

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



## OPERATING SYSTEM (CO2017)

---

Report

# Simple Operating System

---

<b>Advisor:</b>	Nguyễn Thanh Quân	
<b>Students:</b>	Bùi Đức Anh	- 2112751
	Nguyễn Sỹ Dương	- 2113097
	Cao Đức Vinh	- 2115290

HO CHI MINH CITY, 2023

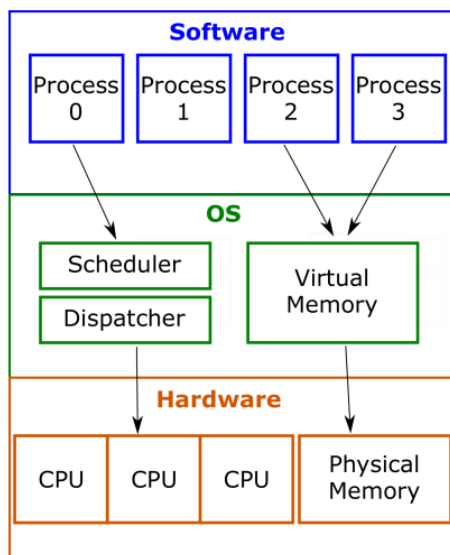
# Contents

<b>1</b>	<b>Lời mở đầu</b>	<b>2</b>
<b>2</b>	<b>Giải thích input process</b>	<b>3</b>
<b>3</b>	<b>Scheduler</b>	<b>5</b>
3.1	Cơ sở lý thuyết . . . . .	5
3.2	Trả lời câu hỏi . . . . .	5
3.3	Hiện thực Scheduler . . . . .	6
3.3.1	Hiện thực code trong file queue.c . . . . .	6
3.3.2	Hiện thực code trong file sched.c . . . . .	7
3.3.3	Trường hợp sched . . . . .	9
3.3.4	Trường hợp sched_0 . . . . .	10
3.3.5	Trường hợp sched_1 . . . . .	12
<b>4</b>	<b>Memory management</b>	<b>16</b>
4.1	Cơ sở lý thuyết . . . . .	16
4.1.1	Bộ nhớ ảo (Virtual memory) của mỗi quá trình . . . . .	16
4.1.2	Bộ nhớ vật lý (Physical memory) . . . . .	17
4.1.3	Dịch địa chỉ dựa trên phân trang (Paging-based address translation scheme) . . . . .	17
4.2	Trả lời câu hỏi . . . . .	18
4.3	Hiện thực . . . . .	20
4.3.1	ALLOC . . . . .	20
4.3.2	FREE . . . . .	23
4.3.3	READ . . . . .	23
4.4	Giải thích output . . . . .	24
<b>5</b>	<b>Put It All Together</b>	<b>32</b>
5.1	Cơ sở lý thuyết . . . . .	32
5.2	Trả lời câu hỏi . . . . .	32
5.3	Kết quả hiện thực . . . . .	33
<b>6</b>	<b>Tài liệu tham khảo</b>	<b>37</b>

## 1 Lời mở đầu

Mục đích của bài tập lớn là về mô phỏng một hệ điều hành đơn giản (**Simple OS**), qua đó giúp sinh viên hiểu được những khái niệm cơ bản về định thời (**scheduling**), đồng bộ (**synchronization**) và quản lý bộ nhớ (**memory management**). Hình 1 thể hiện kiến trúc tổng quan của một hệ điều hành mà chúng ta sẽ hiện thực. Về tổng quan, hệ điều hành phải quản lý hai nguồn tài nguyên ảo: CPU(s) và RAM qua việc sử dụng hai thành phần chính:

- **Scheduler** (và Dispatcher): quyết định quá trình (process) nào được chạy trên CPU
- **Virtual Memory Engine** (VME): cô lập không gian bộ nhớ của từng quá trình. RAM vật lý được chia sẻ bởi nhiều quá trình nhưng mỗi quá trình sẽ bỏ qua sự tồn tại của các quá trình khác. Để làm điều này, mỗi quá trình sẽ có vùng nhớ ảo riêng, VME giúp ánh xạ và dịch địa chỉ ảo cung cấp bởi các quá trình thành địa chỉ vật lý tương ứng.



Hình 1: Các module chính

Qua các module trên, hệ điều hành cho phép nhiều quá trình được tạo bởi người dùng để chia sẻ và sử dụng tài nguyên máy tính ảo. Cho nên, trong bài tập lớn này, ta sẽ tập trung vào hiện thực scheduler/dispatcher và VME.

## 2 Giải thích input process

Trong bài tập lớn này, ngoài việc phải quan tâm đến các giải thuật thì chúng ta cũng cần quan tâm đến các process input đầu vào và các mô tả của nó trong hệ điều hành giả lập được tạo ra.

- **m0s**

```
1 6
  alloc 300 0
  alloc 100 1
  free 0
  alloc 100 2
  write 102 1 20
  write 1 2 1000
```

Process này thực hiện 6 lệnh như mô tả, và số timeslot cần chạy để thực hiện process là 6.

- **m1s**

```
1 6
  alloc 300 0
  alloc 100 1
  free 0
  alloc 100 2
  free 2
  free 1
```

Process này thực hiện 6 lệnh như mô tả, và số timeslot cần chạy để thực hiện process là 6.

- **p0s**

```
1 14
  calc
  alloc 300 0
  alloc 300 4
  free 0
  alloc 100 1
  write 100 1 20
  read 1 20 20
  write 102 2 20
  read 2 20 20
  write 103 3 20
  read 3 20 20
  calc
  free 4
  calc
```

Process này thực hiện 14 lệnh như mô tả, và số timeslot cần chạy để thực hiện process là 14.

- **p1s**

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

Process này thực hiện 10 lệnh calc như mô tả, và số timeslot cần chạy để thực hiện process là 10.

Ta có thể hiểu tương tự với các input process còn lại như **p2s, p3s, s0, s1, s2, s3, s4**.

### 3 Scheduler

#### 3.1 Cơ sở lý thuyết

Trong bài tập lớn này, chúng ta sẽ sử dụng **giải thuật MLQ** (Multi Level Queue) để xác định tiến trình (process) sẽ được thực hiện trong quá trình định thời.

**Sơ lược về giải thuật MLQ:** là một giải thuật định thời CPU trong đó các tiến trình được phân thành nhiều hàng đợi khác nhau dựa trên mức độ ưu tiên của chúng. Các tiến trình ưu tiên cao được đặt trong các hàng đợi có mức độ ưu tiên cao hơn, trong khi các tiến trình ưu tiên thấp được đặt trong các hàng đợi có mức độ ưu tiên thấp hơn.

#### Các đặc trưng của giải thuật MLQ:

- Ready queue được chia thành nhiều hàng đợi riêng biệt theo một số tiêu chuẩn như:
  - Đặc điểm và yêu cầu định thời của tiến trình (process)
  - Foreground (interactive) và background process,...
- Process được gán cố định vào một hàng đợi, mỗi hàng đợi sử dụng giải thuật định thời riêng
- Hệ điều hành cần phải định thời cho các hàng đợi.
  - Fixed priority scheduling: phục vụ từ hàng đợi có độ ưu tiên cao đến thấp. Vấn đề: có thể có starvation
  - Time slice: mỗi hàng đợi được nhận một khoảng thời gian chiếm CPU và phân phối cho các process trong hàng đợi khoảng thời gian đó. Trong bài tập lớn lần này, các hàng đợi sẽ được thực thi bằng giải thuật định thời RR với số lượng phân bổ trong 1 lần chạy bằng `max_priority` trừ đi `priority`

#### 3.2 Trả lời câu hỏi

**Câu hỏi 1: Lợi thế của việc sử dụng Priority Queue so với các giải thuật định thời khác là gì?**

**Trả lời:** Khi so sánh với các giải thuật định thời khác, Priority Queue có một số ưu điểm:

- **Đảm bảo ưu tiên cho các tiến trình quan trọng hơn:** Khi sử dụng Priority Queue, các tiến trình quan trọng hơn sẽ được ưu tiên thực thi trước, đảm bảo rằng các tiến trình này sẽ được xử lý đúng lúc và đảm bảo tính đúng đắn của hệ thống.
- **Tối ưu hóa sử dụng CPU:** Priority Queue giúp tối ưu hóa sử dụng CPU bằng cách ưu tiên thực thi các tiến trình quan trọng hơn. Khi các tiến trình quan trọng được thực thi nhanh hơn, CPU có thể được sử dụng hiệu quả hơn để xử lý các tiến trình khác.
- **Tính linh hoạt và động lực cao:** Hàng đợi ưu tiên có thể được cập nhật động lực và ưu tiên của các tiến trình dựa trên sự thay đổi của hệ thống. Điều này giúp đảm bảo tính linh hoạt của hệ thống và cho phép các tiến trình quan trọng được thực thi đúng lúc trong mọi tình huống.
- **Thích ứng với các tác vụ có độ ưu tiên khác nhau:** Hàng đợi ưu tiên cho phép xử lý các tác vụ có độ ưu tiên khác nhau một cách hiệu quả, giúp hệ thống có thể thích ứng với nhiều loại tác vụ và môi trường khác nhau.

### 3.3 Hiện thực Scheduler

#### 3.3.1 Hiện thực code trong file queue.c

Hiện thực queue cơ bản theo nguyên tắc FIFO. Chúng ta có `MAX_QUEUE_SIZE = 10` được khai báo trong file `queue.h`, vậy nên mỗi queue không chứa quá 10 process.

