdef MLQ_SCHED
    int i ;
    for (i = 0; i < MAX_PRIO; i ++){
        mlq_ready_queue[i].size = 0;
        mlq_ready_queue[i].slot = MAX_PRIO-i;
    }
```

```
#endif
    ready_queue.size = 0;
    run_queue.size = 0;
    pthread_mutex_init(&queue_lock, NULL);
}
```

Kết thúc quá trình xử lý sơ bộ, ta tạo được một tiền đề với đủ các cơ sở để phát triển giải thuật MLSQ. Theo đó, nội dung các file queue.c và sched.c sẽ bị ảnh hưởng trực tiếp bởi sự thay đổi này.

2.2.3.b Hàm *enqueue*()

- Công dụng: đưa một tiến trình vào hàng đợi (queue) được định nghĩa trước đó, mở rộng hàng đợi nếu cần.
- Cơ chế:
  - Hàm nhận hai đối số:
    - `q`: Con trỏ đến hàng đợi (queue) cần thêm tiến trình.
    - `proc`: Con trỏ đến tiến trình cần được thêm vào hàng đợi.
  - Trước khi thêm tiến trình vào hàng đợi, hàm kiểm tra xem hàng đợi đã đầy chưa (đã đạt đến kích thước tối đa hay không).
  - Nếu hàng đợi đã đầy (`q->size = MAX_QUEUE_SIZE`), hàm sẽ không thực hiện thêm tiến trình nào và kết thúc ngay lập tức.
  - Nếu hàng đợi chưa đầy, tiến trình được truyền vào sẽ được đặt vào cuối hàng đợi (`q->proc[q->size] = proc`) và biến `q->size` sẽ được tăng lên một (`q->size++`). Kết quả của quá trình này là làm mở rộng kích thước của hàng đợi và thêm tiến trình mới vào hàng đợi.

```
void enqueue(struct queue_t * q, struct pcb_t * proc) {
    /* TODO: put a new process to queue [q] */
    if (q->size == MAX_QUEUE_SIZE) {
        return;
    }
    q->proc[q->size++] = proc;
}
```

2.2.3.c Hàm *dequeue*()

- Công dụng: Trả về tiến trình đầu tiên từ hàng đợi và loại bỏ khỏi hàng đợi.
- Cơ chế:
  - Hàm nhận một tham số đầu vào là con trỏ tới hàng đợi q.
  - Đầu tiên, hàm kiểm tra xem hàng đợi có trống không (empty(q)).
    - Nếu có, nghĩa là không có tiến trình nào trong hàng đợi, hàm trả về NULL.

* Nếu không, tiến trình có vị trí đầu tiên trong hàng đợi (index = 0) sẽ được chọn.

- Sau đó, khi tiến trình được chọn, hàm di chuyển các tiến trình còn lại trong hàng đợi về phía trước 1 index để loại bỏ tiến trình đã chọn.
- Kích thước của hàng đợi q sẽ giảm đi một sau khi tiến trình được loại bỏ khỏi hàng đợi.
- Tiến trình được chọn được trả về.

```
struct pcb_t * dequeue(struct queue_t * q) {
    /* TODO: return a pcb whose prioprity is the highest
    * in the queue [q] and remember to remove it from q
    * */
    if (!q || empty(q)) return NULL;
    struct pcb_t * front = q->proc[0];
    for (int i = 0; i < q->size - 1; i++) {
            q->proc[i] = q->proc[i + 1];
    }
    q->size--;
    return front;
}
```

2.2.3.d Hàm *get_proc*() , *put_proc*() và *add_proc*()

Hệ thống có thể được cấu hình để sử dụng một trong hai chính sách định th khác nhau: Multiple-Level-Slot-Queue (MLQ) hoặc Single-Level Queue. Mỗi hàm trong đoạn mã thực hiện một phần của quy trình quản lý tiến trình.

1. `get_mlq_proc()`:
   - Cơ chế: Hàm này được sử dụng khi hệ thống sử dụng MLQ scheduling.
     - Nó lấy một tiến trình từ hàng đợi ưu tiên MLQ dựa trên các quy tắc của hệ thống MLQ.
     - Sau khi duyệt tới hàng đợi có độ ưu tiên thấp nhất thì thiết lập lại trạng thái của các hàng đợi theo quy tắc MLQ.
   - Mối liên hệ: Sử dụng các hàm `dequeue()` và `empty()` để lấy và kiểm tra hàng đợi.
2. `put_mlq_proc()` và `add_mlq_proc()`:
   - Cơ chế: Đặt một tiến trình vào hàng đợi ưu tiên MLQ.
   - Mối liên hệ: Sử dụng hàm enqueue() để thêm tiến trình vào hàng đợi.
3. `get_proc()`, `put_proc()`, và `add_proc()`:
   - Cơ chế: Các hàm này là các bản gốc của các hàm tương ứng trong MLQ scheduling nhưng được sử dụng khi hệ thống không sử dụng MLQ scheduling.
   - Mối liên hệ: Nó chuyển hướng yêu cầu tới các hàm tương ứng của MLQ hoặc hàng đợi đơn khi hệ thống được cấu hình sử dụng một trong hai chính sách lập lịch.

```
    #ifdef MLQ_SCHED
```

```c
/*
 *  Stateful design for routine calling
 *  based on the priority and our MLQ policy
 *  We implement stateful here using transition technique
 *  State representation   prio = 0 .. MAX_PRIO, curr_slot = 0..(MAX_PRIO - prio)
 */
struct pcb_t * get_mlq_proc(void) {
  struct pcb_t * proc = NULL;
  /*TODO: get a process from PRIORITY [ready_queue].
   * Remember to use lock to protect the queue.
   */
  pthread_mutex_lock(&queue_lock);
  int i=current_queue_index;
  int num_queue_empty = 0;
  while(1) {
    current_queue_index=i;
    if (!empty(&mlq_ready_queue[i])) {
      if (mlq_ready_queue[i].slot > 0) {
        proc = dequeue(&mlq_ready_queue[i]);
        mlq_ready_queue[i].slot--;
        break;
      }
    }
    else {
      num_queue_empty++;
    }
    if (num_queue_empty == MAX_PRIO) {
      current_queue_index=(current_queue_index+1)%MAX_PRIO;
      break;
    }
    if (i == MAX_PRIO-1) {
      for (int j = 0; j < MAX_PRIO; j++) {
        mlq_ready_queue[j].slot = MAX_PRIO - j;
      }
    }
    i=(i+1)%MAX_PRIO;
  }
  pthread_mutex_unlock(&queue_lock);
  return proc;
}

void put_mlq_proc(struct pcb_t * proc) {
  pthread_mutex_lock(&queue_lock);
  enqueue(&mlq_ready_queue[proc->prio], proc);
  pthread_mutex_unlock(&queue_lock);
}

void add_mlq_proc(struct pcb_t * proc) {
  pthread_mutex_lock(&queue_lock);
  enqueue(&mlq_ready_queue[proc->prio], proc);
  pthread_mutex_unlock(&queue_lock);
}
```

```
struct pcb_t * get_proc(void) {
  return get_mlq_proc();
}

void put_proc(struct pcb_t * proc) {
  return put_mlq_proc(proc);
}

void add_proc(struct pcb_t * proc) {
  return add_mlq_proc(proc);
}
#else
struct pcb_t * get_proc(void) {
  struct pcb_t * proc = NULL;
  /*TODO: get a process from [ready_queue].
   * Remember to use lock to protect the queue.
   * */

  pthread_mutex_lock(&queue_lock);
  proc = dequeue(&ready_queue);
  pthread_mutex_unlock(&queue_lock);
  return proc;
}

void put_proc(struct pcb_t * proc) {
  pthread_mutex_lock(&queue_lock);
  enqueue(&run_queue, proc);
  pthread_mutex_unlock(&queue_lock);
}

