

**REPORT** ● ● ● ● ●

# **OPERATING SYSTEM ASSIGNMENT**

**[Simple Operating System]**

10/04/2023 To 29/05/2023, HO CHI MINH, VIETNAM

[AUTHORS]

TRẦN MINH HIẾU - 2113363  
NGUYỄN PHẠM THIỀN PHÚC - 2114445  
TRƯƠNG THUẬN HƯNG - 2113619  
NGUYỄN HOÀNG KIM - 2111617



HCMC University of Technology

Block A3, 268 Ly Thuong Kiet

Ward 14, District 10

HCM City

T +84 28 3864 7256

Faculty of Computer Science and Engineering,  
Department of Systems and Networkings

*Operating System*

*Simple Operating System*

### **Assignment Report**

Author(s) : Hieu, Phuc, Hung, Kim  
Title : OS Assignment  
Publisher : HCMUT  
Project Number : 28  
Country: Vietnam

# CONTENTS

2.1	Scheduler	7
2.2	Memory Management	7
3.1	Module 1 - Scheduler	9
3.2	Module 2 - Paging-based memory management	16
3.3	Module 3 - Put it all together	30
4.1	Role and contribution of each member 1	35
4.2	Project output	36
4.3	Project outcome	37

# SUMMARY

Bài tập lớn yêu cầu các thành viên của nhóm triển khai, mô phỏng một hệ điều hành đơn giản viết bằng ngôn ngữ C. Thông qua các cơ chế định thời và quản lý bộ nhớ, hệ điều hành có thể quản lý và phân phối các tài nguyên trong máy tính. Qua bài tập lớn lần này, các thành viên trong nhóm đã tiếp cận và hiểu rõ hơn về các thành phần cơ bản của một hệ điều hành và cách hoạt động của chúng. Điều này sẽ giúp các thành viên nắm vững kiến thức của môn Hệ điều hành nói riêng, và tạo ra nền tảng vững chắc cho các môn học của chuyên ngành Khoa học máy tính nói chung.

Dự án này bao gồm hiện thực hai phần là định thời và quản lý bộ nhớ của một hệ điều hành. Đối với phần định thời, nhóm sử dụng giải thuật multilevel queue (MLQ) với priority kết hợp với cơ chế Round Robin để hiện thực. Để có thể thực hiện nó, các thành viên nhóm đã phải tìm hiểu thật kỹ lý thuyết về MLQ, các giải thuật định thời và các khái niệm về process, thread, mutex... trước khi thực hành để tránh những sai sót có thể xảy ra. Đối với phần quản lý bộ nhớ, nhóm sử dụng cơ chế paging để hiện thực. Bên cạnh đó, cần xây dựng giải thuật thay trang cho mỗi lần page fault (ở đây nhóm sử dụng giải thuật FIFO). Tiếp đó, các thành viên nhóm phải tìm hiểu về các cơ chế như mapping từ bộ nhớ ảo sang bộ nhớ vật lý, cách để cấp phát vùng nhớ cho một process và cách giải phóng vùng nhớ sau khi thực thi để tránh memory leak...

Checklist	Y/N
Is it ½ page maximum	Y
Is it written in an active voice?	Y
Does it answer Who, What, When, Where, Why and How?	Y
Does the opening paragraph describe project objectives and its purpose?	Y
Does it present contextual background and principle (in short aka less than 1 page)?	Y
Does it highlight the report's final finding or project contributions or results?	Y
Does the closing paragraph provide a conclusion (optional: next step recommendation)?	Y
Is it consistent in the usage of the same keywords, terms, technical phrases?	Y
Have the authors check and eliminate the statement/information that is not discussed in the main body of the report	Y

## Author(s)

Nhóm trưởng: Nguyễn Phạm Thiên Phúc

Đóng góp của các thành viên:

- Nguyễn Phạm Thiên Phúc: thiết kế phần định thời, hiện thực các cơ chế cấp phát và thu hồi của memory
- Trần Minh Hiếu: thiết kế phần định thời, hiện thực các cơ chế cấp phát và thu hồi của memory
- Trương Thuận Hưng: thiết kế phần định thời, giải thuật thay trang, hàm pg\_getpage(), viết báo cáo
- Nguyễn Hoàng Kim: thiết kế phần định thời, giải thuật thay trang, hàm pg\_getpage(), viết báo cáo

# 1

# INTRODUCTION TO THE ASSIGNMENT

Bài tập lớn yêu cầu mô phỏng một hệ điều hành đơn giản giúp sinh viên hiểu các khái niệm cơ bản về định thời (scheduling), đồng bộ (synchronization) và quản lý bộ nhớ (memory management).