- Hàm enqueue

**Chức năng:** Hàm enqueue giúp đưa process mới vào trong ready queue

```
1 void enqueue(struct queue_t *q, struct pcb_t *proc)
2 {
3     /* TODO: put a new process to queue [q] */
4     q->proc[q->size] = proc;
5     q->size++;
6 }
7
8 struct pcb_t *dequeue(struct queue_t *q)
9 {
10    /* TODO: return a pcb whose priority is the highest
11     * in the queue [q] and remember to remove it from q
12     */
13    if (empty(q))
14    {
15        return NULL;
16    }
17    struct pcb_t *temp = q->proc[0];
18    for (int i = 0; i < q->size - 1; i++)
19    {
20        q->proc[i] = q->proc[i + 1];
21    }
22    q->proc[q->size - 1] = NULL;
23    q->size--;
24    return temp;
25 }
```

- Hàm dequeue

**Chức năng:** Hàm de\_queue trả về process có có priority cao nhất trong ready queue và lấy nó ra khỏi ready queue.

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

### 3.3.2 Hiện thực code trong file sched.c

Scheduler có chức năng quản lý việc cập nhật các process sẽ được thực thi cho CPU, cụ thể là quản lý 2 hàng đợi ready và run như ở trên đã mô tả. Cụ thể:

- Bổ sung hàm `init_scheduler()`

```
1 {
2     #ifdef MLQ_SCHED
3     int i;
4     for (i = 0; i < MAX_PRIO; i++)
5     {
6         mlq_ready_queue[i].size = 0;
7         // init number of cpu each queue can use maximally
8         mlq_ready_queue[i].slot_cpu_can_use = MAX_PRIO - 1;
9     }
10    #endif
11    ready_queue.size = 0;
12    run_queue.size = 0;
13    pthread_mutex_init(&queue_lock, NULL);
14 }
```

- Hàm `struct pcb_t *get_proc()`

```
1 {
2     struct pcb_t *proc = NULL;
3     /*TODO: get a process from [ready_queue].
4      * Remember to use lock to protect the queue.
5      */
6     struct pcb_t *proc = NULL;
7     /*TODO: get a process from [ready_queue].
8      * Remember to use lock to protect the queue.
9      */
10    pthread_mutex_lock(&queue_lock);
11
12    if(empty(&ready_queue)){
13        /* if ready queue is empty, push all processes in run queue back to ready
14         queue*/
15        while(!empty(&run_queue)){
16            enqueue(&ready_queue, dequeue(&run_queue));
17        }
18    }
19    if(!empty(&ready_queue)){
20        proc=dequeue(&ready_queue);
21    }
22    /*unlock*/
23    pthread_mutex_unlock(&queue_lock);
24    return proc;
25 }
```

**Giải thích hàm `struct pcb_t *get_proc(void)`** Lấy PCB của một tiến trình đang chờ ở hàng đợi sẵn sàng. Nếu hàng đợi rỗng tại thời điểm hàm được gọi, bạn phải di chuyển tất cả PCB của các quy trình đang chờ ở hàng đợi chạy trở lại vào hàng đợi sẵn sàng trước khi nhận một tiến trình từ hàng đợi sẵn sàng.

- Hàm `get_mlq_proc()`

```
1 struct pcb_t *get_mlq_proc(void)
2 {
3     struct pcb_t *proc = NULL;
```



```
4 bool flag = false;
5 pthread_mutex_lock(&queue_lock);
6 for (int i = prio; i < MAX_PRIO; i++)
7 {
8     if (!empty(&mlq_ready_queue[i]) && queue_slot[i] > 0)
9     {
10         prio = i;
11         flag = true;
12         break;
13     }
14 }
15 if (!flag)
16 {
17     for (int i = 0; i < prio; i++)
18     {
19         if (!empty(&mlq_ready_queue[i]) && queue_slot[i] > 0)
20         {
21             prio = i;
22             flag = true;
23             break;
24         }
25     }
26     if (!flag)
27     {
28         prio = 0;
29         pthread_mutex_unlock(&queue_lock);
30         return NULL;
31     }
32 }
33 proc = dequeue(&mlq_ready_queue[prio]);
34 queue_slot[prio]--;
35 if (queue_slot[prio] == 0)
36 {
37     queue_slot[prio] = MAX_PRIO - prio;
38     prio++;
39     if (prio == MAX_PRIO)
40     {
41         prio = 0;
42     }
43 }
44 pthread_mutex_unlock(&queue_lock);
45 return proc;
46 }
```

#### Giải thích hàm struct pcb\_t \*get\_mlq\_proc(void)

- Ta tiến hành duyệt từng queue trong mlq\_ready\_queue từ vị trí prio đến cuối. Nếu lần đầu tiên tìm được queue không rỗng và slot > 0 thì dequeue queue đó để thực thi. Cùng với đó sẽ giảm slot đi 1.
- Nếu quá trình duyệt ở trên không lấy ra được process nào (ta nhận biết bằng một biến flag) thì sẽ một lần nữa duyệt từ đầu đến cuối tất cả hàng đợi trong ready queue với mục đích như cũ.
- Nếu lần duyệt thứ hai vẫn không lấy ra được process thì sẽ trả về NULL và gán prio = 0.
- Khi giải slot đi 1 sau khi đã lấy tiến trình ra khỏi hàng đợi, ta cần phải lưu s rằng nếu slot đã giảm về 0 thì phải phục hồi slot về giá trị MAX\_PRIO - prio như ban đầu và tăng prio lên 1 (queue hiện tại hết lượt truy cập, nhường tài nguyên cho queue tiếp

theo) nếu prio tăng đến MAX\_PRIO (đã duyệt xong queue cuối cùng) thì cập nhật prio = 0.

### 3.3.3 Trường hợp sched

Để chạy các test sched cần phải bỏ comment 3 dòng trong file **os-cfg.h**

4 2 3
0 p1s
1 p2s
2 p3s

#### Giải thích input

- Dòng đầu input "4 2 3" có nghĩa là
  - 4: Mỗi timeslice sẽ là 4
  - 2: 2 CPU là CPU0 và CPU1
  - 3: tổng cộng 3 process. Các process đều được định nghĩa là p1s nên thời gian thực thi của mỗi process là 10s. Các process có độ ưu tiên bé nhất là 1 nên số queue ở đây là 2.
- Gọi các process từ trên xuống lần lượt là process 1, process 2 và process 3. Do độ ưu tiên cao nhất của các process là 1 (ở process 1) nên số queue cần dùng là 2 queue, queue 0 (độ ưu tiên cao, được lấy 2 lần liên tiếp) và queue 1 (độ ưu tiên thấp hơn, chỉ được lấy 1 lần).

Listing 1: Output of sched

```
1 vinh@DESKTOP-AUULAN9:/mnt/c/VSCODE/C++/Assignment_L01_Nh m 9$ ./os sched
2 Time slot 0
3 ld_routine
4     Loaded a process at input/proc/p1s, PID: 1 PRI0: 1
5 Time slot 1
6     CPU 1: Dispatched process 1
7     Loaded a process at input/proc/p1s, PID: 2 PRI0: 0
8 Time slot 2
9     CPU 0: Dispatched process 2
10    Loaded a process at input/proc/p1s, PID: 3 PRI0: 0
11 Time slot 3
12 Time slot 4
13 Time slot 5
14    CPU 1: Put process 1 to run queue
15    CPU 1: Dispatched process 3
16 Time slot 6
17    CPU 0: Put process 2 to run queue
18    CPU 0: Dispatched process 2
19 Time slot 7
20 Time slot 8
21    CPU 1: Put process 3 to run queue
22    CPU 1: Dispatched process 3
23 Time slot 9
24 Time slot 10
25    CPU 0: Put process 2 to run queue
26    CPU 0: Dispatched process 2
27 Time slot 11
28 Time slot 12
29    CPU 0: Processed 2 has finished
```

```

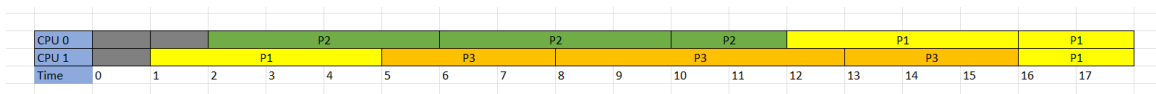
30      CPU 0: Dispatched process 1
31 Time slot 13
32      CPU 1: Put process 3 to run queue
33      CPU 1: Dispatched process 3
34 Time slot 14
35      CPU 1: Processed 3 has finished
36 Time slot 15
37      CPU 1 stopped
38 Time slot 16
39      CPU 0: Put process 1 to run queue
40      CPU 0: Dispatched process 1
41 Time slot 17
42 Time slot 18
43      CPU 0: Processed 1 has finished
44      CPU 0 stopped

```

### Giải thích output

- Tại timeslot 1, process 1 được load vào queue 1 và được thực thi ngay lập tức trong CPU0. (Hết 1 / 1 lần dispatch của queue 1)
- Tại timeslot 2, process 2 xuất hiện và được đưa vào queue 0, lúc này CPU1 đang trống nên process 2 được load vào CPU1 và thực thi. (1 / 2 lần dispatch của queue 0)
- Tại timeslot 5, process 3 xuất hiện sau đó và được đưa vào queue 0 đang trống. Tuy nhiên lúc này 2 CPU đều đã được sử dụng nên phải chờ.
- Sau khi hết timeslice 4s đến timeslot 4, process 1 được đưa ra khỏi CPU0 và trở về queue 1, lúc này ta thấy chỉ mới lấy từ queue 0 một lần, ta tiếp tục lấy process từ queue 0 một lần nữa nên process 3 được lấy ra và đưa vào CPU 0 thực thi (hết 2 / 2 lần dispatch của queue 0, trở về
- Đến timeslot 6, process 2 thực hiện xong 4s sẽ được đưa trở lại queue 0. Lúc này queue 0 đã được lấy 2 lần, con trỏ queue trở lại lấy các process từ queue 1, process 2 được đưa vào CPU1 đang trống queue 1).
- Tiếp tục tuần tự như vậy đến timeslot 17, khi CPU0 hoàn thành các process thì chương trình kết thúc

### Biểu đồ Gantt và kết quả chạy test sched



Hình 2: GANTT

### 3.3.4 Trường hợp sched\_0

2 1 2
0 s0
4 s1

### Giải thích input

- Dòng đầu input "2 1 2" có nghĩa là
  - 2: Mỗi timeslice sẽ là 4
  - 1: 1 CPU là CPU0
  - 2: tổng cộng 2 process. Các process đều được định nghĩa là s0 nên thời gian thực thi của mỗi process là 15s. Các process có độ ưu tiên bé nhất là 1 nên số queue ở đây là 2.