void add_proc(struct pcb_t * proc) {
  pthread_mutex_lock(&queue_lock);
  enqueue(&ready_queue, proc);
  pthread_mutex_unlock(&queue_lock);
}
#endif
```

Cả hai phần của đoạn mã đều sử dụng khóa mutex (queue_lock) để bảo vệ khỏi sự cạnh tranh giữa các luồng khi truy cập và sửa đổi hàng đợi. Dưới đây là phân tích sơ bộ về khóa mutex:

1. Vùng tranh chấp (Critical section):
   - Các vùng chứa các thao tác trên hàng đợi (queue), bao gồm các thao tác thêm (enqueue) và lấy ra (dequeue) các tiến trình.
   - Trong MLQ scheduling:
     - Vùng tranh chấp nằm trong các hàm `get_mlq_proc()`, `put_mlq_proc()`, và `add_mlq_proc()`.
     - Trong hàng đợi MLQ, các thao tác như lấy tiến trình, thêm tiến trình vào hàng đợi và đặt tiến trình vào hàng đợi đều là các thao tác quan trọng và cần phải được thực hiện một cách an toàn để tránh sự cạnh tranh và xung đột dữ liệu.

- Trong Single-Level Queue:
  - Vùng tranh chấp nằm trong các hàm `get_proc()`, `put_proc()`, và `add_proc()`.

2. Vì sao sử dụng mutex trong trường hợp này:
   - Mutex được sử dụng để đảm bảo rằng chỉ có một luồng có thể thực hiện một thao tác trên hàng đợi vào một thời điểm.
   - Mục tiêu là ngăn chặn xung đột dữ liệu khi nhiều luồng cùng truy cập và sửa đổi các dữ liệu chia sẻ, như là hàng đợi tiến trình trong trường hợp mà ta đang xét.
3. Cách sử dụng mutex:
   - Trước khi truy cập hoặc sửa đổi hàng đợi, luồng cần đợi để giữ khóa mutex. Khi một luồng đã có khóa mutex, nó có thể thực hiện các thao tác trên hàng đợi. Sau khi hoàn thành các thao tác, luồng sẽ giải phóng khóa mutex để cho phép các luồng khác tiếp tục thực hiện thao tác của mình.
   - Cụ thể:
     - Mutex lock (`pthread_mutex_lock`) được sử dụng để khóa vùng nhớ chia sẻ trước khi thực hiện bất kỳ thao tác nào trên hàng đợi.
     - Sau khi thực hiện xong các thao tác trên hàng đợi, mutex được mở khóa (`pthread_mutex_unlock`).
4. Đánh giá:
   - Sử dụng mutex lock là một cách tiếp cận tốt để đảm bảo tính nhất quán và độ tin cậy của dữ liệu trong môi trường đa luồng.
   - Bằng cách sử dụng mutex, đoạn mã tránh được sự cạnh tranh và xung đột dữ liệu giữa các luồng khi truy cập và sửa đổi hàng đợi.
   - Tuy nhiên, sử dụng mutex cũng có thể gây ra tình trạng đợi (waiting) nếu một luồng đang giữ khóa mutex và một luồng khác muốn truy cập vùng nhớ được khóa.
   - Để tối ưu hiệu suất, có thể cân nhắc sử dụng các cơ chế đồng bộ hóa khác như spinlock hoặc các cấu trúc dữ liệu đồng bộ hóa không chặn (non-blocking synchronization). Tuy nhiên, sử dụng mutex lock vẫn là một giải pháp đơn giản và hiệu quả trong nhiều trường hợp.

2.3 Trả lời câu hỏi

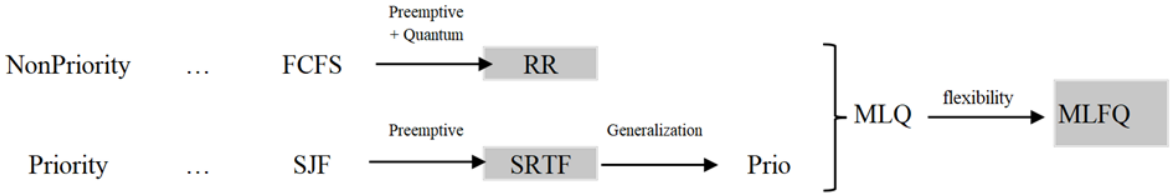
Câu hỏi:

What is the advantage of the scheduling strategy used in this assignment in comparison with other scheduling algorithms you have learned? *(Việc thực hiện chiến lược định thời như trong Bài tập lớn lần này có những ưu điểm gì so với việc thực hiện những chiến lược định thời khác mà bạn đã học?)*

Trả lời:

Như đã đề cập trước đó, chúng ta đã tiếp cận được 06 giải thuật định thời khác nhau. Tuy nhiên, nhằm nâng cao chất lượng và cải thiện mức độ khách quan của việc so sánh, chúng ta sẽ xét các phiên bản nâng cao của các giải thuật đó. Và sau khi xem xét và loại bỏ các giải thuật sơ khai, ta có 03 giải thuật tốt nhất: Shortest_Remaining_Time_First (SRTF), Round_Robin (RR), Multilevel_FeedBack_Queue (MLFQ). Chúng ta sẽ tiến hành so sánh 03 giải thuật trên với giải thuật được sử dụng trong khuôn khổ Bài tập lớn.

Trong dung lượng phần định thời của Bài tập lớn học kỳ 232, ta xem xét giải thuật Multilevel_Queue (priority + roundrobin) kết hợp cơ chế slot, ta gọi giải thuật này là Multilevel_Slot_Queue (MLSQ).



Hình 2: Sơ đồ tóm tắt mối liên hệ giữa các giải thuật

Bảng 1: Bảng so sánh chi tiết các chiến lược định thời

Chiến lược định thời	**Ưu điểm**	**Nhược điểm**
Shortest Remaining Time First (SRTF)	• *Thời gian đáp ứng được tối ưu hóa (optimal response time)*: lần lượt thực thi các tiến trình có thời gian sử dụng CPU ngắn nhất đến dài nhất, điều này cũng làm giảm thời gian chờ trong hàng đợi ready (Average waiting time) xuống mức thấp nhất. • *Starvation được giảm thiểu cho các tiến trình với thời gian thực thi CPU ít*: các tiến trình có CPU burst ngắn không bị cản trở bởi các tiến trình có CPU burst dài.	• *Không thực tế* : vì hầu như không thể biết trước lượng thời gian sử dụng CPU trong tương lai. • *Phương pháp giải quyết*: nhìn về quá khứ để ước lượng thời gian sử dụng CPU trong tương lai. • *Starvation có thể xảy ra với các tiến trình có thời gian thực thi CPU dài..* • *Tăng nguy cơ overhead*: do đặc tính preemptive, cần phải kiểm tra và thực hiện chuyển ngữ cảnh thường xuyên.

Tiếp tục vào trang sau

Bảng 1 – Tiếp tục từ trang trước

Chiến lược định thời	**Ưu điểm**	**Nhược điểm**
Round Robin (RR)	• *Nâng cao tính tương tác của hệ thống*: mọi tiến trình đều có lượng thời gian sử dụng CPU như nhau trong một chu kỳ (công bằng), đây là tiêu chí của hệ thống multi-tasking. • *Dễ hiện thực*: kế thừa tính giản đơn của giải thuật FCFS, chỉ bổ sung công tác chuyển ngữ cảnh khi cần thiết. • *Không xuất hiện Starvation*: các tiến trình lần lượt sử dụng CPU trong 1 time quantum (time slice) cố định, sau đó nhường lại cho các tiến trình khác, do đó không có tiến trình nào bị "bỏ rơi". Nếu biết trước được time quantum và số lượng tiến trình trong ready-queue, có thể xác định chính xác thời gian tối đa một tiến trình phải chờ để có thể tiếp tục sử dụng CPU.	• *Average waiting time tương đối lớn nếu số lượng tiến trình nhiều*: càng nhiều tiến trình trong ready-queue, thời gian tối đa để một tiến trình có thể tiếp tục sử dụng CPU càng lớn, thời gian chờ đợi trong ready-queue tỷ lệ thuận với số tiến trình hiện có. • *Tăng nguy cơ overhead*: do đặc tính preemptive, cần phải kiểm tra và thực hiện chuyển ngữ cảnh thường xuyên. • *Vấn đề cài đặt time quantum*: nếu time quantum không đủ bé, hiệu suất của RR tiệm cận với FCFS, nếu time quantum không đủ lớn, hiệu suất của RR giảm rõ vì thời gian chuyển ngữ cảnh lớn so với thời gian thực thi với CPU.

Tiếp tục vào trang sau

Bảng 1 – Tiếp tục từ trang trước

Chiến lược định thời	Ưu điểm	Nhược điểm
Multilevel Feedback Queue (MLFQ)	Là sự kết hợp giữa RR và Prio: • *Cân bằng average waiting time và đặc tính interactive*: các level có độ ưu tiên khác nhau làm giảm thời gian chờ đợi giữa các tiến trình, các level có cùng độ ưu tiên có cơ hội sử dụng CPU như nhau làm tăng tính tương tác của hệ thống. • *Hạn chế Starvation*: các tiến trình có độ ưu tiên thấp áp dụng phương pháp aging (định tuổi) để tăng độ ưu tiên, trở thành level có độ ưu tiên cao hơn, từ đó tiến trình chắc chắn sẽ được thực thi. • *Các tiến trình có độ ưu tiên cao sẽ được ưu ái*: tuy MLFQ vẫn có sự ưu ái, nhưng so với MLSQ việc sử dụng cơ chế slot thể hiện rõ hơn về mức độ quan trọng của các tiến trình có độ ưu tiên cao. Nâng cao tính thực dụng của chương trình vì đảm bảo đủ khả năng đáp ứng được các nhu cầu quan trọng nhất. • *Cân bằng average waiting time và đặc tính interactive*: các level có độ ưu tiên khác nhau làm giảm thời gian chờ đợi giữa các tiến trình, các level có cùng độ ưu tiên có cơ hội sử dụng CPU như nhau làm tăng tính tương tác của hệ thống.	• *Phức tạp hóa công tác hiện thực*: phát triển từ MLQ, thừa hưởng trọn vẹn những công đoạn khó nhằn vốn có, đồng thời phải cung cấp cơ chế switching giữa các level. • *Tăng nguy cơ overhead*: do cần phải theo dõi và điều chỉnh các thông số biến động của tiến trình. • *Nguy cơ Starvation tiềm tàng*: việc không hỗ trợ phương pháp aging để thay đổi độ ưu tiên các tiến trình làm giảm tính linh hoạt của giải thuật, đồng thời tăng nguy cơ "đói" cho các tiến trình vốn dĩ có độ ưu tiên thấp. Tuy nhiên với cơ chế slot, đã giảm thiểu được nguy cơ này. (phân tích rõ ở phần sau). • *Không tối ưu hóa hiệu suất*: việc sử dụng độ ưu tiên cố định cho tiến trình có thể dẫn đến lãng phí tài nguyên CPU vì làm tăng average waiting time. Ví dụ: một tiến trình ngắn bị đặt vào hàng đợi level có độ ưu tiên thấp khiến nó dành nhiều thời gian trong ready queue hơn là thực thi với CPU.

Phân tích cơ chế slot trong MLSQ:

Việc sử dụng slot trong MLSQ có những tác dụng sau:

- *Đặt ra sự tách biệt về tầm quan trọng giữa các tiến trình có độ ưu tiên thấp và các tiến trình có độ ưu tiên cao:* Các tiến trình với độ ưu tiên thấp sẽ được cấp phát ít slot để thực thi hơn và ngược

lại, các tiến trình với độ ưu tiên cao sẽ được cấp phát nhiều slot để thực thi hơn. Bởi vốn dĩ, trong công thức $slot = MAX\_PRIO - priority$ đã thể hiện rõ mối tương quan tuyến tính nghịch giữa biến slot và priority, rằng: độ lớn priority càng lớn (độ ưu tiên thấp) thì slot càng ít. Một số sự so sánh nhỏ như sau với chủ thể là MLSQ:

  - So với SRTF: Trong SRTF, thời gian còn lại của mỗi tiến trình được sử dụng để quyết định tiến trình nào sẽ được chạy tiếp theo. Trong khi đó, MLSQ sử dụng slot cố định, không phụ thuộc vào thời gian còn lại mà chỉ dựa trên mức ưu tiên.

  - So với RR: RR chia thời gian CPU thành các time slice đồng đều cho mỗi tiến trình. Trái lại, MLSQ sử dụng slot cố định dựa trên mức ưu tiên, cho phép các tiến trình ưu tiên cao hơn chạy nhiều hơn trong một đơn vị thời gian.

- *Giảm thiểu nguy cơ Starvation:* mặc dù trong bảng so sánh việc sử dụng MLSQ sẽ gây ra Starvation bởi tính không linh hoạt về độ ưu tiên giữa các tiến trình, thế nhưng cơ chế slot đã cung cấp một giải pháp giảm thiểu nguy cơ đó. Bằng cách cấp phát một lượng slot sử dụng CPU cố định cho mỗi level khác nhau, khi đó nếu một tiến trình có độ ưu tiên cao mà slot đã hết thì nó buộc phải nhường CPU lại cho các tiến trình ở level thấp hơn.

- *Giảm độ phức tạp trong công tác hiện thực:* khác với MLFQ cần phải cung cấp cơ chế switching level giữa các tiến trình, MLSQ không thực hiện điều đó, mà chỉ cần cung cấp thêm duy nhất một thuộc tính trong mỗi hàng đợi là slot. Việc sử dụng slot được ví như phần bù cho tính không linh hoạt của giải thuật, chính slot sẽ tạo ra sự linh hoạt đó nhưng theo cách riêng của mình.

Như vậy, ***MLFQ buộc tiến trình thực hiện nội biến đổi để giành được quyền sử dụng CPU, trong khi MLSQ chủ động thực hiện ngoại biến đổi mang quyền sử dụng CPU cho tiến trình***. MLSQ không chỉ kế thừa được những lợi điểm của MLFQ (tức kế thừa được mặt tốt của cả RR và Prio), mà còn có một sử cải biến hết sức mới lạ, và sự mới lạ này đã mang lại một góc nhìn khác của công việc định thời, đó là: các tiến trình không cần phải tự thay đổi bản thân để có thể giành lấy quyền thực thi với CPU, mà chỉ cần thực hiện một thay đổi nhỏ với thuộc tính của mình và quyền thực thi CPU tự khắc sẽ đến.

3 Memory Management

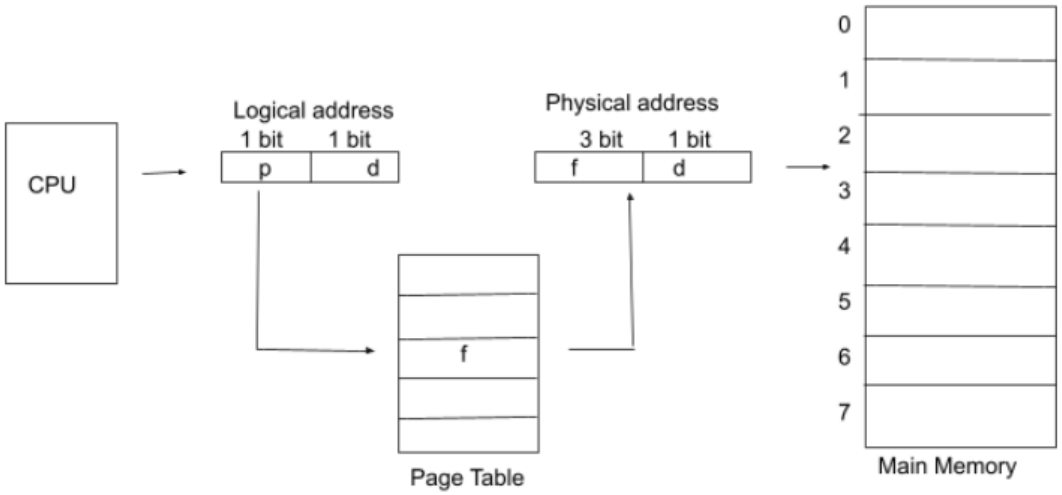
3.1 Phân trang (Paging)

3.1.1 Phân trang (paging) là gì

Paging là một phương pháp cho phép tiến trình truy cập vào bộ nhớ ảo. Tiến trình truy cập vào cái trang mà nó cần sẽ không phải đợi chờ nó được load trên bộ nhớ vật lý. Kỹ thuật phân trang cho phép ta lưu trữ và truy xuất dữ liệu từ bộ nhớ thứ 2 của máy tính (ổ cứng, SSD) hoặc từ bộ nhớ ảo đến bộ nhớ chính (RAM).

3.1.2 Cách thức hoạt động của Paging

Ban đầu, hệ điều hành sẽ chia cả bộ nhớ chính và bộ nhớ thứ cấp thành các khối có kích thước cố định. Các khối ở bộ nhớ thứ cấp được gọi là Các trang (Pages), trong khi ở bộ nhớ chính các khối được gọi là Frames. Khi một chương trình được thực thi, chương trình đó sẽ được phân chia thành các trang và Hệ điều hành sẽ chuyển các trang đó lưu vào bộ nhớ thứ cấp. Khi CPU muốn gọi 1 tiến trình nào đó trong bộ nhớ thì CPU cần được cung cấp thông tin về số trang (page number) và số offset (offset) của tiến trình đó trong bộ nhớ thứ cấp. Hình dưới đây mô tả rõ về cách thức hoạt động của việc chuyển từ bộ nhớ ảo sang bộ nhớ chính. Ở đây ta có, p là page number và d là offset. Khi CPU yêu cầu 1 tiến trình



nào đó, CPU sẽ cung cấp địa chỉ trang và offset của tiến trình đó. Lúc này, hệ điều hành sẽ đi vào Page Table (nơi chứa các frame number) và lấy ra số trang của frame đó kết hợp với offset đã có ta sẽ có được địa chỉ vật lý của Process cần tìm.

3.1.3 Paging trong Bài tập lớn

Trong bài tập lớn này, trong bộ nhớ ảo sẽ gồm nhiều area giống nhau (tương ứng với page table như trên). Và kích thước của Page Table được khai báo như sau trong chương trình C:

```
struct mm_struct {
uint32_t *pgd;
```

```
struct vm_area_struct *mmap;

/* Currently we support a fixed number of symbol */
struct vm_rg_struct symrgtbl[PAGING_MAX_SYMTBL_SZ];

/* list of free page */
struct pgn_t *fifo_pgn;
};
```

Ta có pgd là lưu toàn bộ những cái page table entry của hệ thống, có kích thước Page number là cố định. Code trên tạo ra để ta có thể dễ dàng theo dõi từng area của virtual memory thông qua con trỏ mmap. Nhiệm vụ của chúng em là cần phải hiện thực các hàm allocate, free, read và write để có thể cung cấp cho process 1 vùng nhớ nhất định và thực hiện trên đó. Trong bài tập lớn này, vùng nhớ ảo có thêm 1 cái Buffer là Translation Lookaside Buffer, nên là việc cấp phát bộ nhớ ảo, giải phóng... sẽ được thực hiện chung ở file *cpu-tlb.c*.

3.1.3.a Hàm alloc

Ở trong file mm-vm.c, đây là hàm được gọi để cấp phát 1 vùng nhớ ảo mới cho tiến trình. Hàm này được mô tả như sau:

```
int __alloc(struct pcb_t *caller, int vmaid, int rgid, int size, int *alloc_addr)
{
/*Allocate at the toproof */
struct vm_rg_struct rgnode;

if (get_free_vmrg_area(caller, vmaid, size, &rgnode) == 0)
{
caller->mm->symrgtbl[rgid].rg_start = rgnode.rg_start;
caller->mm->symrgtbl[rgid].rg_end = rgnode.rg_end;

*alloc_addr = rgnode.rg_start;

return 0;
}

/* TODO get_free_vmrg_area FAILED handle the region management (Fig.6)*/

/*Attempt to increate limit to get space */
struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);
int old_sbrk;

old_sbrk = cur_vma->sbrk;
int remain = cur_vma->vm_end - old_sbrk;
int inc_sz;
/* TODO INCREASE THE LIMIT
* inc_vma_limit(caller, vmaid, inc_sz)
*/
// newcode
if (size > remain) // not enough space for allocation
```

```
  {
    inc_sz = PAGING_PAGE_ALIGNSZ(size - remain);
    if (inc_vma_limit(caller, vmaid, inc_sz) < 0) // overlap or out of mem
    return -1;
  }
// endnewcode
/*Successful increase limit */
caller->mm->symrgtbl[rgid].rg_start = old_sbrk;
caller->mm->symrgtbl[rgid].rg_end = old_sbrk + size;

*alloc_addr = old_sbrk;
cur_vma->sbrk += size;
return 0;
}
```

Hàm bắt đầu bằng việc tìm kiếm những vùng nhớ ảo còn trống có kích thước đủ để đáp ứng yêu cầu. Nếu tìm thấy, nó cập nhật địa chỉ của vùng nhớ đó và trả về địa chỉ bắt đầu của vùng nhớ ảo đã được cấp phát. Nó cần biết vị trí bắt đầu, để từ đó có thể mở rộng từ vị trí bắt đầu đó. Cụ thể là nó sẽ được lưu vào biến *alloc_addr* nếu tìm thấy. Nếu như không thấy vùng nhớ ảo với kích thước cần thiết, hàm sẽ tăng giới hạn của vùng nhớ ảo hiện tại bằng cách thay đổi giá trị của sbrk. Trường hợp kích thước bộ nhớ không đủ, hàm sẽ trả về -1. Ngược lại nếu thành công, sẽ cập nhật giá trị alloc_addr = old_skbr, và tiến hành cộng thêm cho sbrk kích thước mà ta cần mở rộng.

3.1.3.b Hàm free

Ở trong file *mm-vm.c*, đây là hàm được gọi để giải phóng 1 vùng nhớ ảo đã được cấp phát.

```
int __free(struct pcb_t *caller, int vmaid, int rgid)
{
    struct vm_rg_struct *rgnode;

    if (rgid < 0 || rgid > PAGING_MAX_SYMTBL_SZ)
    return -1;

    /* TODO: Manage the collect freed region to freerg_list */
    rgnode = get_symrg_byid(caller->mm, rgid);
    /*enlist the obsoleted memory region */
    enlist_vm_freerg_list(caller->mm, rgnode);

    return 0;
  }
```

Khi được gọi hàm, ban đầu hàm kiểm tra xem rgid (ID của vùng nhớ ảo được cấp phát) có hợp lệ không, nếu không hợp lệ thì return -1. Nếu hợp lệ, gọi đến hàm get_symrg_byid để có được thông tin của vùng nhớ ảo được cấp phát từ bảng chỉ thị của tiến trình. Sau đó là tiến hành giải phóng vùng nhớ này sử dụng lệnh enlist_vm_free_list để thêm vùng nhớ vừa giải phóng vào freelist, cuối cùng trả về 0 để báo rằng hàm này đã thực hiện thành công.

3.1.3.c Hàm pgread

Ở trong file *mm-vm.c*, mục đích của hàm này là đọc dữ liệu của tiến trình có trong Page Table.

```
int pgread(
struct pcb_t *proc, // Process executing the instruction
uint32_t source,    // Index of source register
uint32_t offset,    // Source address = [source] + [offset]
uint32_t destination)
{
    BYTE data;
    int val = __read(proc, 0, source, offset, &data);

    destination = (uint32_t)data;
    #ifdef IODUMP
    printf("read region=%d offset=%d value=%d\n", source, offset, data);
    #ifdef PAGETBL_DUMP
    print_pgtbl(proc, 0, -1); // print max TBL
    #endif
    MEMPHY_dump(proc->mram);
    #endif

    return val;
  }
```

Hàm tiến hành đọc dữ liệu của 1 tiến trình nào đó trong Page Table nếu có bằng cách gọi hàm __read

```
int __read(struct pcb_t *caller, int vmaid, int rgid, int offset, BYTE *data)
{
    struct vm_rg_struct *currg = get_symrg_byid(caller->mm, rgid);

    struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);

    if (currg == NULL || cur_vma == NULL) /* Invalid memory identify */
    return -1;

    if (currg->rg_start + offset > currg->rg_end) /* Invalid offset*/
    return -1;
    pg_getval(caller->mm, currg->rg_start + offset, data, caller);

    return 0;
  }
```

Bắt đầu với việc lấy thông tin về vùng nhớ ảo được chỉ định bởi rgid và vmaid. Tiến hành kiểm tra xem 2 con trỏ này có hợp lệ không, nếu 1 trong 2 con trỏ này là null thì sẽ trả về -1 báo lỗi. Trường hợp 2 con trỏ này hợp lệ, tiến hành kiểm tra xem offset có phải là vị trí có tồn tại trong vùng destination này không. Sau khi đã hoàn tất việc kiểm tra, lấy ra giá trị tại vị trí đó bằng hàm pg_getval và sau đó trả về 0 báo hiệu đã thực hiện hàm thành công.

3.1.3.d Hàm pgwrite

Ở trong file *mm-vm.c*, đây là hàm được gọi để thực hiện việc ghi data vào Page Table nếu như không tìm thấy data đó ở trong Translation Lookaside Buffer (sẽ được đề cập ở mục 3.2.1). Hàm này được mô tả như sau:

```
int pgwrite(
struct pcb_t *proc,   // Process executing the instruction
BYTE data,            // Data to be wrttien into memory
uint32_t destination, // Index of destination register
uint32_t offset)
{
  int val = __write(proc, 0, destination, offset, data);
    #ifdef IODUMP
    printf("write region=%d offset=%d value=%d\n", destination, offset, data);
    #ifdef PAGETBL_DUMP
    print_pgtbl(proc, 0, -1); // print max TBL
    #endif
    MEMPHY_dump(proc->mram);
    #endif

    return val;
  }
```

Hàm này có các biến như sau: process cần được thực thi, dữ liệu cần được ghi (data), vị trí cần được ghi (destination) và vị trí offset trong destination đó (offset). Ta sử dụng hàm __write để nhập thông tin vào region với vị trí offset. Sau đó sử dụng câu lệnh để in ra màn hình các giá trị mà ta cần write tại câu lệnh *IODUMP*. Cuối cùbng gọi *print_pgtbl* để in ra toàn bộ thông tin của page table đang có và *MEMPHY_dump* để in ra thông tin bộ nhớ vật lý

```
int __write(struct pcb_t *caller, int vmaid, int rgid, int offset, BYTE value)
{
    struct vm_rg_struct *currg = get_symrg_byid(caller->mm, rgid);

    struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);

    if (currg == NULL || cur_vma == NULL) /* Invalid memory identify */
    return -1;
    if (currg->rg_start + offset > currg->rg_end) /* Invalid offset*/
    return -1;
    pg_setval(caller->mm, currg->rg_start + offset, value, caller);