Các kiến thức cần được trang bị để hiểu phần còn lại của báo cáo:

- Scheduler algorithm và dispatcher: quyết định xem process nào được đưa vào ready queue và thời điểm process được đưa vào CPU.
- Virtual memory engine (VME): cỗ lập không gian bộ nhớ của mỗi process ra khỏi các process khác. Bộ nhớ RAM vật lý được chia sẻ bởi nhiều process nhưng mỗi process lại không biết sự tồn tại của những process khác. Điều này được thực hiện bằng cách để mỗi process có không gian bộ nhớ ảo riêng của nó và VME sẽ ánh xạ và dịch địa chỉ ảo được các process cung cấp thành địa chỉ vật lý.
- Synchronization: là cơ chế đảm bảo rằng chỉ có một tiến trình truy cập tài nguyên (như bộ nhớ) tại một thời điểm. Nó ngăn chặn xung đột khi nhiều tiến trình cùng truy cập tài nguyên đó.

Nghiên cứu này được thực hiện để minh họa các lý thuyết được học trong lớp Hệ điều hành. Bài báo cáo không đưa ra những phát hiện mới mà chủ yếu là tổng hợp và vận dụng kiến thức đã có. Mặc dù vậy, nó vẫn mang ý nghĩa quan trọng trong việc củng cố hiểu biết về định thời và quản lý bộ nhớ - hai chủ đề quan trọng trong hệ điều hành.

## CHECKLIST

Checklists	Y/N
Why was the work done?	Y
What must be known to understand the rest of the report?	Y
Explain the historical issue behind the research and its significance?  Did a specific issue/event impact this study?	Y N
Is it a subsequent phase of R&D study? (YES- it illustrate class theory)  Is it breaking new ground? (Usually assignment NO)	Y N

# 2

# FINDINGS AND CONCLUSION

## 2.1

### Scheduler

#### 2.1.1 Multilevel queue (MLQ)

Multilevel queue (MLQ) là một kỹ thuật định thời (scheduling) trong hệ điều hành, cho phép phân loại các process vào các hàng đợi khác nhau dựa theo độ ưu tiên. Các process có độ ưu tiên cao hơn được đặt ở những hàng đợi cao hơn. Các hàng đợi có time slot khác nhau. Đây cũng là một giải thuật preemptive. Ngoài ra, ở mỗi hàng đợi cũng có thể xây dựng các giải thuật định thời khác nhau như FCFS, Round-Robin... Ở bài tập lớn lần này, MLQ gồm 140 hàng chờ với độ ưu tiên từ 0 tới 139 và time slot tương ứng từ 140 đến 1 đơn vị thời gian.

#### 2.1.2 Các công việc cần thực hiện

- Các hàm thực hiện thao tác thêm và xóa PCB trong hàng đợi: enqueue() và dequeue()
- Thiết kế và hiện thực hàm get\_mlq\_proc() để lấy ra một PCB của process trong MLQ thích hợp
- Chạy thử code và kiểm tra lại kết quả

## 2.2

### Memory Management

Sử dụng cơ chế phân trang (paging) để thực hiện quản lý bộ nhớ. Trong đó bộ nhớ vật lý được chia thành các khối có kích thước cố định được gọi là frame và bộ nhớ logic được chia thành các khối có kích thước bằng nhau được gọi là khung trang (page). Hệ điều hành ánh xạ các trang thành các khung trang, cho phép các tiến trình truy cập bộ nhớ trong các đơn vị nhỏ hơn, có kích thước cố định.