Listing 2: Output of sched\_0

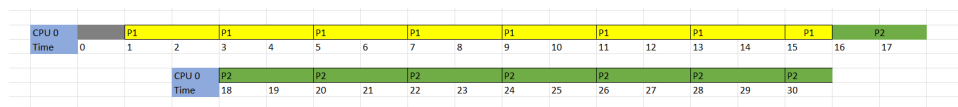
```
1 vinh@DESKTOP-AUULAN9:/mnt/c/VSCODE/C++/Assigment_L01_Nh m 9$ ./os sched_0
2 Time slot 0
3 ld_routine
4     Loaded a process at input/proc/s0, PID: 1 PRI0: 4
5 Time slot 1
6     CPU 0: Dispatched process 1
7     Loaded a process at input/proc/s0, PID: 2 PRI0: 0
8 Time slot 2
9 Time slot 3
10    CPU 0: Put process 1 to run queue
11    CPU 0: Dispatched process 1
12 Time slot 4
13 Time slot 5
14    CPU 0: Put process 1 to run queue
15    CPU 0: Dispatched process 1
16 Time slot 6
17 Time slot 7
18    CPU 0: Put process 1 to run queue
19    CPU 0: Dispatched process 1
20 Time slot 8
21 Time slot 9
22    CPU 0: Put process 1 to run queue
23    CPU 0: Dispatched process 1
24 Time slot 10
25 Time slot 11
26    CPU 0: Put process 1 to run queue
27    CPU 0: Dispatched process 1
28 Time slot 12
29 Time slot 13
30    CPU 0: Put process 1 to run queue
31    CPU 0: Dispatched process 1
32 Time slot 14
33 Time slot 15
34    CPU 0: Put process 1 to run queue
35    CPU 0: Dispatched process 1
36 Time slot 16
37    CPU 0: Processed 1 has finished
38    CPU 0: Dispatched process 2
39 Time slot 17
40 Time slot 18
41    CPU 0: Put process 2 to run queue
42    CPU 0: Dispatched process 2
43 Time slot 19
44 Time slot 20
45    CPU 0: Put process 2 to run queue
46    CPU 0: Dispatched process 2
47 Time slot 21
48 Time slot 22
49    CPU 0: Put process 2 to run queue
50    CPU 0: Dispatched process 2
```

```

51 Time slot 23
52 Time slot 24
53     CPU 0: Put process 2 to run queue
54     CPU 0: Dispatched process 2
55 Time slot 25
56 Time slot 26
57     CPU 0: Put process 2 to run queue
58     CPU 0: Dispatched process 2
59 Time slot 27
60 Time slot 28
61     CPU 0: Put process 2 to run queue
62     CPU 0: Dispatched process 2
63 Time slot 29
64 Time slot 30
65     CPU 0: Put process 2 to run queue
66     CPU 0: Dispatched process 2
67 Time slot 31
68     CPU 0: Processed 2 has finished
69     CPU 0 stopped

```

**Biểu đồ Gantt và kết quả chạy testcase sched\_0**



Hình 3: GANTT

### 3.3.5 Trường hợp sched\_1

2 1 4
0 s0
4 s1
6 s2
7 s3

#### Giải thích input

- Dòng đầu input "2 1 4" có nghĩa là
  - 2: Mỗi timeslice sẽ là 2
  - 1: 1 CPU là CPU0
  - 4: tổng cộng 4 process. Các process đều được định nghĩa s0 nên thời gian thực thi của mỗi process là 15s. Các process có độ ưu tiên thấp nhất là 4 nên số queue sử dụng là 5.
- Gọi các process từ trên xuống lần lượt là process 1, process 2, process 3 và process 4. Do độ ưu tiên cao nhất của các process là 4 (ở process 1) nên số queue cần dùng là 5 queue, queue 0 (độ ưu tiên cao, được lấy 2 lần liên tiếp), queue 4 (độ ưu tiên thấp hơn, chỉ được lấy 1 lần) các queue 1, 2, 3 được khai báo nhưng không được sử dụng do không có process có priority tương ứng

Listing 3: Output of sched\_1

```
1 Time slot 0
2 ld_routine
3   Loaded a process at input/proc/s0, PID: 1 PRI0: 4
4 Time slot 1
5   CPU 0: Dispatched process 1
6   Loaded a process at input/proc/s0, PID: 2 PRI0: 0
7 Time slot 2
8   Loaded a process at input/proc/s0, PID: 3 PRI0: 0
9 Time slot 3
10  CPU 0: Put process 1 to run queue
11  CPU 0: Dispatched process 2
12  Loaded a process at input/proc/s0, PID: 4 PRI0: 0
13 Time slot 4
14 Time slot 5
15  CPU 0: Put process 2 to run queue
16  CPU 0: Dispatched process 3
17 Time slot 6
18 Time slot 7
19  CPU 0: Put process 3 to run queue
20  CPU 0: Dispatched process 4
21 Time slot 8
22 Time slot 9
23  CPU 0: Put process 4 to run queue
24  CPU 0: Dispatched process 2
25 Time slot 10
26 Time slot 11
27  CPU 0: Put process 2 to run queue
28  CPU 0: Dispatched process 3
29 Time slot 12
30 Time slot 13
31  CPU 0: Put process 3 to run queue
32  CPU 0: Dispatched process 1
33 Time slot 14
34 Time slot 15
35  CPU 0: Put process 1 to run queue
36  CPU 0: Dispatched process 4
37 Time slot 16
38 Time slot 17
39  CPU 0: Put process 4 to run queue
40  CPU 0: Dispatched process 2
41 Time slot 18
42 Time slot 19
43  CPU 0: Put process 2 to run queue
44  CPU 0: Dispatched process 3
45 Time slot 20
46 Time slot 21
47  CPU 0: Put process 3 to run queue
48  CPU 0: Dispatched process 4
49 Time slot 22
50 Time slot 23
51  CPU 0: Put process 4 to run queue
52  CPU 0: Dispatched process 2
53 Time slot 24
54 Time slot 25
55  CPU 0: Put process 2 to run queue
56  CPU 0: Dispatched process 1
57 Time slot 26
58 Time slot 27
59  CPU 0: Put process 1 to run queue
60  CPU 0: Dispatched process 3
```



```
61 Time slot 28
62 Time slot 29
63     CPU 0: Put process 3 to run queue
64     CPU 0: Dispatched process 4
65 Time slot 30
66 Time slot 31
67     CPU 0: Put process 4 to run queue
68     CPU 0: Dispatched process 2
69 Time slot 32
70 Time slot 33
71     CPU 0: Put process 2 to run queue
72     CPU 0: Dispatched process 3
73 Time slot 34
74 Time slot 35
75     CPU 0: Put process 3 to run queue
76     CPU 0: Dispatched process 4
77 Time slot 36
78 Time slot 37
79     CPU 0: Put process 4 to run queue
80     CPU 0: Dispatched process 1
81 Time slot 38
82 Time slot 39
83     CPU 0: Put process 1 to run queue
84     CPU 0: Dispatched process 2
85 Time slot 40
86 Time slot 41
87     CPU 0: Put process 2 to run queue
88     CPU 0: Dispatched process 3
89 Time slot 42
90 Time slot 43
91     CPU 0: Put process 3 to run queue
92     CPU 0: Dispatched process 4
93 Time slot 44
94 Time slot 45
95     CPU 0: Put process 4 to run queue
96     CPU 0: Dispatched process 2
97 Time slot 46
98 Time slot 47
99     CPU 0: Put process 2 to run queue
100    CPU 0: Dispatched process 3
101 Time slot 48
102 Time slot 49
103    CPU 0: Put process 3 to run queue
104    CPU 0: Dispatched process 1
105 Time slot 50
106 Time slot 51
107    CPU 0: Put process 1 to run queue
108    CPU 0: Dispatched process 4
109 Time slot 52
110 Time slot 53
111    CPU 0: Put process 4 to run queue
112    CPU 0: Dispatched process 2
113 Time slot 54
114    CPU 0: Processed 2 has finished
115    CPU 0: Dispatched process 3
116 Time slot 55
117    CPU 0: Processed 3 has finished
118    CPU 0: Dispatched process 4
119 Time slot 56
120    CPU 0: Processed 4 has finished
121    CPU 0: Dispatched process 1
122 Time slot 57
```

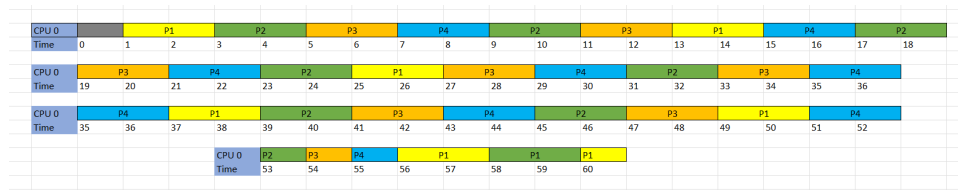
```
123 Time slot 58
124 CPU 0: Put process 1 to run queue
125 CPU 0: Dispatched process 1
126 Time slot 59
127 Time slot 60
128 CPU 0: Put process 1 to run queue
129 CPU 0: Dispatched process 1
130 Time slot 61
131 CPU 0: Processed 1 has finished
132 CPU 0 stopped
```