    return 0;
  }
```

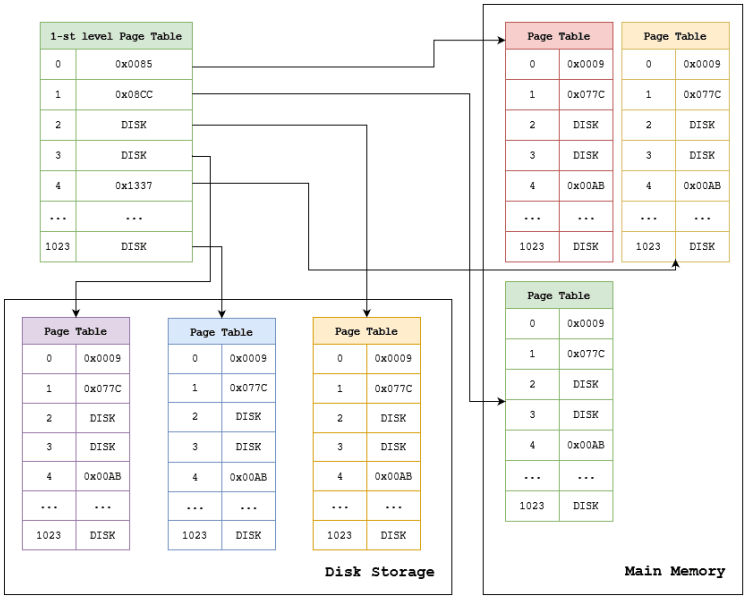
Bắt đầu với việc lấy thông tin về vùng nhớ ảo được chỉ định bởi rgid và vmaid. Tiến hành kiểm tra xem 2 con trỏ này có hợp lệ không, nếu 1 trong 2 con trỏ này là null thì sẽ trả về -1 báo lỗi. Trường hợp 2 con trỏ này hợp lệ, tiến hành kiểm tra xem offset có phải là vị trí có tồn tại trong vùng destination này không. Sau khi đã hoàn tất việc kiểm tra, lấy ra giá trị tại vị trí đó bằng hàm pg_setval và sau đó trả về 0 báo hiệu đã thực hiện hàm thành công.

Câu hỏi:

What will happen if we divide the address to more than 2-levels in the paging memory management system? *Điều gì sẽ xảy ra nếu chúng ta chia địa chỉ thành mô hình hơn 2-levels trong hệ thống quản lý phân trang*

Trả lời:

Ta cần hiểu trước là Multi-level paging (Phân trang đa cấp) là 1 dạng Phân trang gồm 2 hoặc nhiều cấp độ của Page Table trong 1 cái hệ thống cấp bậc đó. Có thể tham khảo hình bên dưới đây:



Mô hình trên chỉ rõ cho ta thấy Page Table của chúng ta chia làm 2 cấp độ: cấp độ đầu tiên sẽ chứa địa chỉ của nhiều page table khác. Trong khi cấp độ thứ 2 của Page table chứa dữ liệu cần để truy xuất. Điều này sẽ giúp cho các tiến trình được CPU yêu cầu nhiều sẽ luôn ở trong Main Memory giúp tối ưu hiệu năng hoạt động cho máy tính. Câu hỏi trên yêu cầu ta chỉ ra những thay đổi khi mô hình này hơn cấp độ 2, nó có những ưu điểm và nhược điểm như sau:

- Cung cấp thêm không gian bộ nhớ: Một trong những nhiệm vụ chính yếu nhất là để ta có thể truy cập được nhiều hơn số lượng lớn địa chỉ vật lý. Với nhiều cấp độ hơn, ta có thể chứa được nhiều địa chỉ hơn, truy cập được nhiều dữ liệu hơn.

- Giảm phân mảnh nội: Với nhiều level hơn, ta có thể tạo ra nhiều page table với kích thước nhỏ hơn, giúp giảm thiểu việc phân mảnh nội khi mà trong bộ nhớ của ta không có đủ kích thước để chứa dữ liệu mặc dù bộ nhớ vẫn còn ô trống.

Bên cạnh những ưu điểm đã nêu trên, việc phân nhiều hơn 2 cấp độ cũng mang lại 1 số ít bất lợi:

- Tăng chi phí để dịch địa chỉ: Nhiều page table hơn có nghĩa là ta cần phải truy cập đến nhiều các

Page Table này nhiều hơn để tìm kiếm dữ liệu thích hợp, việc này dẫn đến gia tăng chi phí.

- Phức tạp trong việc implement: Có quá nhiều cấp độ sẽ khiến cho việc quản lý và implement trở nên khó khăn hơn rất nhiều.

Câu hỏi:

In this simple OS, we implement a design of multiple memory segments or memory areas in source code declaration. What is the advantage of the proposed design of multiple segments? *Trong hệ điều hành này, chúng ta đã thiết kế nhiều memory segments và memory areas trong code. Đâu là những ưu điểm cho việc thiết kế này?*

Trả lời:

Trong hệ điều hành này, việc thiết kế nhiều memory segments và memory areas mang lại những lợi ích rất đáng kể: Khi chương trình nạp vào 1 đoạn code, thay vì đoạn code đó sẽ được phân thành nhiều phần nhỏ, sau đó được chia ra tại các vị trí bất kì trong bộ nhớ thứ cấp của chúng ta thì việc thiết kế để đoạn code đó trở thành nhiều segments là vô cùng hợp lý. Điều này giúp ích cho lập trình viên khi họ có thể xem và hiểu được đoạn code đó vì nó được chia liền mạch trong bộ nhớ (thay vì là chia nhỏ như kia). Bên cạnh đó, nó còn mang nhiều lợi ích như sau:

- Giảm thiểu công việc của hệ thống quản lý bộ nhớ: Mỗi segments, vùng nhớ đều có riêng cho mình 1 hệ thống cấp phát và huỷ cấp phát trong chính segment đó, điều này giúp cho hệ thống quản lý bộ nhớ không cần phải can thiệp quá nhiều vào công việc của từng segment và vùng nhớ đó.

- Tăng tính bảo mật: Mỗi segment, vùng nhớ riêng biệt có các quyền hạn khác nhau, tăng tính bảo mật cho toàn hệ thống. Ví dụ như: code segment thì ta có thể cấp quyền read-only, trong khi các phần như data-segment, những phần liên quan đến dữ liệu thì có quyền read-write.

Câu hỏi:

What is the advantage and disadvantage of segmentation with paging? *Kể ra những ưu điểm và nhược điểm của segmentation with paging*

Trả lời:

Segmentation with Paging là 1 kỹ thuật quản lý bộ nhớ kết hợp cả "segmentation" và "Paging" lại. Trong segmentation, đoạn code được chia thành các phân đoạn logic, như đoạn mã code (code segment), phân đoạn dữ liệu (data segment), stack segment... Các phân đoạn này có thể được rải rác, không liên tục trong bộ nhớ. Trong khi paging là kỹ thuật chia bộ nhớ ảo và bộ nhớ vật lý thành các trang có kích thước cố định đã được đề cập ở mục trên. Vì thế, ta có thể kể đến các ưu điểm rõ rệt khi kết hợp cả 2 kỹ thuật này lại:

- Tính bảo mật: Mỗi segmentation được cấp cho 1 phân đoạn với các quyền khác nhau (read, write,

execute), việc này sẽ giúp cho segmentation này không chạm đến dữ liệu của segmentation khác

- Chia sẻ: Segmentation cho phép các dữ liệu được chia sẽ lẫn nhau, giúp giảm bộ nhớ khi truy cập và tăng hiệu năng máy tính
- Kết hợp với Paging sẽ giúp cho Segmentation liên tục, liền mạch: Giúp ích cho các lập trình viên dễ dàng đọc và hiểu code hơn.

Bên cạnh những ưu điểm đã được liệt kê bên trên, segmentation with paging cũng có 1 số ít nhược điểm mà ta cần lưu ý:

- Độ phức tạp: Việc triển khai Segmentation with Paging đòi hỏi sự quản lý phức tạp hơn, yêu cầu cần phải có 1 thuật toán phù hợp hơn.
- Phân mảnh nội: Việc xuất hiện phân mảnh nội vẫn có trong kỹ thuật này.
- Việc chuyển đổi (Translation) xảy ra thường xuyên hơn nên sẽ gia tăng thời gian truy cập bộ nhớ.

3.2 Translation Lookaside Buffer

3.2.1 Translation Lookaside Buffer là gì?

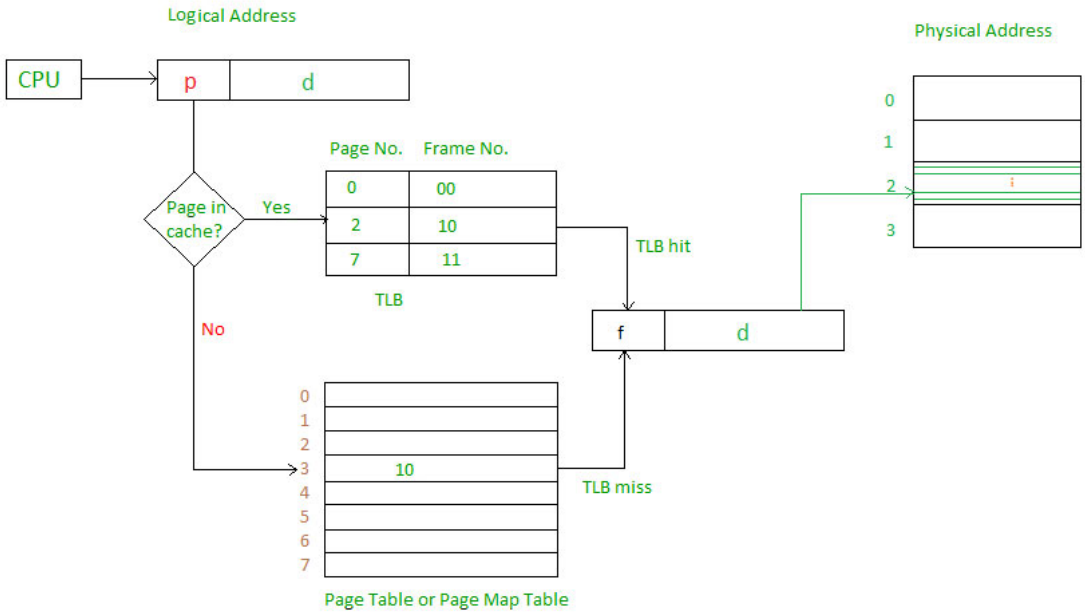
Chúng ta đã được đề cập ở trên về Paging, mỗi Process đều có cho mình 1 Page Table. Câu hỏi đặt ra là, nên đặt Page Table ở đâu để thời gian tổng thể được giảm xuống. Ý tưởng ban đầu có lẽ là lưu các Page Table Entry trong thanh ghi, vốn là không khả thi lắm vì kích thước của các thanh ghi quá nhỏ so với kích thước của các process. Thế là chúng ta có 1 ý tưởng khác, đó là sử dụng 1 cái cache tốc độ cao để lưu page table được gọi là Translation Lookaside Buffer (TLB). TLB là 1 bộ nhớ đệm đặc biệt dùng để lưu những Page Table Entry nào được sử dụng nhiều nhất. Khi CPU cung cấp 1 địa chỉ nào đó, nếu page table entry của địa chỉ đó hiện có trong TLB thì sẽ gọi là TLB hit, và ngay lập tức Frame number sẽ được trả về. Nếu chưa có page table entry đó trong TLB, TLB sẽ tìm kiếm trong Page Table, và TLB miss xảy ra, lúc này PTE mới đó sẽ được cập nhật vào TLB. Hình sau mô tả chi tiết cách hoạt động của TLB.

3.2.2 Translation Lookaside Buffer trong Bài tập lớn

Bài tập lớn yêu cầu chúng ta thiết kế phương pháp mapping trong TLB, kiểm tra xem coi Process đó có trong Cache của hệ điều hành chưa. Để làm việc này thì nhóm chúng em đã tiến hành khởi tạo 1 struct mới, là struct tlb cache, struct này có chức năng để lưu dữ liệu của các Page Table Entry vào khi Process yêu cầu nó.

```
struct tlb_entry tlb_cache[TLB_SIZE];
```

Với tlb_entry được khai báo ở file os_mm.h




```
struct tlb_entry {
int pid;
int pgnum;
int frame;
int valid; //Valid = 1 when entry contain page, otherwise valid = -1
int time;
};
```

3.2.2.a Hàm tlb_cache_read

Hàm này thực hiện việc tìm kiếm trong bộ nhớ cache TLB có đang có pid và pagenum mà cần tìm không.

```
int tlb_cache_read(struct memphy_struct * mp, int pid, int pgnum,struct tlb_entry*
                                            tlb_cache) // type
{
/* TODO: the identify info is mapped to
*      cache line by employing:
*      direct mapped, associated mapping etc.
*/
for (int i = 0; i < TLB_SIZE/PAGE_SIZE; ++i) {
if (tlb_cache[i].valid==1 && tlb_cache[i].pid == pid && tlb_cache[i].pgnum == pgnum) {
// TLB cache hit, return the physical address
return tlb_cache[i].frame ;
}

}
return -1;
}
```

Ban đầu hàm duyệt qua từng phần tử trong TLB Cache, và kiểm tra xem nếu tlb_cache[i].pid = với pid

đang cần tìm và tlb_cache[i].pgnum = với pgnum thì sẽ trả về frame number mà nó đang nắm giữ (gọi là TLB hit). Trường hợp không tìm thấy thì trả về -1.

3.2.2.b Hàm tlb_cache_write

Hàm này thiết kế để ghi dữ liệu vào cache TLB khi page đó chưa nằm trong cache TLB.