#### 2.2.1 Bộ nhớ ảo

Bộ nhớ ảo được tạo ra bằng cách tạo ra các khu vực nhớ ảo (vm\_area) cho các process sử dụng. Mỗi khu vực nhớ ảo được chia thành các vm\_region tương ứng cho các biến. Những biến này sẽ được quản lý bởi một mảng symrtbl.

#### 2.2.2 Bộ nhớ vật lý

Trong bài tập lớn lần này, bộ nhớ vật lý được mô phỏng qua 1 thiết bị bộ nhớ sơ cấp RAM và 4 thiết bị bộ nhớ thứ cấp SWAP.

Mỗi khi RAM đã hết, sử dụng cơ chế thay trang để swap in/swap out giữa các frame trong RAM và trong SWAP.

#### 2.2.3 Các công việc cần thực hiện

- Xây dựng giải thuật FIFO để tìm victim page trong page table với hàm find\_victim\_page
- Hiện thực các cơ chế cấp phát và giải phóng bộ nhớ qua các hàm \_alloc và \_free
- Ánh xạ (mapping) giữa bộ nhớ ảo và bộ nhớ vật lý khi cấp phát thêm vùng nhớ qua hai hàm alloc\_pages\_range và hàm vmap\_page\_range.
- Hiện thực hàm pg\_getpage với cơ chế swap in/swap out để tìm frame tương ứng từ bộ nhớ SWAP lên bộ nhớ chính RAM, hỗ trợ việc đọc và ghi trong hàm \_\_read và \_\_write.
- Chạy thử code và kiểm tra lại kết quả.

**2.3****Put it all together**

Kết hợp cả hai phần định thời và quản lý bộ nhớ để hoàn thành một hệ điều hành cơ bản. Và tiến hành xử lý bất đồng bộ với cơ chế bảo vệ mutex.

**Kết quả:**

Sau khi hiện thực phần định thời và quản lý bộ nhớ, kết hợp với xử lý bất đồng bộ, code đã hiện thực được và chạy được các input test case của bài tập lớn.

**CHECKLIST**

Checklists	Y/N
Provides statements (keep it (max) of 5 bullet items)	Y
What project should we achieve?  Verify the consistency these statement with evaluations do later	Y
What was the work conducted for the project?  Minus listing the work conducted. Major task completed for the projects	Y

# 3

## ACTIONS FOR FOLLOW-UP

### 3.1

#### Module 1 - Scheduler

##### 3.1.1 Implementation

###### 3.1.1.1 Thêm PCB mới vào queue bằng hàm enqueue()

- **Ý tưởng:**

Kiểm tra xem hàng đợi đã đầy hay chưa, nếu chưa thì thêm PCB vào hàng đợi và tăng size của hàng đợi lên 1 đơn vị.

- **Code:**



```
1 void enqueue(struct queue_t * q, struct pcb_t * proc) {  
2     if((q->size) == MAX_QUEUE_SIZE) return;  
3     (q->proc)[q->size] = proc;  
4     (q->size)++;  
5 }
```

###### 3.1.1.2 Lấy PCB ra khỏi queue bằng hàm dequeue()

- **Ý tưởng:**

Lấy ra PCB đầu tiên trong hàng đợi, sau đó dời tất cả các PCB phía sau xuống 1 đơn vị và giảm size của hàng đợi xuống 1 đơn vị.

- **Code:**



```
1 struct pcb_t * dequeue(struct queue_t * q) {  
2     struct pcb_t * proc = q->proc[0];  
3     int i;  
4     for(i = 0; i < q->size - 1 ; i++) {  
5         (q->proc)[i] = (q->proc)[i + 1];  
6     }  
7     (q->proc)[i] = NULL;  
8     (q->size)--;  
9     return proc;  
10 }
```

###### 3.1.1.3 Lấy PCB thích hợp trong multi level queue đưa vào CPU bằng hàm get\_miq\_proc()

- **Ý tưởng:**

Lần lượt kiểm tra các queue từ queue 0 đến queue 139 (MAX\_PRIO) của multi level queue. Nếu tại một queue vẫn còn time slot chiếm hoạt động của CPU, ta lập tức lấy PCB của process có độ ưu tiên cao nhất tại queue đó ra bằng hàm dequeue(), độ ưu tiên trong cùng một queue sẽ được xét theo thứ tự đi vào queue.