### Giải thích output

- Tại timeslot 0, process 1 được load vào queue 4 và được thực thi ngay lập tức. Sau đó 1 giây process 2 cũng đã xuất hiện và được đưa vào queue 0.
- Tại timeslot 2, process 1 đã thực hiện xong 2s thì sẽ được lấy ra khỏi CPU và đưa về queue 4 vì queue 4 chỉ có độ ưu tiên thấp, mỗi lần gọi chỉ được 1 lần dispatch. Lúc này, queue 0 đang có process 2, process này được load vào CPU và thực thi, lúc này process 3 được đưa vào queue 0 đang trống. Sau đó 1 giây tại timeslot 3, process 4 được đưa vào queue 0 ngay sau process 3.
- Tại timeslot 4, process 2 ra khỏi CPU, đưa về queue 0 sau process 4 và hệ điều hành tiếp tục lấy process từ queue 0 (lần 2 / 5 lần dispatch), lúc này process 3 được đưa vào CPU và thực thi.
- • Tại timeslot 6, process 3 ra khỏi CPU, đưa về queue 0 sau process 2 và hệ điều hành tiếp tục lấy process từ queue 0 (lần 3 / 5 lần dispatch), lúc này process 4 được đưa vào CPU và thực thi.
- • Tại timeslot 8, process 4 ra khỏi CPU, đưa về queue 0 sau process 3 và hệ điều hành tiếp tục lấy process từ queue 0 (lần 4 / 5 lần dispatch), lúc này process 2 được đưa vào CPU và thực thi.
- • Tại timeslot 10, process 2 ra khỏi CPU, đưa về queue 0 sau process 4 và hệ điều hành tiếp tục lấy process từ queue 0 (lần 5 / 5 lần dispatch), lúc này process 3 được đưa vào CPU và thực thi.
- • Tại timeslot 12, sau khi đủ 5 lần dispatch, hệ điều hành tiếp tục tìm các process ở các queue thấp hơn để thực hiện, nó xuống queue 4 và đưa process 1 vào CPU.
- • Tại timeslot 14, sau khi process 1 thực hiện xong (lần 1 / 1 lần dispatch) queue 4, hệ điều hành sẽ lại quay về dispatch 5 lần trên queue 0 và cứ tiếp tục như thế đến khi các process hoàn thành

### Biểu đồ Gantt và kết quả chạy testcase sched\_1





Hình 4: GANTT

## 4 Memory management

### 4.1 Cơ sở lý thuyết

**Quản lý bộ nhớ** (tiếng Anh: memory management) là việc điều hành bộ nhớ máy tính ở cấp bậc hệ thống. Mục đích quan trọng của việc quản lý bộ nhớ là cung cấp những cách thức để cấp phát động các ô nhớ cho chương trình khi được yêu cầu và giải phóng các ô nhớ đó khi không cần dùng nữa. Đây là việc rất quan trọng đối với bất kỳ hệ thống máy tính cao cấp nào vì sẽ có nhiều công việc được tiến hành ở mọi thời điểm.

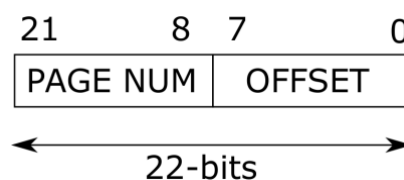
#### 4.1.1 Bộ nhớ ảo (Virtual memory) của mỗi quá trình

Không gian vùng nhớ ảo (virtual memory space) được thiết kế sử dụng phương pháp ánh xạ bộ nhớ (memory mapping) cho mỗi process PCB.

**Vùng nhớ (Memory area)** là một dải liên tục dọc các địa chỉ ô nhớ ảo được ánh xạ trực tiếp tới các địa chỉ vật lý tương ứng. Trong đó, có các vùng nhớ con liên tục (hoặc rời rạc) chứa nội dung của các quá trình (code, data,...) được gọi là **Memory Region**.

**Địa chỉ CPU (CPU Address):** địa chỉ được tạo ra bởi CPU nhằm truy cập tới một vị trí ô nhớ cụ thể. Trong hệ thống với cấu trúc phân trang, nó được chia thành:

- **Page number (p):** được dùng như chỉ số của bảng phân trang đang lưu trữ địa chỉ nền của các trang ở bộ nhớ vật lý (**Physical memory**).
- **Page offset (d):** kết hợp với địa chỉ nền để định nghĩa địa chỉ bộ nhớ vật lý mà sẽ được gửi cho **Memory Management Unit**



Hình 5: CPU Address

#### 4.1.2 Bộ nhớ vật lý (Physical memory)

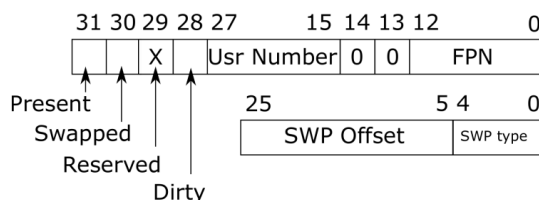
Tương tự như **bộ nhớ ảo (Virtual memory)**, **bộ nhớ vật lý (Physical memory)** được chia thành các khối nhớ nhỏ có kích thước cố định gọi là **frame** hay **page frame**. Mỗi frame trong bài tập lớn này được lưu frame number sẵn có trong mã nguồn định nghĩa về cấu trúc của frame.

Như đã đề cập, tất cả vùng nhớ có vùng nhớ ảo được ánh xạ, và chúng độc lập với nhau. Tuy nhiên, tất cả ánh xạ đó đều có chỉ tới một thiết bị nhớ vật lý đơn lẻ (**Singleton physical device**). Trong bài tập lớn này có hai thiết bị vật lý cần quan tâm:

- **RAM (Random Access Memory)**: có thể được truy cập trực tiếp từ CPU address bus.
- **SWAP**: thiết bị nhớ thứ cấp, không thể truy cập trực tiếp từ CPU. Để thực thi các chương trình và dữ liệu lưu ở thiết bị này, chúng phải được chuyển lên bộ nhớ chính (**main memory**) để thực thi.

#### 4.1.3 Dịch địa chỉ dựa trên phân trang (Paging-based address translation scheme)

Mỗi một **virtual page** của process đều có một bảng phân trang (**page table**). Bảng phân trang giúp **userspace process** tìm ra được **physical frame** của mỗi **virtual page** được ánh xạ tới. Trong mỗi **page table**, có các **Page Table Entries (PTE)**. Trong các page table entries bao gồm một giá trị 32-bit cho mỗi virtual page, có định nghĩa về data và cấu trúc một PTE như sau:



Hình 6: Page Table Entry format

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

Hình 7: Page table format

**Memory Swapping:** Khi thực thi một process, các nội dung trong page của process tương ứng sẽ được tải vào bất kỳ frame nào trong bộ nhớ chính, cụ thể ở đây là **MEMRAM**. Khi máy tính vượt quá lượng bộ nhớ chính cho phép, tức là không còn frame trống trong **MEMRAM**, hệ điều hành sẽ di chuyển những nội dung trong page không mong muốn đến bộ nhớ thứ cấp, cụ thể là vào **frame MEMSWAP** (swapping out). Ngược lại, cơ chế swapping in sẽ di chuyển các nội dung của page ta cần dùng để thực thi process tương ứng, nhưng hiện tại đang ở trong **MEMSWAP**, vào các frame **MEMRAM** bộ nhớ chính. Cơ chế thực hiện swapping out trong bài tập lớn này được thực hiện theo quy tắc **First In First Out (FIFO)**, tức là nội dung trong page được tải vào frame **MEMRAM** sớm nhất trong số các page đang ở trong **MEMRAM**, sẽ được sao chép hoặc chuyển vào frame trong **MEMSWAP**, sau đó giải phóng bộ nhớ trong frame **MEMRAM** đó.

## 4.2 Trả lời câu hỏi

**Câu hỏi 1: Điều gì sẽ xảy ra nếu chúng ta chia địa chỉ thành nhiều hơn 2 cấp trong hệ thống quản lý bộ nhớ phân trang?**

Trong hệ thống quản lý bộ nhớ phân trang, địa chỉ bộ nhớ được chia thành hai cấp:

- **Trang (Page)**
- **Độ dời offset (Page offset)**

Tuy nhiên, nếu chúng ta chia địa chỉ thành nhiều hơn 2 cấp, điều này sẽ dẫn đến sự phức tạp hóa trong việc quản lý bộ nhớ và có thể ảnh hưởng đến hiệu suất hệ thống.

Khi chia địa chỉ thành nhiều hơn 2 cấp, ta sẽ cần sử dụng nhiều bảng trang để ánh xạ từng cấp địa chỉ vào vị trí bộ nhớ tương ứng. Việc sử dụng nhiều bảng trang này sẽ làm tăng thời gian truy cập vào bộ nhớ để ánh xạ địa chỉ, dẫn đến giảm hiệu suất của hệ thống.

Ngoài ra, việc chia địa chỉ thành nhiều hơn 2 cấp cũng đòi hỏi bộ nhớ phải lưu trữ nhiều thông tin hơn, dẫn đến sự lãng phí tài nguyên bộ nhớ. Điều này có thể gây ra các vấn đề liên quan đến khả năng lưu trữ và quản lý bộ nhớ, như thiếu bộ nhớ hoặc sự chậm trễ trong việc truy cập bộ nhớ.

Vì vậy, trong hệ thống quản lý bộ nhớ phân trang, chia địa chỉ thành nhiều hơn 2 cấp không được khuyến khích để đảm bảo tính đơn giản và hiệu quả của hệ thống quản lý bộ nhớ.

**Câu hỏi 2: Lợi ích và bất cập trong việc phân đoạn bộ nhớ**

Thiết kế nhiều phân đoạn bộ nhớ trong hệ điều hành có một số lợi ích quan trọng như sau:

- **Sử dụng hiệu quả bộ nhớ:** Với thiết kế nhiều phân đoạn, bộ nhớ có thể được sử dụng hiệu quả hơn bởi vì không cần phải tải toàn bộ chương trình hoặc quá trình vào bộ nhớ khi chúng được khởi động. Thay vào đó, chỉ có phần của chương trình hoặc quá trình cần được tải vào bộ nhớ, giúp tiết kiệm tài nguyên bộ nhớ.
- **Bảo vệ dữ liệu:** Với nhiều phân đoạn, các khu vực bộ nhớ khác nhau có thể được phân biệt và bảo vệ khỏi sự xâm nhập của các chương trình khác hoặc người dùng, giúp đảm bảo tính toàn vẹn của dữ liệu và cải thiện bảo mật hệ thống.