```
  int tlb_cache_write(struct memphy_struct *mp, int pid, int pgnum, BYTE value,int frame,
                                              int swapType,struct tlb_entry* tlb_cache)
{
   /* TODO: the identify info is mapped to
    *      cache line by employing:
    *      direct mapped, associated mapping etc.
    */
   /*
   swapType 1 == add
   swapType 2 == free
   */
   if(swapType==1){
      for (int i = 0; i < TLB_SIZE/PAGE_SIZE; ++i) {
         if (tlb_cache[i].valid==-1) {
               tlb_cache[i].pid = pid;
               tlb_cache[i].pgnum = pgnum;
               tlb_cache[i].frame = frame;
               tlb_cache[i].valid = 1;
               tlb_cache[i].time = ++timeFIFO;

               return 1;
               break;
         }
      }
      int minTime=99999999;
            int minPos=-1;
            int i;
            for(i=0;i<TLB_SIZE/PAGE_SIZE;++i){
               if(tlb_cache[i].time<minTime) {
                  minTime=tlb_cache[i].time;
                  minPos=i;
               }
            }
            if(minPos==-1) printf("Loi o delete tlb_cache_write");
            tlb_cache[i].pid = pid;
            tlb_cache[i].pgnum = pgnum;
            tlb_cache[i].frame = frame;
            tlb_cache[i].valid = 1;
            tlb_cache[i].time = ++timeFIFO;
            return 1;
   }
    else{
      for (int i = 0; i < TLB_SIZE/PAGE_SIZE; ++i) {
        if (tlb_cache[i].valid==1 && tlb_cache[i].pid == pid && tlb_cache[i].pgnum ==
                                                      pgnum) {
```

```
            tlb_cache[i].valid = -1;
            return 1;
            break;
        }
    }
    printf("loi o free tlbcache in tlb_cache_write");
    }

   return 0;
}
```

Hàm này sẽ phụ thuộc vào chỉ số swapType mà thực hiện, swapType 1 tức là thêm 1 dữ liệu mới vào TLB cache, trong khi swapType 2 có nghĩa là giải phóng dữ liệu đó ra khỏi TLB cache. Đối với trường hợp swapType = 1, ban đầu hệ thống sẽ duyệt qua từng tlb Cache, nếu tìm thấy TLB cache chưa có giá trị nào thì ta sẽ tiến hành ghi giá trị đó vào vị trí TLB cache đó. Trong trường hợp không có vị trí nào đang trống. Ta sẽ thực hiện tìm kiếm victim theo chính sách FIFO, lúc nào thuộc tính time sẽ được sử dụng. Đối với swapType = 2, hàm này sẽ đánh dấu vị trí cần giải phóng là -1 cho biết rằng mục này có thể được thêm dữ liệu vào. Nếu hàm không tìm thấy mục tương ứng trong swapType 2, hàm sẽ trả về 0 và in ra dòng báo lỗi.

3.2.3 Hàm tlballoc

Trong mục *cpu-tlb.c*, hàm này được thiết kế để thực hiện việc cấp phát bộ nhớ.

```
int tlballoc(struct pcb_t *proc, uint32_t size, uint32_t reg_index)
{
    // printf("tlb_alloc\n");
    int addr, val;
    /* By default using vmaid = 0 */
    printf("Alloc at region:%d size:%d in process %d\n", reg_index, size, proc->pid);
    val = __alloc(proc, 0, reg_index, size, &addr);
    /* TODO update TLB CACHED frame num of the new allocated page(s)*/
    /* by using tlb_cache_read()/tlb_cache_write()*/
    struct vm_rg_struct *rgnode = get_symrg_byid(proc->mm, reg_index);
    printf("Finish alloc region=%d size=%d adrress form %ld to %ld at process %d\n",
                                        reg_index, size, rgnode->rg_start, rgnode->
                                        rg_end, proc->pid);
    return val;
  }
```

Gán giá trị *val* là giá trị trả về của hàm alloc đã được đề cập ở trên, sau khi alloc xong thì in ra dòng thông báo cũng như trả về giá trị. Hàm sẽ không thực hiện truy xuất vào TLB do chị cấp phát ở bộ nhớ ảo, không thực hiện truy xuất bộ nhớ vật lý *val*.

3.2.4 Hàm tlbfree_data

Ở trong mục *cpu-tlb.c*, hàm này được thiết kế để giải phóng bộ nhớ đã được cấp phát, cũng như cập nhật lại thông tin trong TLB khi bộ nhớ nào đó được giải phóng.

```
int tlbfree_data(struct pcb_t *proc, uint32_t reg_index)
{
    // printf("tlb_free\n");
    printf("Free at region=%d at process %d\n", reg_index, proc->pid);
    struct vm_rg_struct *rgnode = get_symrg_byid(proc->mm, reg_index);
    int addr = rgnode->rg_start;
    if (addr == -1)
    {
        printf("Bo nho chua duoc cap phat\n");
        return -1;
      }
    int pgn = PAGING_PGN(addr);
    __free(proc, 0, reg_index);

    /* TODO update TLB CACHED frame num of freed page(s)*/
    /* by using tlb_cache_read()/tlb_cache_write()*/
    if (tlb_cache_read(proc->mram, proc->pid, pgn, proc->tlb_cache) != -1)
    {
        printf("co trong tlb\n");
        tlb_cache_write(proc->mram, proc->pid, pgn, 0, 0, 0, proc->tlb_cache);
      }
    printf("Finish free region=%d adrress form %ld to %ld at process %d\n", reg_index,
                                        rgnode->rg_start, rgnode->rg_end, proc->pid)
                                        ;
    rgnode->rg_start = -1;
    return 0;
  }
```

Ban đầu, tiến hành gọi hàm get_symrg_byid để lấy thông tin về vùng nhớ cần giải phóng từ cấu trúc quản lý bộ nhớ của tiến trình. Sau đó, lấy ra địa chỉ của vùng nhớ đó. Bắt đầu tiến hành kiểm tra các điều kiện, nếu địa chỉ này = -1, tức là vùng này chưa được cấp phát, trả về giá trị -1, trường hợp còn lại thì tiến hành gọi hàm __free để giải phóng vùng nhớ đó. Trong quá trình đó thì kiểm tra xem thông tin của page này đã trong TLB chưa, nếu đã trong TLB thì thực hiện giải phóng vị trí đó trong TLB. Ở bước này ta chỉ đơn giản đổi biến valid ở entry TLB lưu vùng nhớ này thành -1, không cần phải xóa ở bộ nhớ vật lý vì khi vùng nhớ vật lý nhận giá trị mới ta sẽ tiến hành ghi đè. Sau cùng chỉ ra vùng nhớ đã được giải phóng.

3.2.5 Hàm tlbread

Nằm trong mục *cpu-tlb.c*, hàm này có công dụng đọc dữ liệu từ 1 vùng nhớ và xử lý các công việc của TLB (kiểm tra xem TLB miss hay TLB hit)

```
int tlbread(struct pcb_t * proc, uint32_t source,
           uint32_t offset,  uint32_t destination,int* countTLB, int* countTLBMiss)
{
  BYTE data;
  int frmnum = -1;
  /* TODO retrieve TLB CACHED frame num of accessing page(s)*/
  /* by using tlb_cache_read()/tlb_cache_write()*/
  /* frmnum is return value of tlb_cache_read/write value*/
```

```
  struct vm_rg_struct *rgnode = get_symrg_byid(proc->mm, source);
  if(rgnode->rg_start==-1) {
    printf("\tRegion %d chua duoc cap phat\n",source);fflush(stdout);
    return -1;
  }
  int sourceAddress= rgnode->rg_start + offset;;
  int pgn = PAGING_PGN(sourceAddress);
  int off = PAGING_OFFST(sourceAddress);
  frmnum= tlb_cache_read(proc->mram,proc->pid,pgn,proc->tlb_cache);

#ifdef IODUMP
  if (frmnum >= 0){
    printf("\tTLB hit at read region=%d offset=%d\n", source, offset);fflush(stdout);
    *countTLB=*countTLB+1;
  }
  else{
    printf("\tTLB miss at read region=%d offset=%d\n",source, offset);fflush(stdout);
    *countTLB=*countTLB+1;
    *countTLBMiss=*countTLBMiss+1;
  }
  int val;
  if(frmnum>=0){
   int phyaddr = (frmnum << PAGING_ADDR_FPN_LOBIT) + off;
  TLBMEMPHY_read(proc->mram,phyaddr,&data);
  }
  else{
  val = __read(proc, 0, source, offset, &data);
   uint32_t pte = proc->mm->pgd[pgn];
   frmnum=PAGING_FPN(pte);

    tlb_cache_write(proc->mram,proc->pid,pgn,0,frmnum,1,proc->tlb_cache);
  }
  destination = (uint32_t) data;
#ifdef IODUMP
 printf("\tread region=%d offset=%d value=%d process=%d\n", source, offset, data,proc->
                                                pid);fflush(stdout);
#endif

  /* TODO update TLB CACHED with frame num of recent accessing page(s)*/
  /* by using tlb_cache_read()/tlb_cache_write()*/

#ifdef PAGETBL_DUMP
  print_pgtbl(proc, 0, -1); //print max TBL
#endif
  MEMPHY_dump(proc->mram);
#endif

  return val;
}
```

Ban đầu hàm kiểm tra xem Region đang được gọi đến đã được cấp phát hay chưa (nếu giá trị là -1 thì tức là chưa được cấp phát). Trường hợp region đã được cấp phát thì tiến hành tính toán số trang của địa

chỉ này, gọi hàm tlb_cache_read để xem coi Page này đã nằm trong TLB hay chưa. Nếu đã nằm trong TLB thì giá trị trả về sẽ >= 0, lúc này sẽ báo rằng TLB hit tại region đó (và tiến hành việc giải mã giá trị FrameNum sang địa chỉ vật lý), trường hợp ngược lại là miss thì ta tiến hành đọc dữ liệu theo cơ chế **Paging** sau đó cập nhật trang vừa đọc vào TLB bằng hàm tlb_cache_write. Sau khi tiến hành kiểm tra xong,in ra giá trị đọc được nếu không xảy ra lỗi. in ra thông tin Page Table và bộ nhớ vật lý hiện tại.

3.2.6 Hàm tlbwrite

Trong mục *cpu-tlb.c*, hàm này được thiết kế để kiểm tra xem TLB miss hoặc hit và tiến hành ghi dữ liệu vào trong bộ nhớ vật lý.

```
  int tlbwrite(struct pcb_t * proc, BYTE data,
              uint32_t destination, uint32_t offset,int* countTLB, int* countTLBMiss)
{
  //printf("tlb_write\n");
  int val;
  int frmnum = -1;

  /* TODO retrieve TLB CACHED frame num of accessing page(s))*/
  /* by using tlb_cache_read()/tlb_cache_write()
  frmnum is return value of tlb_cache_read/write value*/
 struct vm_rg_struct *rgnode = get_symrg_byid(proc->mm, destination);
  int sourceAddress= rgnode->rg_start;
  if(sourceAddress==-1) {
    printf("\tRegion %d chua duoc cap phat\n",destination);fflush(stdout);
    return -1;
  }
  int pgn = PAGING_PGN(sourceAddress);
  int offAdd=sourceAddress+offset;
  int off = PAGING_OFFST(offAdd);
  frmnum= tlb_cache_read(proc->mram,proc->pid,pgn,proc->tlb_cache);

#ifdef IODUMP
  if (frmnum >= 0){
    printf("\tTLB hit at write region=%d offset=%d value=%d\n",
            destination, offset, data);fflush(stdout);
    *countTLB=*countTLB+1;
  }
  else{
    printf("\tTLB miss at write region=%d offset=%d value=%d\n",
            destination, offset, data);fflush(stdout);
    *countTLB=*countTLB+1;
    *countTLBMiss=*countTLBMiss+1;
  }
  if(frmnum>=0){
    int phyaddr = (frmnum << PAGING_ADDR_FPN_LOBIT) + off;
    TLBMEMPHY_write(proc->mram,phyaddr,data);
  }
  else{
  val = __write(proc, 0, destination, offset, data);
  uint32_t pte = proc->mm->pgd[pgn];
  frmnum=PAGING_FPN(pte);
```

```
    tlb_cache_write(proc->mram,proc->pid,pgn,0,frmnum,1,proc->tlb_cache);