- Nếu sau khi duyệt đến queue có độ ưu tiên thấp nhất mà không lấy ra được process nào. Có nghĩa là ở mỗi queue có thể là không có process tức size=0 hoặc là queue hết lượng time slot thì ta sẽ reset lại lượng time slot cho tất cả các queue của multilevel queue.
- Time slot ở đây sẽ tránh cho việc các process ở các queue với priority thấp bị đợi quá lâu gây ra hiện tượng Starvation.
- **Code:**



```

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

### 3.1.2 Kết quả kiểm thử và biểu đồ Gantt mô tả các quá trình được thực thi bởi CPU

#### 3.1.2.1 Kiểm thử test case sched\_0

- **Test case sched\_0:**

input > ≡ os_mlq_1			
1	2	4	8
2	1	p0	139
3	2	s3	39
4	4	m1	15
5	6	s2	120
6	7	m0	120
7	9	p1	15
8	11	s0	38
9	16	s1	0
10			

- **Kết quả kiểm thử:**

```

Time slot  0
    Loaded a process at input/proc/p0, PID: 1 PRIO: 139
    CPU 2: Dispatched process  1
Time slot  1
Time slot  2
    Loaded a process at input/proc/s3, PID: 2 PRIO: 39
Time slot  3
    CPU 2: Put process  1 to run queue
    CPU 2: Dispatched process  1
    CPU 0: Dispatched process  2
Time slot  4
    Loaded a process at input/proc/m1, PID: 3 PRIO: 15
Time slot  5
    CPU 3: Dispatched process  3
    CPU 0: Put process  2 to run queue
    CPU 0: Dispatched process  2
    CPU 2: Put process  1 to run queue
    CPU 2: Dispatched process  1
Time slot  6
    Loaded a process at input/proc/s2, PID: 4 PRIO: 120
Time slot  7
    CPU 3: Put process  3 to run queue
    CPU 3: Dispatched process  3
    CPU 0: Put process  2 to run queue
    CPU 0: Dispatched process  2
    CPU 1: Dispatched process  4
    CPU 2: Put process  1 to run queue
    Loaded a process at input/proc/m0, PID: 5 PRIO: 120
Time slot  8
    CPU 2: Dispatched process  5
Time slot  9
    CPU 0: Put process  2 to run queue
    CPU 0: Dispatched process  2
    CPU 1: Put process  4 to run queue
    CPU 1: Dispatched process  4
    CPU 3: Put process  3 to run queue
    CPU 3: Dispatched process  3
    Loaded a process at input/proc/p1, PID: 6 PRIO: 15
Time slot  10
    CPU 2: Put process  5 to run queue
    CPU 2: Dispatched process  6
Time slot  11
    CPU 0: Put process  2 to run queue
    CPU 0: Dispatched process  2
    CPU 1: Put process  4 to run queue
    CPU 1: Dispatched process  5
    CPU 3: Put process  3 to run queue
    CPU 3: Dispatched process  3
    Loaded a process at input/proc/s0, PID: 7 PRIO: 38
Time slot  12
    CPU 2: Put process  6 to run queue
    CPU 2: Dispatched process  6

```