- **Tăng tốc độ thực thi:** Khi chỉ một phần của chương trình hoặc quá trình được tải vào bộ nhớ, việc truy cập và thực thi các phần của chương trình hoặc quá trình này sẽ nhanh hơn, giúp cải thiện tốc độ thực thi và hiệu suất của hệ thống.
- **Dễ dàng quản lý:** Thiết kế nhiều phân đoạn giúp quản lý bộ nhớ và các tài nguyên của hệ thống dễ dàng hơn, bởi vì các khu vực bộ nhớ khác nhau có thể được quản lý và giám sát độc lập nhau.

Tuy nhiên, thiết kế nhiều phân đoạn cũng có một số nhược điểm như tăng khối lượng công việc của bộ điều khiển bộ nhớ và độ phức tạp của quá trình quản lý bộ nhớ. Do đó, việc thiết kế phân đoạn bộ nhớ cần được cân nhắc kỹ lưỡng để đảm bảo rằng nó phù hợp với các yêu cầu và mục đích của hệ thống.

## 4.3 Hiện thực

### 4.3.1 ALLOC

- `int find_victim_page(struct mm_struct *mm, int *retpgn);`

```
1 int find_victim_page(struct mm_struct *mm, int *retpgn) {
2     struct pgm_t *pg = mm->fifo_pgn;
3
4     /* TODO: Implement the theoretical mechanism to find the victim page */
5     if (pg == NULL) return -1;
6
7     struct pgm_t *prev = NULL;
8     while (pg->pg_next) {
9         prev = pg;
10        pg = pg->pg_next;
11    }
12
13    *retpgn = pg->pgn;
14    prev->pg_next = NULL;
15
16    free(pg);
17
18    return 0;
19 }
20
```

**Chức năng:** Hàm `find_victim_page` có chức năng là tìm và trả về trang (page) nạn nhân từ một không gian bộ nhớ ảo (virtual memory space) được đại diện bởi một cấu trúc dữ liệu `mm_struct`. Trang nạn nhân là trang cần bị loại bỏ để tạo chỗ cho trang mới (Trong hệ điều hành này sử dụng thuật toán FIFO).

- `int alloc_pages_range(struct pcb_t *caller, int req_pgn, struct framephy_struct **frm_lst);`

```
1 int alloc_pages_range(struct pcb_t *caller, int req_pgn, struct
2     framephy_struct **frm_lst) {
3     int pgit, fpn;
4     struct framephy_struct *newfp_str = NULL;
5
6     for (pgit = 0; pgit < req_pgn; pgit++) {
7         newfp_str = (struct framephy_struct *)malloc(sizeof(struct
8             framephy_struct));
9
10        if (MEMPHY_get_freefp(caller->mram, &fpn) == 0) {
11            newfp_str->fpn = fpn;
12        }
13        else { // ERROR CODE of obtaining some but not enough frames
14            int vicpgn, swpfpn;
15            if (find_victim_page(caller->mm, &vicpgn) == -1
16                || MEMPHY_get_freefp(caller->active_mswp, &swpfpn) == -1)
17            {
18                if (*frm_lst == NULL) {
19                    return -1;
20                }
21                else {
22                    struct framephy_struct *freefp_str;
23                    while (*frm_lst != NULL) {
24                        freefp_str = *frm_lst;
25                        *frm_lst = (*frm_lst)->fp_next;
26                        free(freefp_str);
27                    }
28                }
29            }
30        }
31    }
32
33    return 0;
34 }
```

```
25     }
26     return -3000;
27 }
28 }
29
30 uint32_t vicpte = caller->mm->pgd[vicpgn];
31 int vicfpn = PAGING_FPN(vicpte);
32 __swap_cp_page(caller->mram, vicfpn, caller->active_mswp, swpfpn);
33 pte_set_swap(&caller->mm->pgd[vicpgn], 0, swpfpn);
34 newfp_str->fpn = vicfpn;
35 }
36
37 newfp_str->fp_next = *frm_lst;
38 *frm_lst = newfp_str;
39 }
40
41 return 0;
42 }
43 }
```

**Chức năng:** Hàm `alloc_pages_range` có chức năng cấp phát một dãy các trang vật lý (physical pages) cho một process với cấu trúc `pcb_t`. Các trang vật lý này được lưu trữ trong danh sách liên kết `frm_lst`, và số lượng trang cần cấp phát được xác định bởi tham số `req_pgnum`.

- `int vmmap_page_range(struct pcb_t *caller, int addr, int pgnum, struct framephy_struct *frames, struct vm_rg_struct *ret_rg);`

```
1  /*
2   * vmmap_page_range - map a range of page at aligned address
3   */
4  int vmmap_page_range(
5      struct pcb_t *caller,          // process call
6      int addr,                     // start address which is aligned to pagesz
7      int pgnum,                    // num of mapping page
8      struct framephy_struct *frames, // list of the mapped frames
9      struct vm_rg_struct *ret_rg)   // return mapped region, the real mapped fp
10 {
11     struct framephy_struct *fpit;
12     int pgit = 0;
13     int pgn = PAGING_PGN(addr);
14
15     ret_rg->rg_start = addr; // at least the very first space is usable
16     ret_rg->rg_end = ret_rg->rg_start + pgnum * PAGING_PAGESZ;
17
18     /* TODO map range of frame to address space
19      * [addr to addr + pgnum*PAGING_PAGESZ
20      * in page table caller->mm->pgd[]
21      */
22     for (; pgit < pgnum; ++pgit)
23     {
24         fpit = frames;
25         pte_set_fpn(&caller->mm->pgd[pgn + pgit], fpit->fpn);
26         frames = frames->fp_next;
27         free(fpit);
28
29         /* Tracking for later page replacement activities (if needed)
30          * Enqueue new usage page */
31         enlist_pgn_node(&caller->mm->fifo_pgn, pgn + pgit);
32     }
33 }
```

```
34     return 0;  
35 }
```

**Chức năng:** Hàm `vmap_page_range` có chức năng ánh xạ một dãy trang vật lý (physical pages) vào không gian địa chỉ ảo (virtual address space) của một tiến trình, được đại diện bởi cấu trúc `pcb_t`. Dãy trang vật lý này được đại diện bởi danh sách liên kết frames và được ánh xạ vào không gian địa chỉ ảo từ địa chỉ `addr` với số lượng trang là `pgnum`.

- `int __alloc(struct pcb_t *caller, int vmaid, int rgid, int size, int *alloc_addr);`

```
1  int __alloc(struct pcb_t *caller, int vmaid, int rgid, int size, int *  
    alloc_addr) {  
2      /*Allocate at the topproof */  
3      pthread_mutex_lock(&mmvm_lock);  
4      struct vm_rg_struct rgnode;  
5  
6      if (get_free_vmrg_area(caller, vmaid, size, &rgnode) == 0)  
7      {  
8          caller->mm->symrgtbl[rgid].rg_start = rgnode.rg_start;  
9          caller->mm->symrgtbl[rgid].rg_end = rgnode.rg_end;  
10  
11         *alloc_addr = rgnode.rg_start;  
12         pthread_mutex_unlock(&mmvm_lock);  
13         return 0;  
14     }  
15  
16     /* TODO get_free_vmrg_area FAILED handle the region management (Fig.6)*/  
17  
18     /*Attempt to incrate limit to get space */  
19     struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);  
20     int inc_sz = PAGING_PAGE_ALIGNSZ(size);  
21     // int inc_limit_ret  
22     int old_sbrk;  
23  
24     old_sbrk = cur_vma->sbrk;  
25  
26     /* TODO INCREASE THE LIMIT  
27      * inc_vma_limit(caller, vmaid, inc_sz)  
28      */  
29     inc_vma_limit(caller, vmaid, inc_sz);  
30  
31     /*Successful increase limit */  
32     caller->mm->symrgtbl[rgid].rg_start = old_sbrk;  
33     caller->mm->symrgtbl[rgid].rg_end = old_sbrk + size;  
34  
35     *alloc_addr = old_sbrk;  
36  
37     struct vm_area_struct *remain_rg = get_vma_by_num(caller->mm, vmaid);  
38     if (old_sbrk + size < remain_rg->sbrk) {  
39         struct vm_rg_struct *rg_free = malloc(sizeof(struct vm_rg_struct));  
40         rg_free->rg_start = old_sbrk + size;  
41         rg_free->rg_end = remain_rg->sbrk;  
42         enlist_vm_freerg_list(caller->mm, rg_free);  
43     }  
44  
45     pthread_mutex_unlock(&mmvm_lock);  
46     return 0;  
47 }
```

**Chức năng:** Hàm `__alloc` có chức năng cấp phát một vùng nhớ cho một tiến trình thông qua việc quản lý không gian địa chỉ ảo (virtual address space).

### 4.3.2 FREE

- `int __free(struct pcb_t *caller, int vmaid, int rgid);`

```
1 int __free(struct pcb_t *caller, int vmaid, int rgid) {
2     pthread_mutex_lock(&mmvm_lock);
3
4     if (rgid < 0 || rgid > PAGING_MAX_SYMTBL_SZ) {
5         pthread_mutex_unlock(&mmvm_lock);
6         return -1;
7     }
8
9     /* TODO: Manage the collect freed region to freerg_list */
10    struct vm_rg_struct *rgnode = get_symrg_byid(caller->mm, rgid);
11
12    if (rgnode->rg_start == 0 && rgnode->rg_end == 0) {
13        pthread_mutex_unlock(&mmvm_lock);
14        return -1;
15    }
16
17    struct vm_rg_struct *freerg_node = malloc(sizeof(struct vm_rg_struct));
18    freerg_node->rg_start = rgnode->rg_start;
19    freerg_node->rg_end = rgnode->rg_end;
20    freerg_node->rg_next = NULL;
21
22    rgnode->rg_start = rgnode->rg_end = 0;
23    rgnode->rg_next = NULL;
24
25    /*enlist the obsoleted memory region */
26    enlist_vm_freerg_list(caller->mm, freerg_node);
27
28    pthread_mutex_unlock(&mmvm_lock);
29    return 0;
30 }
```

**Chức năng:** Hàm `__free` có chức năng giải phóng một vùng nhớ trong không gian địa chỉ ảo của một tiến trình, thông qua quản lý bảng địa chỉ ảo và danh sách vùng nhớ không sử dụng.