  }
  /* TODO update TLB CACHED with frame num of recent accessing page(s)*/
  /* by using tlb_cache_read()/tlb_cache_write()*/
  #ifdef IODUMP
  printf("\twrite region=%d offset=%d value=%d process=%d\n", destination, offset, data,
                                                  proc->pid);fflush(stdout);
  #endif
#ifdef PAGETBL_DUMP
  print_pgtbl(proc, 0, -1); //print max TBL
#endif
  MEMPHY_dump(proc->mram);
#endif
  return val;
}
```

Ban đầu hàm kiểm tra xem vùng nhớ này đã được cấp phát chưa, nếu đã được cấp phát rồi hàm tiến hành xác định số trang của vùng nhớ và đọc thông tin từ TLB bằng việc gọi hàm *tlb_cache_read*, nếu đã có trong TLB thì giá trị frmnum sẽ lớn hơn bằng 0, lúc này sẽ báo TLB hit. Tiếp sau đó tiến hành việc đưa dữ liệu ở FRMNUM này đến địa chỉ vật lý. Trường hợp ngược lại thì ta sẽ tiến hành ghi dữ liệu bằng có chế **Paging** và cập nhật dữ liệu của Page này vào TLB bằng hàm *tlb_cache_write* và in ra dòng write tại region, sau đó in ra thông tin Page Table và bộ nhớ vật lý hiện tại.

Câu hỏi:

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?

Điều gì sẽ xảy ra nếu các lõi CPU trong hệ thống đa nhân có thể chạy trong ngữ cảnh khác nhau, và mỗi lõi có MMU và TLB riêng? Trong các kiến trúc CPU hiện đại, TLB 2 cấp bậc đã trở nên phổ biến, ảnh hưởng của cấu hình phần cứng bộ nhớ mới này đối với các chiến lược "dịch" của chúng ta sẽ như thế nào?

Trả lời:

CPU nhiều core thực hiện nhiều tác vụ khác nhau và mỗi core mang trong mình Memory Managment Unit (MMU) và TLB (Translation Lookaside Buffer) riêng sẽ mang lại nhiều ưu điểm. Mỗi nhân của từng CPU sẽ có thể chạy tác vụ riêng biệt khác nhau, với mỗi page table khác nhau. Mỗi core đều có riêng cho mình MMU (nó không phải là 1 unit tách biệt khác, nó vẫn là 1 phần của nhân). Đối với những dữ liệu được chia sẻ chung, những địa chỉ của dữ liệu đó được chỉ rõ trong bộ nhớ vật lý của MMU. Từ những đặc điểm được nêu trên, ta sẽ thấy rõ những ưu điểm khi 1 CPU đa nhân có thể chạy tác vụ khác nhau và mỗi CPU có riêng cho mình MMU và TLB là:

- Cải thiện khả năng chạy song song: Mỗi core đều có thể chạy từng tác vụ riêng biệt nên có thể phân chia công việc hợp lý, cải thiện tốc độ, hiệu năng của máy tính.

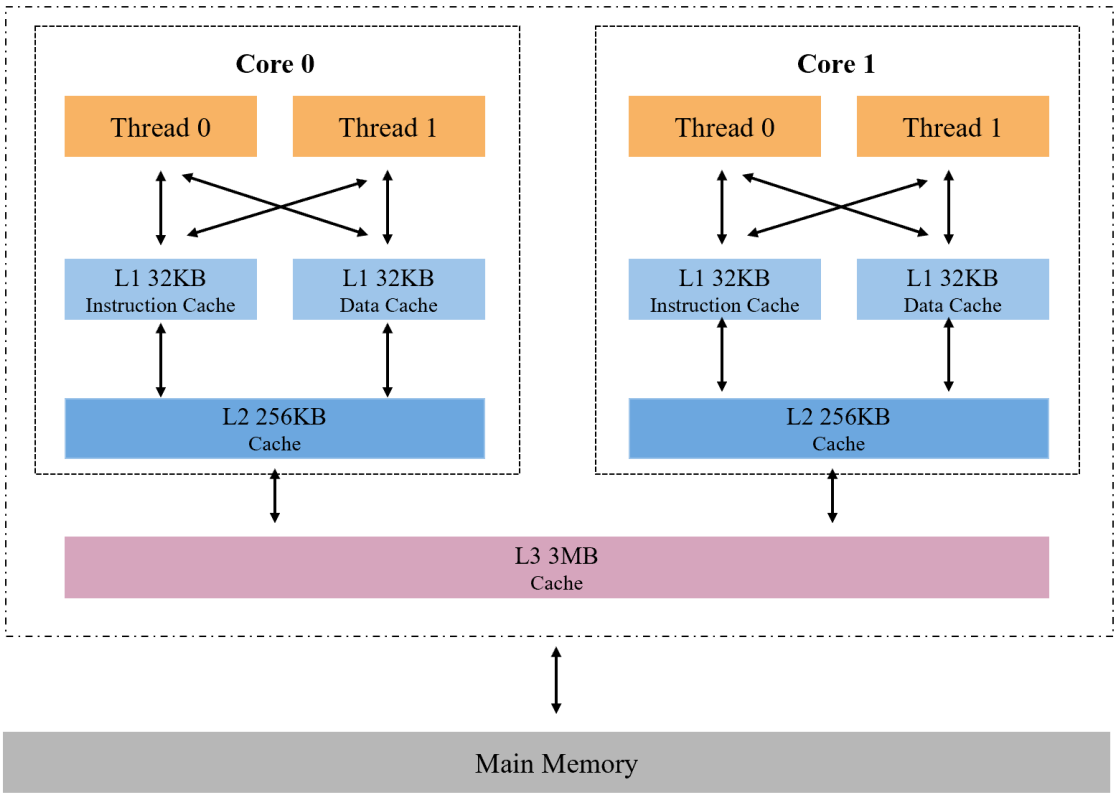
- Với từng tác vụ khác nhau: Các core sẽ không can thiệp vào bộ nhớ của nhau giúp cải thiện tính bảo mật, và nếu 1 tác vụ bị lỗi thì không ảnh hưởng đến tác vụ khác.

- Mỗi nhân đều có riêng cho mình MMU và TLB nên việc đợi MMU và TLB để thực hiện công việc của mình là không cần thiết.

Có những điểm mạnh đã kể trên, nhưng đi cùng với đó là những điều cần xem xét khi hiện thực:

- Hệ điều hành cần phải kiểm soát, lập lịch cho từng nhân trong CPU và kiểm soát cấp phát tài nguyên hợp lý.

- Có nhiều dữ liệu cần phải được chia sẻ chung, việc kiểm soát những dữ liệu chung này có thể dẫn đến overhead. Như hình dưới đây, L3 là dữ liệu chung được chia sẻ.



Trong các CPU hiện đại ngày nay, mô hình 2-level TLB được sử dụng vô cùng rộng rãi. Vậy, điều đó có những tác động như thế nào đến mô hình translation của chúng ta? Câu trả lời là nó tác động đáng kể đến việc memory translation schemes của chúng ta. Nó cung cấp những lợi ích như sau:

- Việc tìm kiếm ra địa chỉ trong bộ nhớ vật lý sẽ nhanh hơn: 2-level TLB được thiết kế sao cho level đầu tiên sẽ nhỏ hơn nhưng nhanh hơn trong khi level thứ 2 sẽ lưu giữ nhiều Page Table Entry hơn. Đối với những Page Table Entry được yêu cầu (request) nhiều, ta sẽ lưu nó ở level 1, từ đó trích xuất ra địa chỉ 1 cách nhanh chóng hơn.

- Đã được đề cập ở trên, level 2 sẽ chứa nhiều Page Table Entry hơn, từ đó ta có thể giảm số lần TLB miss.

Những tác động mà 2-level mang lại đều là lợi ích và không làm thay đổi quá nhiều Memory Translation Schemes của chúng ta: Nó vẫn mang những nội dung chuyển từ bộ nhớ ảo đến bộ nhớ thật như mô hình cũ.

4 Put It All Together

Câu hỏi:

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. Note: You need to run two versions of your simple OS: the program with/without synchronization, then observe their performance based on demo results and explain their differences.
Điều gì sẽ xảy ra nếu như hệ thống đồng bộ không được xử lý trong hệ điều hành của bạn? Mô phỏng bằng ví dụ nếu có

Trả lời:

Nếu như không có hệ thống đồng bộ trong Hệ điều hành của chúng ta, sẽ có kha khá các vấn đề xảy ra như sau.

- Race Condition: Không có cơ chế đồng bộ như semaphores hoặc tương tự sẽ dẫn đến nhiều process khác nhau cùng truy cập 1 nguồn tài nguyên nào đó cùng lúc, gây ra sự không lường trước được của tài nguyên đó. (giá trị của tài nguyên đó ta không theo kết quả ta mong muốn)

- Deadlocks: Giả sử như có 1 biến A cần phải có giá trị bao nhiêu đó thì Process B mới thực hiện được. Tuy nhiên, biến A này đều đang được cả Process A và C sử dụng, dẫn đến kết quả của biến A này sẽ không theo 1 kết quả ta mong muốn, từ đó Process B sẽ luôn luôn chờ biến A và không thể thực hiện, vô tình thì Process C lại đang chờ 1 biến B đang nằm trong Process B, các process chờ đợi lẫn nhau mà không thể thực hiện được.

- Data Corruption: Có quá nhiều lượt truy cập đồng thời trong 1 dữ liệu sẽ khiến cho dữ liệu đó bị lỗi. Ví dụ nhiều tiến trình cũng nhau write 1 giá trị biến, biến đó sẽ không biết nên nhận giá trị nào.

Minh hoạ về vấn đề đồng bộ của chương trình đơn giản, nhóm chúng em đang có 1 chương trình: Process A và B, cả 2 đều đang cố gắng để tăng giá trị của biến *counter*. Chúng em sẽ cùng viết chương trình này theo 2 kiểu: *đồng bộ* và *bất đồng bộ*, cùng nhau quan sát kết quả (kết quả mà chúng em mong muốn là 200000000). Dưới đây là Code được viết theo kiểu *đồng bộ*:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NUM_THREADS 2
#define NUM_INCREMENTS 1000000

int counter = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *increment(void *thread_id)
{
    for (int i = 0; i < NUM_INCREMENTS; ++i)
```

```
    {
        pthread_mutex_lock(&mutex);
        counter++;
        pthread_mutex_unlock(&mutex);
      }
    pthread_exit(NULL);
  }

int main()
{
pthread_t threads[NUM_THREADS];
int rc;
long t;
for (t = 0; t < NUM_THREADS; t++)
{
rc = pthread_create(&threads[t], NULL, increment, (void *)t);
if (rc)
{
    printf("Loi:  %d\n", rc);
    exit(-1);
  }
}

for (t = 0; t < NUM_THREADS; t++)
{
    pthread_join(threads[t], NULL);
  }

printf("Counter value %d\n", counter);

pthread_exit(NULL);
}
```

Kết quả mà nhóm chúng em nhận được của chương trình đồng bộ trên:

Cơ chế đồng bộ của chúng em được thể hiện qua: pthread_mutex_t mutex, chúng em khởi tạo ban đầu 1 cái mutex và sau đó yêu cầu cần phải có lock của mutex thì mới có thể chạy vào và tăng biến counter. Khi có 1 thread đang thực thi trong hàm increment thì các threads khác không thể truy cập vào được. Trái ngược với đồng bộ, code ở dạng bất đồng bộ sẽ không hề có mutex, dẫn đến việc kết quả xảy ra là không mong muốn. Dưới đây là code của *bất đồng bộ*:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NUM_THREADS 2
#define NUM_INCREMENTS 1000000

int counter = 0;
```

```
void *increment(void *thread_id)
{
    for (int i = 0; i < NUM_INCREMENTS; ++i)
    {
        counter++;
      }
    pthread_exit(NULL);
  }

int main()
{
pthread_t threads[NUM_THREADS];
int rc;
long t;
for (t = 0; t < NUM_THREADS; t++)
{
rc = pthread_create(&threads[t], NULL, increment, (void *)t);
if (rc)
{
    printf("Loi: %d\n", rc);
    exit(-1);
  }
}

for (t = 0; t < NUM_THREADS; t++)
{
    pthread_join(threads[t], NULL);
  }

printf("Counter value: %d\n", counter);

pthread_exit(NULL);
}
```

Kết quả mà chúng em nhận được mỗi lúc là khác nhau:

5 Chạy thử Testcase

5.1 Tổng quan

Bọn em sẽ tiến hành thực thi để in ra thông tin, tình trạng của các TLB cũng như các Process, tiến hành mở Terminal và nhập đoạn lệnh:

```
$ make
```

Sau khi gõ đoạn lệnh xong, toàn bộ các file thực thi sẽ được chạy, tiếp sau đó bọn em sẽ sử dụng lệnh