```

Time slot 13
    CPU 0: Put process 2 to run queue
    CPU 0: Dispatched process 7
    CPU 1: Put process 5 to run queue
    CPU 1: Dispatched process 2
    CPU 3: Processed 3 has finished
    CPU 3: Dispatched process 4
Time slot 14
    CPU 1: Processed 2 has finished
    CPU 1: Dispatched process 5
    CPU 2: Put process 6 to run queue
    CPU 2: Dispatched process 6
Time slot 15
    CPU 0: Put process 7 to run queue
    CPU 0: Dispatched process 7
    CPU 3: Put process 4 to run queue
    CPU 3: Dispatched process 4
Time slot 16
    CPU 1: Put process 5 to run queue
    CPU 1: Dispatched process 5
    CPU 2: Put process 6 to run queue
    CPU 2: Dispatched process 6
    Loaded a process at input/proc/s1, PID: 8 PRIO: 0
Time slot 17
    CPU 0: Put process 7 to run queue
    CPU 0: Dispatched process 8
    CPU 1: Processed 5 has finished
    CPU 1: Dispatched process 7
    CPU 3: Put process 4 to run queue
    CPU 3: Dispatched process 4
Time slot 18
    CPU 2: Put process 6 to run queue|
    CPU 2: Dispatched process 6
Time slot 19
    CPU 0: Put process 8 to run queue
    CPU 0: Dispatched process 8
    CPU 1: Put process 7 to run queue
    CPU 1: Dispatched process 7
    CPU 3: Put process 4 to run queue
    CPU 3: Dispatched process 4
Time slot 20
    CPU 2: Processed 6 has finished
    CPU 2: Dispatched process 1
Time slot 21
    CPU 0: Put process 8 to run queue
    CPU 0: Dispatched process 8
    CPU 1: Put process 7 to run queue
    CPU 1: Dispatched process 7
    CPU 3: Processed 4 has finished
    CPU 3 stopped
Time slot 22
    CPU 2: Put process 1 to run queue
    CPU 2: Dispatched process 1

```

```

Time slot 23
    CPU 0: Put process 8 to run queue
    CPU 0: Dispatched process 8
    CPU 1: Put process 7 to run queue
    CPU 1: Dispatched process 7
Time slot 24
    CPU 0: Processed 8 has finished
    CPU 0 stopped
    CPU 2: Processed 1 has finished
    CPU 2 stopped
Time slot 25
    CPU 1: Put process 7 to run queue
    CPU 1: Dispatched process 7
Time slot 26
Time slot 27
    CPU 1: Put process 7 to run queue
    CPU 1: Dispatched process 7
Time slot 28
    CPU 1: Processed 7 has finished
    CPU 1 stopped

```

- **Sơ đồ Gantt mô tả các quá trình được hiện thực bởi CPU:**

Time	0	1	2	3	4	5	6	7	8	9	10
CPU 0				P2							
CPU 1								P4	P4	P4	P4
CPU 2	P1	P5	P5	P6							
CPU 3						P3	P3	P3	P3	P3	P3

  

Time	11	12	13	14	15	16	17	18	19	20	21
CPU 0	P2	P2	P7	P7	P7	P7	P8	P8	P8	P8	P8
CPU 1	P5	P5	P2 END	P5	P5	P5	P7	P7	P7	P7	P7
CPU 2	P6	P6	P6	P6	P6	P6	P6	P6	P6 END	P1	P1
CPU 3	P3	P3 END	P4	P4	P4	P4	P4	P4	P4	P4	P4 END

  

Time	22	23	24	25	26	27	28
CPU 0	P8	P8	P8 END				
CPU 1	P7	P7	P7	P7	P7	P7	P7 END
CPU 2	P1	P1	P1 END				
CPU 3							

### 3.1.3 Trả lời câu hỏi

**Câu hỏi:** What is the advantage of using priority queue in comparison with other scheduling algorithms you have learned?

**Trả lời:**

- Tầm quan trọng của mỗi process được xác định tương đối chính xác.Các process quan trọng hơn (có độ ưu tiên cao hơn) sẽ được thực thi trước, làm giảm thiểu thời gian đợi của các process quan trọng. Có thể được điều chỉnh linh hoạt để phù hợp với các yêu cầu của hệ thống, vì độ ưu tiên có thể được gán dựa trên bất kỳ tiêu chí nào

- Phân bổ tài nguyên hiệu quả, vì các process có độ ưu tiên cao có thể tiêu tốn nhiều tài nguyên hơn và giảm xung đột giữa các tác vụ. Hàng đợi ưu tiên đặc biệt hữu ích trong các hệ thống thời gian thực, nơi các tác vụ quan trọng cần được xử lý nhanh chóng và hiệu quả

Advantage	MLQ	SJF	Round Robin	FIFO
Task Prioritization	Xử lí task priority. Có nhiều queue và process có priority sẽ được xếp vào queue tương ứng	Không xử lí task priority. Mà các tác vụ được định thời dài dựa vào burst time	Không xử lí task priority. Round Robin coi các tác vụ như nhau	Không xử lí task priority. FIFO coi các tác vụ như nhau
Flexible Resource Allocation	MLQ giúp phân bổ tài nguyên hiệu quả bằng cách cung cấp nhiều tài nguyên như CPU time cho các tác vụ có độ ưu tiên cao	SJF không hỗ trợ cơ chế phân bổ tài nguyên hiệu quả	Round Robin chỉ đảm bảo mỗi tác vụ hoạt động trong slice time chứ không hỗ trợ việc phân bổ tài nguyên hiệu quả	FIFO không hỗ trợ cơ chế phân bổ tài nguyên hiệu quả
Throughput and response time	Việc ưu tiên các task có high-priority sẽ phân bổ nhiều CPU time cho tác vụ. Dẫn tới các tác vụ quan trọng sẽ được thi nhau chạy. Từ đó throughput và response time cũng được cải thiện	SJF tập trung giảm waiting time. Nên các tác vụ với burst time nhỏ được xử lí nhanh chóng. Nhưng nếu các tác vụ với burst time lớn vào thì sẽ chờ rất lâu dẫn tới ảnh hưởng throughput và response time	Round Robin cung cấp 1 lượng thời gian time slice đều cho các tác vụ. Do vậy, throughput và response time là tương đối	FIFO thực thi các tác vụ dựa vào thời gian arrival. Do đó các tác vụ tới sớm sẽ được thực thi sớm, còn nếu các tác vụ tới trễ mà có burstime lớn sẽ có thể ảnh hưởng đến throughput và response time

## 3.2 Module 2 - Paging-based memory management

### 3.2.1 Implementation

#### 3.2.1.1 Xây dựng giải thuật thay trang FIFO bằng hàm `find_victim_page()`

- Ý tưởng:**

Trả về page number cuối cùng trong danh sách fifo\_pgn trong mm\_struct \*mm của một process vì page vào sớm nhất sẽ nằm cuối danh sách theo cơ chế FIFO.

- Code:**