### 4.3.3 READ

- `int pg_getpage(struct mm_struct *mm, int pgn, int *fpn, struct pcb_t *caller);`

```
1 int pg_getpage(struct mm_struct *mm, int pgn, int *fpn, struct pcb_t *caller)
2 {
3     uint32_t pte = mm->pgd[pgn];
4
5     if (!PAGING_PAGE_PRESENT(pte))
6     { /* Page is not online, make it actively living */
7         int vicpgn = -1, swpfpn = -1;
8         // uint32_t vicpte;
9
10        int tgtfpn = PAGING_SWP(pte); // the target frame storing our variable
11
12        /* TODO: Play with your paging theory here */
13        /* Find victim page */
14        find_victim_page(caller->mm, &vicpgn);
15        int vicfpn = GETVAL(caller->mm->pgd[vicpgn], PAGING_PTE_FPN_MASK,
16        PAGING_PTE_FPN_LOBIT);
17        /* Get free frame in MEMSWP */
18        MEMPHY_get_freefp(caller->active_mswp, &swpfpn);
```



```
18     if (swfpfn == -1 || vicpgn == -1)
19     {
20         return -1;
21     }
22     /* Do swap frame from MEMRAM to MEMSWP and vice versa*/
23     /* Copy victim frame to swap */
24     __swap_cp_page(caller->mram, vicfpn, caller->active_mswp, swfpfn);
25     /* Copy target frame from swap to mem */
26     __swap_cp_page(caller->active_mswp, tgtfpn, caller->mram, vicfpn);
27
28     /* Update page table */
29     pte_set_swap(&mm->pgd[vicpgn], 0, swfpfn); //// swap
30
31     /* Update its online status of the target page */
32     // pte_set_fpn() & mm->pgd[pgn];
33     pte_set_fpn(&pte, tgtfpn);
34     enlist_pgn_node(&caller->mm->fifo_pgn, pgn);
35 }
36
37 *fpn = PAGING_FPN(pte);
38
39 return 0;
40 }
```

**Chức năng:** Hàm `pg_getpage` có chức năng lấy trang từ bộ nhớ và đảm bảo rằng trang đó đã được đưa vào bộ nhớ. Nếu trang không nằm trong bộ nhớ (không được hiện hành), hàm sẽ thực hiện một loạt các bước để đưa trang đó vào bộ nhớ, có thể thông qua quá trình swap.

#### 4.4 Giải thích output

Trước tiên ta sẽ giải thích ý nghĩa của file input cho chương trình:

`os_1_mlq_paging.txt`

```
2 4 8
1048576 16777216 0 0 0
1 p0s 130
2 s3 39
4 m1s 15
6 s2 120
7 m0s 120
9 p1s 15
11 s0 38
16 s1 0
```

- **Dòng 1:** 2, 4, 8 lần lượt là số lượng CPU, số lượng processes và số lượng time slots.
- **Dòng 2:** thông tin về bộ nhớ ảo:
  - 1048576 là dung lượng của vùng nhớ ảo cho mỗi process (1 MB).
  - 16777216 là tổng dung lượng bộ nhớ ảo (16 MB).
  - 0 0 0 là dung lượng của các khu vực bộ nhớ thực (physical memory regions), ở đây có 3 khu vực.

- **Dòng 3-10:** Mô tả các quy tắc và độ ưu tiên cho việc tải các processes.
  - 1 p0s 130: Load process có PID=1 từ file p0s với độ ưu tiên là 130.
  - 2 s3 39: Schedule process có PID=2 với độ ưu tiên là 39.
  - 4 m1s 15: Load memory image từ file m1s cho process có PID=4 với độ ưu tiên là 15.
  - 6 s2 120: Schedule process có PID=6 với độ ưu tiên là 120.
  - 7 m0s 120: Load memory image từ file m0s cho process có PID=7 với độ ưu tiên là 120.
  - 9 p1s 15: Load process có PID=9 từ file p1s với độ ưu tiên là 15.
  - 11 s0 38: Schedule process có PID=11 với độ ưu tiên là 38.
  - 16 s1 0: Schedule process có PID=16 với độ ưu tiên là 0.

#### os\_1\_mlq\_paging.output

```
1 Time slot 0
2 ld_routine
3 Time slot 1
4   Loaded a process at input/proc/p0s, PID: 1 PRI0: 130
5   CPU 0: Dispatched process 1
6 Time slot 2
7   Loaded a process at input/proc/s3, PID: 2 PRI0: 39
8 Time slot 3
9   CPU 2: Dispatched process 2
10  CPU 0: Put process 1 to run queue
11  CPU 0: Dispatched process 1
12 Time slot 4
13  Loaded a process at input/proc/m1s, PID: 3 PRI0: 15
14 Time slot 5
15  CPU 0: Put process 1 to run queue
16  CPU 0: Dispatched process 1
17  CPU 3: Dispatched process 3
18  CPU 2: Put process 2 to run queue
19  CPU 2: Dispatched process 2
20 Time slot 6
21  Loaded a process at input/proc/s2, PID: 4 PRI0: 120
22  CPU 1: Dispatched process 4
23 write region=1 offset=20 value=100
24 print_pgtbl: 0 - 1024
25 00000000: 80000001
26 00000004: 80000000
27 00000008: 80000003
28 00000012: 80000002
29 ADDRESS | VALUE
30 Time slot 7
31  CPU 3: Put process 3 to run queue
32  CPU 3: Dispatched process 3
33  CPU 2: Put process 2 to run queue
34  CPU 2: Dispatched process 2
35  CPU 0: Put process 1 to run queue
36  CPU 0: Dispatched process 1
37 read region=1 offset=20 value=100
38 print_pgtbl: 0 - 1024
39 00000000: 80000001
40 00000004: 80000000
41 00000008: 80000003
42 00000012: 80000002
```



```
43 ADDRESS | VALUE
44 00000098: 100
45 Loaded a process at input/proc/m0s, PID: 5 PRI0: 120
46 Time slot 8
47 CPU 1: Put process 4 to run queue
48 CPU 1: Dispatched process 5
49 write region=2 offset=20 value=102
50 print_pgtbl: 0 - 1024
51 00000000: 80000001
52 00000004: 80000000
53 00000008: 80000003
54 00000012: 80000002
55 ADDRESS | VALUE
56 00000098: 100
57 Time slot 9
58 CPU 0: Put process 1 to run queue
59 CPU 0: Dispatched process 4
60 CPU 3: Put process 3 to run queue
61 CPU 3: Dispatched process 1
62 CPU 2: Put process 2 to run queue
63 CPU 2: Dispatched process 3
64 Loaded a process at input/proc/p1s, PID: 6 PRI0: 15
65 read region=2 offset=20 value=102
66 print_pgtbl: 0 - 1024
67 00000000: 80000001
68 00000004: 80000000
69 00000008: 80000003
70 00000012: 80000002
71 ADDRESS | VALUE
72 00000014: 102
73 00000098: 100
74 Time slot 10
75 write region=3 offset=20 value=103
76 print_pgtbl: 0 - 1024
77 00000000: 80000001
78 00000004: 80000000
79 00000008: 80000003
80 00000012: 80000002
81 ADDRESS | VALUE
82 00000014: 102
83 00000098: 100
84 CPU 1: Put process 5 to run queue
85 CPU 1: Dispatched process 6
86 Time slot 11
87 CPU 2: Put process 3 to run queue
88 CPU 3: Processed 1 has finished
89 Loaded a process at input/proc/s0, PID: 7 PRI0: 38
90 CPU 2: Dispatched process 3
91 CPU 3: Dispatched process 2
92 CPU 0: Put process 4 to run queue
93 CPU 0: Dispatched process 5
94 Time slot 12
95 CPU 1: Put process 6 to run queue
96 CPU 1: Dispatched process 4
97 Time slot 13
98 CPU 3: Put process 2 to run queue
99 CPU 3: Dispatched process 6
100 CPU 0: Put process 5 to run queue
101 CPU 0: Dispatched process 7
102 CPU 2: Processed 3 has finished
103 CPU 2: Dispatched process 2
104 Time slot 14
```



```
105 CPU 1: Put process 4 to run queue
106 CPU 1: Dispatched process 5
107 write region=1 offset=20 value=102
108 print_pgtbl: 0 - 512
109 00000000: 80000007
110 00000004: 80000006
111 ADDRESS | VALUE
112 00000014: 103
113 00000098: 100
114 Time slot 15
115 CPU 2: Put process 2 to run queue
116 CPU 2: Dispatched process 4
117 write region=2 offset=1000 value=1
118 print_pgtbl: 0 - 512
119 00000000: 80000007
120 00000004: 80000006
121 ADDRESS | VALUE
122 00000014: 103
123 00000098: 100
124 000000a4: 102
125 CPU 0: Put process 7 to run queue
126 CPU 0: Dispatched process 7
127 CPU 3: Put process 6 to run queue
128 CPU 3: Dispatched process 2
129 Time slot 16
130 CPU 3: Processed 2 has finished
131 CPU 3: Dispatched process 6
132 Loaded a process at input/proc/s1, PID: 8 PRIO: 0
133 CPU 1: Put process 5 to run queue
134 CPU 1: Dispatched process 5
135 write region=0 offset=0 value=0
136 print_pgtbl: 0 - 512
137 00000000: c0000000
138 00000004: 80000006
139 ADDRESS | VALUE
140 00000014: 103
141 00000098: 100
142 000000a4: 102
143 000000b0: 1
144 Time slot 17
145 CPU 1: Processed 5 has finished
146 CPU 1: Dispatched process 8
147 CPU 2: Put process 4 to run queue
148 CPU 2: Dispatched process 4
149 CPU 0: Put process 7 to run queue
150 CPU 0: Dispatched process 7
151 Time slot 18
152 CPU 3: Put process 6 to run queue
153 CPU 3: Dispatched process 6
154 Time slot 19
155 CPU 2: Put process 4 to run queue
156 CPU 2: Dispatched process 4
157 CPU 0: Put process 7 to run queue
158 CPU 0: Dispatched process 7
159 CPU 1: Put process 8 to run queue
160 CPU 1: Dispatched process 8
161 Time slot 20
162 CPU 3: Put process 6 to run queue
163 CPU 3: Dispatched process 6
164 Time slot 21
165 CPU 0: Put process 7 to run queue
166 CPU 0: Dispatched process 7
```

```
167 CPU 2: Processed 4 has finished
168 CPU 2 stopped
169 CPU 1: Put process 8 to run queue
170 CPU 1: Dispatched process 8
171 Time slot 22
172 CPU 3: Processed 6 has finished
173 CPU 3 stopped
174 Time slot 23
175 CPU 1: Put process 8 to run queue
176 CPU 1: Dispatched process 8
177 CPU 0: Put process 7 to run queue
178 CPU 0: Dispatched process 7
179 Time slot 24
180 CPU 1: Processed 8 has finished
181 CPU 1 stopped
182 Time slot 25
183 CPU 0: Put process 7 to run queue
184 CPU 0: Dispatched process 7
185 Time slot 26
186 Time slot 27
187 CPU 0: Put process 7 to run queue
188 CPU 0: Dispatched process 7
189 Time slot 28
190 CPU 0: Processed 7 has finished
191 CPU 0 stopped
```