```
./os [FILE INPUT]
```

Trong đó, [FILE INPUT] là tên của file INPUT mà ta muốn chạy thử, tụi em đã thống nhất đã chọn file: *sched_0* và *sched_1* cho phần CPU Scheduling và *os_0_mlq_paging* cho phần Memory.

5.2 CPU Scheduling

Sau khi mô tả giải thuật định thời, cần kiểm tra kết quả kiểm thử và giải thích kết quả. Với dung lượng kiểm thử định thời, báo cáo cung cấp 2 bài kiểm tra, ứng với file đầu vào có sẵn trong đề bài: `sched_0`, `sched_1`. Với $MAX\_PRIO = 12$.

Kiểm thử file *sched_0*

File input `sched` có nội dung như sau:

```
2   1   4
0   s0  8
4   s1  5
10  s2  3
14  s3  8
```

Bảng 2: Bảng tóm tắt nội dung các tiến trình được sử dụng

Tiến trình	**s0**	**s1**	**s2**	**s3**
Thuộc tính	11 13	6 8	4 10	7 15
	calc	calc	calc	calc
	calc	calc	calc	calc
	calc	calc	calc	calc
	calc	calc	calc	calc
	calc	calc	calc	calc
	calc	calc	calc	calc
	calc	calc	calc	calc
	calc	calc	calc	calc
	calc		calc	calc
	calc		calc	calc
	calc			calc
	calc			calc
	calc			calc
				calc
				calc

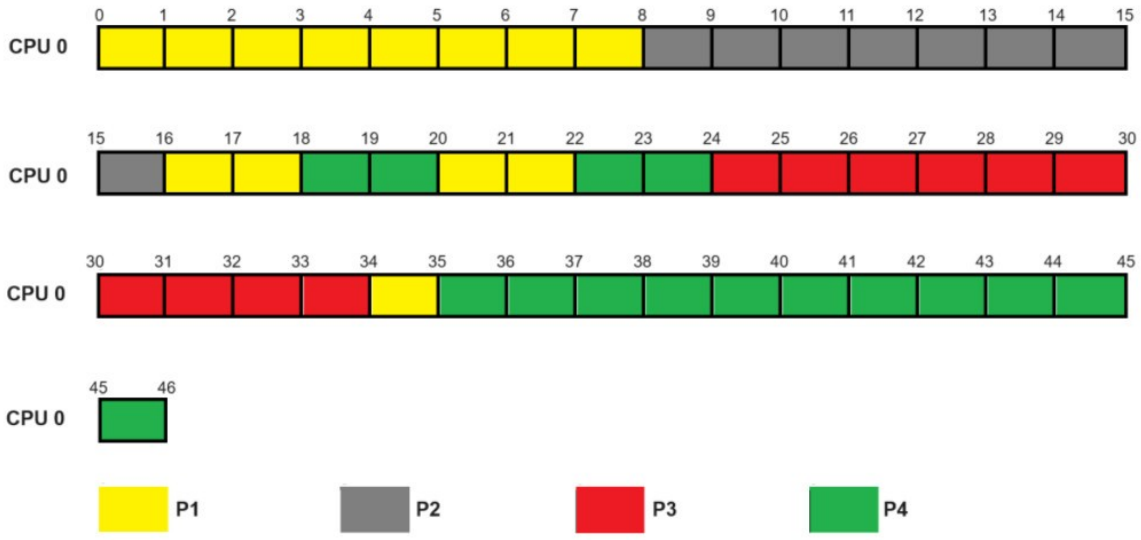
Kết quả đầu ra:

```
Time slot   0
ld_routine
        Loaded a process at input/proc/s0, PID: 1 PRIO: 8
        CPU 0: Dispatched process  1
Time slot   1
Time slot   2
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   3
Time slot   4
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
        Loaded a process at input/proc/s1, PID: 2 PRIO: 5
Time slot   5
Time slot   6
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   7
Time slot   8
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
Time slot   9
Time slot  10
        CPU 0: Put process  2 to run queue
        Loaded a process at input/proc/s2, PID: 3 PRIO: 3
```

```
        CPU 0: Dispatched process  2
Time slot  11
Time slot  12
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot  13
Time slot  14
        Loaded a process at input/proc/s3, PID: 4 PRIO: 8
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot  15
Time slot  16
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  1
Time slot  17
Time slot  18
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  4
Time slot  19
Time slot  20
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  1
Time slot  21
Time slot  22
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  4
Time slot  23
Time slot  24
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  3
Time slot  25
Time slot  26
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  3
Time slot  27
Time slot  28
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  3
Time slot  29
Time slot  30
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  3
Time slot  31
Time slot  32
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  3
Time slot  33
Time slot  34
        CPU 0: Processed  3 has finished
        CPU 0: Dispatched process  1
Time slot  35
        CPU 0: Processed  1 has finished
        CPU 0: Dispatched process  4
```

```
Time slot   36
Time slot   37
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  4
Time slot   38
Time slot   39
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  4
Time slot   40
Time slot   41
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  4
Time slot   42
Time slot   43
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  4
Time slot   44
Time slot   45
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  4
Time slot   46
        CPU 0: Processed  4 has finished
        CPU 0 stopped
```

Giản đồ Gantt:



Hình 3: Giản đồ Gantt mô tả kết quả đầu ra file *sched*.

Giải thích kết quả:

Khởi tạo slot cho các hàng đợi

mlq_ready_queue[0].slot = 12 - 0 = 12.

mlq_ready_queue[1].slot = 12 - 1 = 11.
mlq_ready_queue[2].slot = 12 - 2 = 10.
...
mlq_ready_queue[11].slot = 12 - 11 = 1.

- **Time 0** process 1 vào và có prio bằng 8 nên được đưa vào mlq_ready_queue[8]. Hàm get_mlq_proc() được gọi, lúc này sẽ duyệt từ hàng đợi có PRIO bằng 0. Process 1 thuộc mlq_ready_queue[8] được chọn để sử dụng CPU. Slot của mlq_ready_queue[8] lúc này là 4.

- **Time 4** process 2 vào và có prio bằng 5 nên được xếp vào hàng đợi mlq_ready_queue[5] nhưng hàng đợi mlq_ready_queue[8] vẫn chưa hết slot (slot = 2) và vẫn còn process trong đó nên process 2 phải đợi.

- **Time 8** lúc này hàng đợi mlq_ready_queue[8] đã hết slot (hết lượt sử dụng CPU) ta sẽ dùng duyệt qua từng hàng đợi bằng đầu từ hàng đợi mlq_ready_queue[8] với biến đếm i và $i = (i + 1)\%MAX\_PRIO$. Khi duyệt xong hàng đợi mlq_ready_queue[11] ($prio = MAX\_PRIO - 1$) thì tái khởi tạo lại giá trị biến slot cho các hàng đợi như ban đầu. Sau khi duyệt theo quy luật như vậy thì process 2 thuộc hàng đợi mlq_ready_queue[5] là process được chọn để sử dụng CPU. Slot của hàng đợi mlq_ready_queue[5] lúc này là 7.

- **Time 10** process 3 vào và có prio bằng 3 nên được xếp vào hàng đợi mlq_ready_queue[3] nhưng hàng đợi mlq_ready_queue[5] vẫn chưa hết slot (slot = 6) và vẫn còn process trong đó nên process 3 phải đợi.

- **Time 14** process 4 vào và có prio bằng 8 nên được xếp vào hàng đợi mlq_ready_queue[8] nhưng hàng đợi mlq_ready_queue[5] vẫn chưa hết slot (slot = 4) và vẫn còn process trong đó nên process 4 phải đợi.

- **Time 16** process 2 đã chạy xong và trong hàng đợi mlq_ready_queue[5] không còn process nào khác nên tiếp tục chạy thuật toán để tìm ra process để sử dụng CPU. Hàng đợi tiếp theo được chọn là mlq_ready_queue[8], trong hàng đợi này có process 1 và process 4, vì process 1 tới hàng đợi trước process 4 nên process 1 được chọn để sử dụng CPU. Slot của mlq_ready_queue[8] lúc này là 4 do được khởi tạo lại trước đó.

- **Time 16 tới Time 24** process 1 và process 4 thay phiên nhau sử dụng CPU theo cơ chế Round Robin

- **Time 24** lúc này hàng đợi mlq_ready_queue[8] đã hết slot. Hệ thống sẽ duyệt qua các hàng đợi để chọn ra process được sử dụng CPU, khi duyệt xong hàng đợi mlq_ready_queue[11] ($prio = MAX\_PRIO - 1$) thì tái khởi tạo lại giá trị biến slot cho các hàng đợi như ban đầu. Sau khi duyệt tới hàng đợi mlq_ready_queue[3] thì trong hàng đợi chỉ có một process 3 nên process 3 được sử dụng CPU. Slot của hàng đợi mlq_ready_queue[3] là 9.

- **Time 34** process 3 đã chạy xong và trong hàng đợi mlq_ready_queue[3] không còn process nào khác nên tiếp tục chạy thuật toán để tìm ra process để sử dụng CPU. Hàng đợi tiếp theo được chọn là mlq_ready_queue[8], trong hàng đợi này có process 1 và process 4, vì process 1 tới hàng đợi trước process 4 nên process 1 được chọn để sử dụng CPU. Slot của mlq_ready_queue[8] lúc này là 4 do được khởi tạo lại trước đó.

- **Time 35** process 1 đã chạy xong nhưng hàng đợi mlq_ready_queue[8] vẫn còn slot nên process 4 sẽ được chọn để sử dụng CPU.Slot của mlq_ready_queue[8] lúc này là 3.

- **Time 41** hàng đợi mlq_ready_queue[8] đã hết slot nên hệ thống sẽ chạy giải thuật để tìm ra hàng process được sử dụng CPU tiếp theo, khi duyệt xong hàng đợi mlq_ready_queue[11] ($prio = MAX\_PRIO - 1$) thì tái khởi tạo lại giá trị biến slot cho các hàng đợi như ban đầu.Sau khi duyệt tới hàng đợi mlq_ready_queue[8] thì trong hàng đợi chỉ có một process 4 nên process 4 được sử dụng CPU. Slot của hàng đợi mlq_ready_queue[8] là 4.

- **Time 46** process 4 đã chạy xong và sau khi kiểm tra thì nhận thấy không còn bất kì process nào trong tất cả hàng đợi nên CPU sẽ được dừng lại.

Kiểm thử file *sched_1*

Cho trước $MAX\_PRIO = 12$.

File input `sched_1` có nội dung như sau:

```
2 3 4
0 s0 8
6 s1 5
9 s2 3
12 s3 8
```

Kết quả đầu ra:

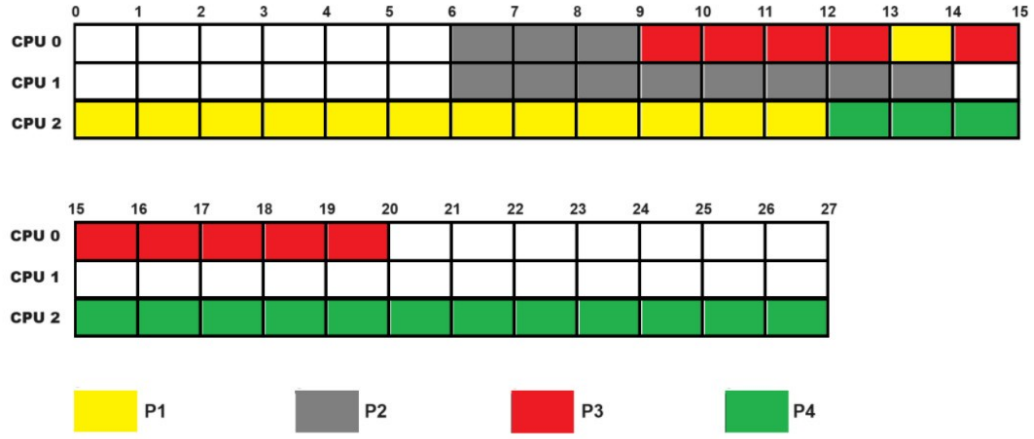
```
Time slot   0
ld_routine
        Loaded a process at input/proc/s0, PID: 1 PRIO: 8
        CPU 2: Dispatched process  1
Time slot   1
Time slot   2
        CPU 2: Put process  1 to run queue
        CPU 2: Dispatched process  1
Time slot   3
        CPU 2: Put process  1 to run queue
Time slot   4
        CPU 2: Dispatched process  1
Time slot   5
        CPU 2: Put process  1 to run queue
        CPU 2: Dispatched process  1
Time slot   6
        Loaded a process at input/proc/s1, PID: 2 PRIO: 5
        CPU 1: Dispatched process  2
Time slot   7
        CPU 2: Put process  1 to run queue
        CPU 2: Dispatched process  1
Time slot   8
        CPU 1: Put process  2 to run queue
        CPU 1: Dispatched process  2
Time slot   9
        Loaded a process at input/proc/s2, PID: 3 PRIO: 3
        CPU 0: Dispatched process  3
Time slot  10
        CPU 2: Put process  1 to run queue
        CPU 2: Dispatched process  1
```