```

1  /*find_victim_page - find victim page
2   *@caller: caller
3   *@pgn: return page number
4   *
5   */
6  int find_victim_page(struct mm_struct *mm, int *retpgn)
7  {
8      struct pgn_t *pg = mm->fifo_pgn;
9      /* TODO: Implement the theoretical mechanism to find the victim page */
10     struct pgn_t *prev_pg = NULL;
11
12     if (pg == NULL) return -1;
13
14     while (pg->pg_next != NULL) {
15         prev_pg = pg;
16         pg = pg->pg_next;
17     }
18
19     if (pg == NULL) return -1;
20
21     *retpgn = pg->pgn;
22
23     if (prev_pg != NULL) {
24         prev_pg->pg_next = NULL;
25     } else {
26         mm->fifo_pgn = NULL;
27     }
28     free(pg);
29     pg = NULL;
30     return 0;
31 }

```

### 3.2.1.2 Tìm frame trong RAM bằng hàm pg\_getpage()

- **Ý tưởng:**

Đầu tiên, ta kiểm tra frame đó có trong page table hay không thông qua page entry. Nếu frame đó có trong RAM, ta trả về giá trị của frame number (fpn) tương ứng với page đó và kết thúc việc thực thi hàm. Nếu frame đó không có trong RAM, ta thực hiện các bước sau để lấy được frame từ SWAP vào RAM:

1. Tìm target frame ứng với page đó trong bộ nhớ SWAP.
2. Tìm một victim page trong page table rồi tìm victim frame tương ứng trong RAM để thực hiện việc swap in/swap out.
3. Tìm một frame trống trong bộ nhớ SWAP, sau đó copy giá trị của victim frame cho nó (swap out).
4. Copy giá trị của target frame cho victim frame (swap in).
5. Cập nhật lại page table, trả về giá trị của frame number (fpn) tương ứng với page đó và kết thúc việc thực thi hàm.

- **Code:**