- **Time slot 0:** Thực hiện `ld_routine`.
- **Time slot 1:**
  - Quá trình với PID 1 được nạp từ `input/proc/p0s`.
  - CPU 0 gán nhiệm vụ cho process 1.
- **Time slot 2:**
  - Quá trình với PID 2 được nạp từ `input/proc/s3`.
  - CPU 2 gán nhiệm vụ cho process 2.
  - CPU 0 đưa process 1 vào hàng đợi chờ và gán nhiệm vụ cho nó.
- **Time slot 3:**
  - Quá trình với PID 3 được nạp từ `input/proc/m1s`.
  - CPU 0 đưa process 1 vào hàng đợi chờ và gán nhiệm vụ cho nó.
  - CPU 3 gán nhiệm vụ cho process 3.
  - CPU 2 đưa process 2 vào hàng đợi chờ và gán nhiệm vụ cho nó.
- **Time slot 4:**
  - Quá trình với PID 4 được nạp từ `input/proc/s2`.
  - CPU 1 gán nhiệm vụ cho process 4.
  - Thực hiện một hoạt động ghi vào region 1, offset 20, với giá trị là 100.
  - In các mục bảng trang.
- **Time slot 5:**

- CPU 3 đưa process 3 vào hàng đợi chờ và gán nhiệm vụ cho nó.
- CPU 2 đưa process 2 vào hàng đợi chờ và gán nhiệm vụ cho nó.
- CPU 0 đưa process 1 vào hàng đợi chờ và gán nhiệm vụ cho nó.

• **Time slot 6:**

- Quá trình với PID 5 được nạp từ input/proc/m0s.
- CPU 1 đưa process 4 vào hàng đợi chờ và gán nhiệm vụ cho process 5.

• **Time slot 7:**

- CPU 3 đưa process 3 vào hàng đợi chờ và gán nhiệm vụ cho nó.
- CPU 2 đưa process 2 vào hàng đợi chờ và gán nhiệm vụ cho nó.
- CPU 0 đưa process 1 vào hàng đợi chờ và gán nhiệm vụ cho nó.
- Thực hiện một hoạt động đọc từ region 1, offset 20, hiển thị giá trị là 100.
- In các mục bảng trang.

• **Time slot 8:**

- CPU 1 đưa process 5 vào hàng đợi chờ và gán nhiệm vụ cho nó.
- Thực hiện một hoạt động ghi vào region 2, offset 20, với giá trị là 102.
- In các mục bảng trang.

• **Time slot 9:**

- CPU 0 đưa process 1 vào hàng đợi chờ và gán nhiệm vụ cho nó.
- CPU 3 đưa process 3 vào hàng đợi chờ và gán nhiệm vụ cho process 1.
- CPU 2 đưa process 2 vào hàng đợi chờ và gán nhiệm vụ cho process 3.
- Quá trình với PID 6 được nạp từ input/proc/pls.
- Thực hiện một hoạt động đọc từ region 2, offset 20, hiển thị giá trị là 102.
- In các mục bảng trang.

• **Time slot 10:**

- Thực hiện một hoạt động ghi vào region 3, offset 20, với giá trị là 103.
- In các mục bảng trang.

• **Time slot 11:**

- CPU 1 đưa process 5 vào hàng đợi chờ và gán nhiệm vụ cho process 6.
- CPU 2 đưa process 3 vào hàng đợi chờ.
- CPU 3 xử lý và hoàn thành process 1.
- Quá trình với PID 7 được nạp từ input/proc/s0.
- CPU 2 gán nhiệm vụ cho process 3.
- CPU 3 gán nhiệm vụ cho process 2.
- CPU 0 đưa process 4 vào hàng đợi chờ và gán nhiệm vụ cho process 5.

- **Time slot 12:**
  - CPU 1 đưa process 6 vào hàng đợi chờ và gán nhiệm vụ cho nó.
- **Time slot 13:**
  - CPU 3 đưa process 2 vào hàng đợi chờ và gán nhiệm vụ cho process 6.
  - CPU 0 đưa process 5 vào hàng đợi chờ và gán nhiệm vụ cho process 7.
  - CPU 2 xử lý và hoàn thành process 3.
  - CPU 2 gán nhiệm vụ cho process 2.
- **Time slot 14:**
  - CPU 1 đưa process 4 vào hàng đợi chờ và gán nhiệm vụ cho process 5.
  - Thực hiện một hoạt động ghi vào region 1, offset 20, với giá trị là 102.
  - In các mục bảng trang.
- **Time slot 15:**
  - CPU 2 đưa process 2 vào hàng đợi chờ và gán nhiệm vụ cho process 4.
  - Thực hiện một hoạt động ghi vào region 2, offset 1000, với giá trị là 1.
  - In các mục bảng trang.
- **Time slot 16:**
  - CPU 3 xử lý và hoàn thành process 2.
  - CPU 3 gán nhiệm vụ cho process 6.
  - Quá trình với PID 8 được nạp từ input/proc/s1.
  - CPU 1 đưa process 5 vào hàng đợi chờ và gán nhiệm vụ cho process 8.
- **Time slot 17:**
  - CPU 1 xử lý và hoàn thành process 5.
  - CPU 1 gán nhiệm vụ cho process 8.
  - CPU 2 đưa process 4 vào hàng đợi chờ và gán nhiệm vụ cho nó.
  - CPU 0 đưa process 7 vào hàng đợi chờ và gán nhiệm vụ cho nó.
- **Time slot 18:**
  - CPU 3 đưa process 6 vào hàng đợi chờ và gán nhiệm vụ cho nó.
- **Time slot 19:**
  - CPU 2 đưa process 4 vào hàng đợi chờ và gán nhiệm vụ cho nó.
  - CPU 0 đưa process 7 vào hàng đợi chờ và gán nhiệm vụ cho nó.
  - CPU 1 đưa process 8 vào hàng đợi chờ và gán nhiệm vụ cho nó.
- **Time slot 20:**
  - CPU 3 đưa process 6 vào hàng đợi chờ và gán nhiệm vụ cho nó.

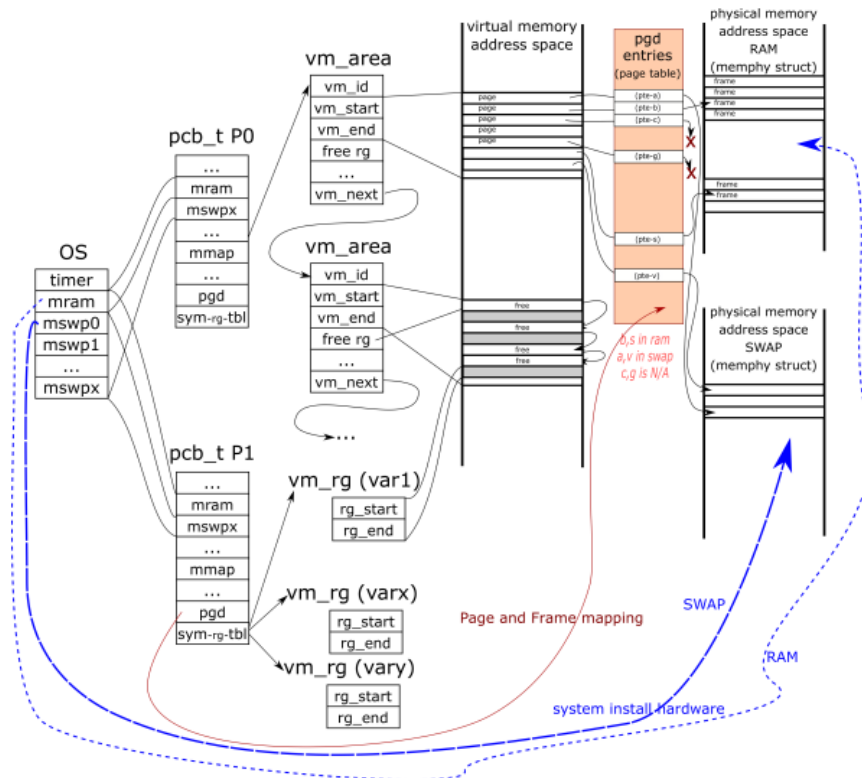
- **Time slot 21:**
  - CPU 0 đưa process 7 vào hàng đợi chờ và gán nhiệm vụ cho nó.
  - CPU 2 xử lý và hoàn thành process 4.
  - CPU 2 dừng lại.
  - CPU 1 đưa process 8 vào hàng đợi chờ và gán nhiệm vụ cho nó.
- **Time slot 22:**
  - CPU 3 xử lý và hoàn thành process 6.
  - CPU 3 dừng lại.
- **Time slot 23:**
  - CPU 1 đưa process 8 vào hàng đợi chờ và gán nhiệm vụ cho nó.
  - CPU 0 đưa process 7 vào hàng đợi chờ và gán nhiệm vụ cho nó.
- **Time slot 24:**
  - CPU 1 xử lý và hoàn thành process 8.
  - CPU 1 dừng lại.
- **Time slot 25:**
  - CPU 0 đưa process 7 vào hàng đợi chờ và gán nhiệm vụ cho nó.
- **Time slot 26:**
  - Không có sự kiện quan trọng.
- **Time slot 27:** CPU 0 đưa process 7 vào hàng đợi chờ và gán nhiệm vụ cho nó.
- **Time slot 28:**
  - CPU 0 xử lý và hoàn thành process 7.
  - CPU 0 dừng lại.