```
        CPU 1: Put process  2 to run queue
        CPU 1: Dispatched process  2
Time slot  11
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  3
Time slot  12
        Loaded a process at input/proc/s3, PID: 4 PRIO: 8
        CPU 1: Put process  2 to run queue
        CPU 1: Dispatched process  2
        CPU 2: Put process  1 to run queue
        CPU 2: Dispatched process  4
Time slot  13
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  1
Time slot  14
        CPU 2: Put process  4 to run queue
        CPU 2: Dispatched process  4
        CPU 0: Processed  1 has finished
        CPU 0: Dispatched process  3
        CPU 1: Processed  2 has finished
        CPU 1 stopped
Time slot  15
        CPU 2: Put process  4 to run queue
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  3
        CPU 2: Dispatched process  4
Time slot  16
Time slot  17
        CPU 2: Put process  4 to run queue
        CPU 2: Dispatched process  4
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  3
Time slot  18
Time slot  19
        CPU 2: Put process  4 to run queue
        CPU 2: Dispatched process  4
Time slot  20
        CPU 0: Processed  3 has finished
        CPU 0 stopped
Time slot  21
Time slot  22
        CPU 2: Put process  4 to run queue
        CPU 2: Dispatched process  4
Time slot  23
Time slot  24
        CPU 2: Put process  4 to run queue
        CPU 2: Dispatched process  4
Time slot  25
Time slot  26
        CPU 2: Put process  4 to run queue
        CPU 2: Dispatched process  4
Time slot  27
        CPU 2: Processed  4 has finished
```

```
        CPU 2 stopped
```

Giải thích kết quả: Cơ chế tương tự trường hợp trên

Giản đồ Gantt:



Hình 4: Giản đồ Gantt mô tả kết quả đầu ra file $sched_1$.

5.3 Memory

Khi thực thi chương trình: Chúng ta sẽ quan sát được việc đưa các tiến trình vào Page Table, hiển thị thông tin của các tiến trình đang có trong Page Table cũng như các việc liên quan đến TLB. Đến với file textittest của chúng ta có những thông tin như sau:

```
6 2 2
1048576 16777216 0 0 0
0 test1 0
2 test2 15
```

Trong đó, 3 dòng đầu tương ứng là: [TIME SLICE] - [SỐ CPU THỰC THI] - [SỐ PROCESS THỰC THI], lần lượt là time slice (6), 2 cpu sẽ được thực thi trong chương trình và 4 process được thực thi. Dòng tiếp theo là khai báo kích thước của Page Table Size. Và 2 dòng cuối chỉ ra thông tin của các tiến trình: *p0s* và *p1s*. Các tiến trình được khai báo theo cú pháp như sau: [Thời gian Process xuất hiện] - [Tên process] - [Priority của Process đó]. Như file INPUT trên thì chúng ta sẽ có *test1* sẽ xuất hiện ở thời gian 0 với độ ưu tiên 0 và process *test* xuất hiện ở thời gian 2 với độ ưu tiên là 15. Chương trình sẽ thực thi như sau: chạy đến vào từng tiến trình và xem các câu lệnh trong tiến trình đó, với test1 và test2 ta có các lệnh như sau:

```
    1 14
    calc
    alloc 300 0
    alloc 300 4
    free 0
```

```
alloc 1024 1
alloc 1024 3
write 100 1 20
read 1 20 20
write 103 3 20
write 96 3 20
read 3 20 20
write 27 1 20
read 1 20 20
free 4
```

```
1 10
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
```

Với dòng đầu tiên, số 13 và tức là số lệnh mà process này sẽ thực hiện (hoặc có thể hiểu là thời gian mà Process này nắm giữ CPU),. Ta bắt đầu chạy sử dụng lệnh như đã đề cập ở trên và thu được kết quả là:

```
Time slot   0
ld_routine
        Loaded a process at input/proc/test1, PID: 1 PRIO: 0
Time slot   1
        CPU 0: Dispatched process  1
Time slot   2
        Alloc at region:0 size:300 in process 1
        Finish alloc region=0 size=300 adrress form 0 to 300 at process 1
        Loaded a process at input/proc/test2, PID: 2 PRIO: 15
Time slot   3
        Alloc at region:4 size:300 in process 1
        Finish alloc region=4 size=300 adrress form 300 to 600 at process 1
        CPU 1: Dispatched process  2
Time slot   4
        Free at region=0 at process 1
        Finish free region=0 adrress form 0 to 300 at process 1
Time slot   5
        Alloc at region:1 size:1024 in process 1
        Finish alloc region=1 size=1024 adrress form 600 to 1624 at process 1
Time slot   6
        Alloc at region:3 size:1024 in process 1
        Finish alloc region=3 size=1024 adrress form 1624 to 2648 at process 1
Time slot   7
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
```

```
        TLB miss at write region=1 offset=20 value=100
        write region=1 offset=20 value=100 process=1
print_pgtbl: 0 - 2816
00000000: 80000001
00000004: 80000000
00000008: 80000002
00000012: 80000006
00000016: 80000005
00000020: 80000004
00000024: 80000003
00000028: 8000000a
00000032: 80000009
00000036: 80000008
00000040: 80000007

        RAM content:
        Index 620, Frame 2, Content 100

Time slot   8
        TLB hit at read region=1 offset=20
        read region=1 offset=20 value=100 process=1
print_pgtbl: 0 - 2816
00000000: 80000001
00000004: 80000000
00000008: 80000002
00000012: 80000006
00000016: 80000005
00000020: 80000004
00000024: 80000003
00000028: 8000000a
00000032: 80000009
00000036: 80000008
00000040: 80000007

        RAM content:
        Index 620, Frame 2, Content 100

Time slot   9
        TLB miss at write region=3 offset=20 value=103
        write region=3 offset=20 value=103 process=1
print_pgtbl: 0 - 2816
00000000: 80000001
00000004: 80000000
00000008: 80000002
00000012: 80000006
00000016: 80000005
00000020: 80000004
00000024: 80000003
00000028: 8000000a
00000032: 80000009
00000036: 80000008
00000040: 80000007
```

```
        RAM content:
        Index 620, Frame 2, Content 100
        Index 876, Frame 3, Content 103
        CPU 1: Put process  2 to run queue
        CPU 1: Dispatched process  2

Time slot  10
        TLB hit at write region=3 offset=20 value=96
        write region=3 offset=20 value=96 process=1
print_pgtbl: 0 - 2816
00000000: 80000001
00000004: 80000000
00000008: 80000002
00000012: 80000006
00000016: 80000005
00000020: 80000004
00000024: 80000003
00000028: 8000000a
00000032: 80000009
00000036: 80000008
00000040: 80000007

        RAM content:
        Index 620, Frame 2, Content 100
        Index 876, Frame 3, Content 96

Time slot  11
        TLB hit at read region=3 offset=20
        read region=3 offset=20 value=96 process=1
print_pgtbl: 0 - 2816
00000000: 80000001
00000004: 80000000
00000008: 80000002
00000012: 80000006
00000016: 80000005
00000020: 80000004
00000024: 80000003
00000028: 8000000a
00000032: 80000009
00000036: 80000008
00000040: 80000007

        RAM content:
        Index 620, Frame 2, Content 100
        Index 876, Frame 3, Content 96

Time slot  12
        TLB hit at write region=1 offset=20 value=27
        write region=1 offset=20 value=27 process=1
print_pgtbl: 0 - 2816
00000000: 80000001
00000004: 80000000
00000008: 80000002
```

```
00000012: 80000006
00000016: 80000005
00000020: 80000004
00000024: 80000003
00000028: 8000000a
00000032: 80000009
00000036: 80000008
00000040: 80000007

        RAM content:
        Index 620, Frame 2, Content 27
        Index 876, Frame 3, Content 96

Time slot  13
        CPU 1: Processed  2 has finished
        CPU 1 stopped
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
        TLB hit at read region=1 offset=20
        read region=1 offset=20 value=27 process=1
print_pgtbl: 0 - 2816
00000000: 80000001
00000004: 80000000
00000008: 80000002
00000012: 80000006
00000016: 80000005
00000020: 80000004
00000024: 80000003
00000028: 8000000a
00000032: 80000009
00000036: 80000008
00000040: 80000007

        RAM content:
        Index 620, Frame 2, Content 27
        Index 876, Frame 3, Content 96

Time slot  14
        Free at region=4 at process 1
        Finish free region=4 adrress form 300 to 600 at process 1
Time slot  15
        CPU 0: Processed  1 has finished
        CPU 0 stopped
countTLB=7 countTLBMISS=2
Miss rate of tlb= 28.57%
```

Ở thời điểm đầu tiên, process mang tên test1 với độ ưu tiên 0 sẽ được chương trình thực thi, chương trình sẽ cấp cho process này với PID 1. Tương tự như thế, process test2 độ ưu tiên 15 sẽ được load ở time slot 2. Ở trong file của *test1* trên, ta thấy có lệnh alloc 300 0. Lệnh này sẽ thực hiện việc gọi hàm alloc, xin chương trình cấp phát cho bộ nhớ ảo 1 kích thước 300 tại region 0. Thật như vậy, kết quả đã được hiển thị trên chương trình việc đã được cấp phát thành công. Tiếp tục như thế, ta cấp phát cho region

4 kích thước là 300, sau đó ta thấy lệnh *free*. Với lệnh free 0, ta sẽ giải phóng vùng đã được cấp phát là vùng 0. Tiếp đến là ta lại cấp phát cho region 1 kích thước là 1024. Ta tiếp tục cấp phát cho region 3 kích thước là 1024 .Với lệnh *write*, ta sẽ thực hiện các bước như sau: Đầu tiên, ta xem coi tiến trình với value và offset này đã có trong TLB hay chưa, nếu chưa thì sẽ báo *TLB miss at ...*, sau đó ta tiến hành *write* bằng phương pháp paging. Ngược lại, nếu page đã có trong TLB (*TLB hit at ...*) , ta tiến hành đọc frame của page đó ở TLB và tiến hành *write* vào vùng nhớ đó. Cuối cùng in ra Page Table hiển thị toàn bộ các process đang có trong page table của các vùng mà ta đã cấp phát thông qua hàm **(print_pgtbl)** và in ra các giá trị đã được lưu trong bộ nhớ vật lý thông qua hàm **(MEMPHY_dump)**. Lúc nãy ta đã write cho region 1 offset 20 giá trị là 100, nên lúc gọi lệnh read này sẽ hiển thị là **TLB HIT**. Tiếp tục thực hiện như vậy, đến với lệnh *write* ở region 3, vì đây là lần đầu tiên ta thực hiện 1 lệnh read hoặc write lên region 3 (region 3 không cùng page với region 1 đang được lưu trong tlb) nên sẽ xảy ra TLB miss. Lúc này ta write cho region 3 offset 20 giá trị là 103. Lệnh tiếp theo là write region 3 offset 20 giá trị 96, vì region 3 vừa được ghi vào tlb nên xảy ra TLB hit, tiến hành write vào region 3. Tương tự với 3 lệnh read region 3 offset 20, lệnh write giá trị 27 vào region 1 offset 20 và lệnh read region 3 offset 20. Do 2 region này đã được lưu vào TLB cache nên xảy ra TLB hit. Lệnh cuối cùng là free region 4 thu hồi bộ nhớ từ địa chỉ 300 tới 600. Tổng cộng ta có 7 lần truy cập vào TLB và có 2 lần xảy ra miss nên ta suy ra được Miss rate của TLB là 28.57%.

Tài liệu

[1] ThS Nguyễn Lê Duy Lai, *Bộ slide bài giảng môn học Hệ điều hành học kỳ 232*, 2023 - 2024.

[2] *Paging in Operating System notes*, truy cập từ: Paging in OS | GATE|ESE

[3] *Translation Lookaside Buffer (TLB) in Paging*, truy cập từ: GeeksforGeeks - Translation lookaside buffer in paging