## 5 Put It All Together

### 5.1 Cơ sở lý thuyết

Sau khi tổng hợp hai phần trên, ta sẽ có được một hệ điều hành đơn giản như sau:



Hình 8: Thao tác liên quan bộ nhớ ảo

### 5.2 Trả lời câu hỏi

**Câu hỏi :** Điều gì sẽ xảy ra nếu việc đồng bộ hóa không được xử lý trong hệ điều hành đơn giản? Minh họa bằng ví dụ vấn đề của hệ điều hành đơn giản này, nếu có?

Trong một hệ điều hành multi-tasking, đồng bộ hóa là rất quan trọng để đảm bảo rằng nhiều process hoặc thread không can thiệp vào việc thực thi lẫn nhau và truy cập vào tài nguyên được chia sẻ. Nếu đồng bộ hóa không được xử lý đúng cách, nó có thể dẫn đến tình trạng race conditions, deadlocks và các sự cố khác có thể khiến hệ thống trở nên không ổn định hoặc không phản hồi.

### 5.3 Kết quả hiện thực

Sử dụng lệnh `./os os_1_singleCPU_mlq_paging` cho file `os_1_singleCPU_mlq_paging` ta thu được output như sau:

```
1 Time slot 0
2 ld_routine
3 Time slot 1
4   Loaded a process at input/proc/s4, PID: 1 PRI0: 4
5 Time slot 2
6   CPU 0: Dispatched process 1
7   Loaded a process at input/proc/s3, PID: 2 PRI0: 3
8 Time slot 3
9 Time slot 4
10  CPU 0: Put process 1 to run queue
11  CPU 0: Dispatched process 1
12  Loaded a process at input/proc/m1s, PID: 3 PRI0: 2
13 Time slot 5
14 Time slot 6
15  CPU 0: Put process 1 to run queue
16  CPU 0: Dispatched process 1
17  Loaded a process at input/proc/s2, PID: 4 PRI0: 3
18 Time slot 7
19  Loaded a process at input/proc/m0s, PID: 5 PRI0: 3
20 Time slot 8
21  CPU 0: Put process 1 to run queue
22  CPU 0: Dispatched process 1
23 Time slot 9
24  CPU 0: Processed 1 has finished
25  Loaded a process at input/proc/p1s, PID: 6 PRI0: 2
26  CPU 0: Dispatched process 3
27 Time slot 10
28 Time slot 11
29  CPU 0: Put process 3 to run queue
30  CPU 0: Dispatched process 6
31  Loaded a process at input/proc/s0, PID: 7 PRI0: 1
32 Time slot 12
33 Time slot 13
34  CPU 0: Put process 6 to run queue
35  CPU 0: Dispatched process 3
36 Time slot 14
37 Time slot 15
38  CPU 0: Put process 3 to run queue
39  CPU 0: Dispatched process 6
40 Time slot 16
41  Loaded a process at input/proc/s1, PID: 8 PRI0: 0
42 Time slot 17
43  CPU 0: Put process 6 to run queue
44  CPU 0: Dispatched process 3
45 Time slot 18
46 Time slot 19
47  CPU 0: Put process 3 to run queue
48  CPU 0: Dispatched process 6
49 Time slot 20
50 Time slot 21
51  CPU 0: Put process 6 to run queue
52  CPU 0: Dispatched process 3
53 Time slot 22
54 Time slot 23
55  CPU 0: Processed 3 has finished
56  CPU 0: Dispatched process 6
57 Time slot 24
```



```
58 Time slot 25
59 CPU 0: Put process 6 to run queue
60 CPU 0: Dispatched process 6
61 Time slot 26
62 Time slot 27
63 CPU 0: Processed 6 has finished
64 CPU 0: Dispatched process 2
65 Time slot 28
66 Time slot 29
67 CPU 0: Put process 2 to run queue
68 CPU 0: Dispatched process 4
69 Time slot 30
70 Time slot 31
71 CPU 0: Put process 4 to run queue
72 CPU 0: Dispatched process 5
73 Time slot 32
74 Time slot 33
75 CPU 0: Put process 5 to run queue
76 CPU 0: Dispatched process 2
77 Time slot 34
78 Time slot 35
79 CPU 0: Put process 2 to run queue
80 CPU 0: Dispatched process 4
81 Time slot 36
82 Time slot 37
83 CPU 0: Put process 4 to run queue
84 CPU 0: Dispatched process 5
85 Time slot 38
86 Time slot 39
87 CPU 0: Put process 5 to run queue
88 CPU 0: Dispatched process 2
89 Time slot 40
90 Time slot 41
91 CPU 0: Put process 2 to run queue
92 CPU 0: Dispatched process 4
93 Time slot 42
94 Time slot 43
95 CPU 0: Put process 4 to run queue
96 CPU 0: Dispatched process 5
97 write region=1 offset=20 value=102
98 print_pgtbl: 0 - 512
99 00000000: 80000003
100 00000004: 80000002
101 ADDRESS | VALUE
102 Time slot 44
103 write region=2 offset=1000 value=1
104 print_pgtbl: 0 - 512
105 00000000: 80000003
106 00000004: 80000002
107 ADDRESS | VALUE
108 000000a4: 102
109 Time slot 45
110 CPU 0: Put process 5 to run queue
111 CPU 0: Dispatched process 2
112 Time slot 46
113 Time slot 47
114 CPU 0: Put process 2 to run queue
115 CPU 0: Dispatched process 4
116 Time slot 48
117 Time slot 49
118 CPU 0: Put process 4 to run queue
119 CPU 0: Dispatched process 5
```



```
120 write region=0 offset=0 value=0
121 print_pgtbl: 0 - 512
122 00000000: c0000000
123 00000004: 80000002
124 ADDRESS | VALUE
125 000000a4: 102
126 000000b0: 1
127 Time slot 50
128   CPU 0: Processed 5 has finished
129   CPU 0: Dispatched process 2
130 Time slot 51
131 Time slot 52
132   CPU 0: Put process 2 to run queue
133   CPU 0: Dispatched process 4
134 Time slot 53
135 Time slot 54
136   CPU 0: Put process 4 to run queue
137   CPU 0: Dispatched process 2
138 Time slot 55
139   CPU 0: Processed 2 has finished
140   CPU 0: Dispatched process 4
141 Time slot 56
142 Time slot 57
143   CPU 0: Processed 4 has finished
144   CPU 0: Dispatched process 8
145 Time slot 58
146 Time slot 59
147   CPU 0: Put process 8 to run queue
148   CPU 0: Dispatched process 8
149 Time slot 60
150 Time slot 61
151   CPU 0: Put process 8 to run queue
152   CPU 0: Dispatched process 8
153 Time slot 62
154 Time slot 63
155   CPU 0: Put process 8 to run queue
156   CPU 0: Dispatched process 8
157 Time slot 64
158   CPU 0: Processed 8 has finished
159   CPU 0: Dispatched process 7
160 Time slot 65
161 Time slot 66
162   CPU 0: Put process 7 to run queue
163   CPU 0: Dispatched process 7
164 Time slot 67
165 Time slot 68
166   CPU 0: Put process 7 to run queue
167   CPU 0: Dispatched process 7
168 Time slot 69
169 Time slot 70
170   CPU 0: Put process 7 to run queue
171   CPU 0: Dispatched process 7
172 Time slot 71
173 Time slot 72
174   CPU 0: Put process 7 to run queue
175   CPU 0: Dispatched process 7
176 Time slot 73
177 Time slot 74
178   CPU 0: Put process 7 to run queue
179   CPU 0: Dispatched process 7
180 Time slot 75
181 Time slot 76
```



```
182 CPU 0: Put process 7 to run queue
183 CPU 0: Dispatched process 7
184 Time slot 77
185 Time slot 78
186 CPU 0: Put process 7 to run queue
187 CPU 0: Dispatched process 7
188 Time slot 79
189 CPU 0: Processed 7 has finished
190 CPU 0 stopped
```



## 6 Tài liệu tham khảo

1. <https://www.geeksforgeeks.org/memory-management-in-operating-system/>
2. <https://www.geeksforgeeks.org/process-schedulers-in-operating-system/>
3. [https://www.tutorialspoint.com/operating\\_system/os\\_process\\_scheduling\\_algorithms.htm](https://www.tutorialspoint.com/operating_system/os_process_scheduling_algorithms.htm)