```

1  /*pg_getpage - get the page in ram
2   *@mm: memory region
3   *@pagenum: PGN
4   *@framenum: return FPN
5   *@caller: caller
6   *
7   */
8  int pg_getpage(struct mm_struct *mm, int pgn, int *fpn, struct pcb_t *caller)
9  {
10     uint32_t pte = mm->pgd[pgn];
11
12     if (pte&PAGING_PTE_SWAPPED_MASK)
13     { /* Page is not online, make it actively living */
14         int vicpgn, swpfpn;
15         int vicfpn;
16         uint32_t vicpte;
17
18         int tgtfpn = PAGING_SWP(pte); //the target frame storing our variable
19
20         /* TODO: Play with your paging theory here */
21         /* Find victim page */
22         find_victim_page(caller->mm, &vicpgn);
23         vicpte = caller->mm->pgd[vicpgn];
24         vicfpn = PAGING_FPN(vicpte);
25
26         /* Get free frame in MEMSWP */
27         MEMPHY_get_freefp(caller->active_mswp, &swpfpn);
28
29         /* Do swap frame from MEMRAM to MEMSWP and vice versa*/
30         /* Copy victim frame to swap */
31         __swap_cp_page(caller->mram , vicfpn , caller->active_mswp , swpfpn);
32
33         /* Copy target frame from swap to mem */
34         __swap_cp_page(caller->active_mswp , tgtfpn , caller->mram , vicfpn);
35
36         /* Update page table */
37         pte_set_swap(&(mm->pgd[vicpgn]), 0 , swpfpn);
38
39         /* Update its online status of the target page */
40         pte_set_fpn(&(mm->pgd[pgn]), vicfpn);
41         // pte_set_fpn(&pte, tgtfpn);
42
43         enlist_pgn_node(&caller->mm->fifo_pgn,pgn);
44     }
45
46     *fpn = PAGING_FPN(pte);
47
48     return 0;
49 }
```

### 3.2.1.3 Cấp phát memory bằng hàm \_\_alloc()

- **Ý tưởng:**

Đầu tiên kiểm tra xem có tồn tại vùng nhớ trống trong vma, tức là kiểm tra xem danh sách free\_region có thể đáp ứng được kích thước cần alloc. Nếu có thì cấp phát vùng nhớ đó cho process

Nếu không có các free\_region trong vma không thể đáp ứng cho việc cấp phát. Ta sẽ tiến hành cấp phát một region mới có thể đáp ứng được kích thước cần alloc.

Trường hợp vùng nhớ từ sbrk đến end của vma lớn hơn size cần cấp phát thì ta nâng sbrk lên và dùng vùng nhớ nay cấp phát cho process. Nếu không thỏa thì sẽ tiến hành mở rộng giới hạn của khu vực vùng nhớ vma, sẽ xảy ra hai trường hợp và được xử lý trong hàm inc\_vma\_limit.

- **Code:**

```
/*__alloc - allocate a region memory
 *@caller: caller
 *@vmaid: ID vm area to alloc memory region
 *@rgid: memory region ID (used to identify variable in symbole table)
 *@size: allocated size
 *@alloc_addr: address of allocated memory region
 */
int __alloc(struct pcb_t *caller, int vmaid, int rgid, int size, int *alloc_addr)
{
    /*Allocate at the top of the vma */
    struct vm_rg_struct rgnode;

    if (get_free_vmrng_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;

        printf("===== ALLOC for Process %d - Region: %d - size: %d Byte >>>\n", caller->pid, rgid, size);
        printf("=====PAGE TABLE AND FREE_RG LIST CONTENT=====n");
        print_pttbl(caller, 0, -1);
        print_list_vma(caller->mm->mmap);
        print_list_rg(caller->mm->mmap->vm_freerg_list);

        return 0;
    } // this if statement is ok (pick a fit free_region in vma / reuse region in vma).

    /* TODO get_free_vmrng_area FAILED handle the region management (Fig.6)*/
    /*Attempt to increase limit to get space */
    //int inc_limit_ret;

    struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);
    // int inc_sz = PAGING_PAGE_ALIGN_SIZE(size);
    int old_sbrk ;
    old_sbrk = cur_vma->sbrk;
```

```

// BEGIN MY CODE
if(old_sbrk + size <= cur_vma->vm_end) {
    caller->mm->symrgtbl[rgid].rg_start = old_sbrk;
    caller->mm->symrgtbl[rgid].rg_end = old_sbrk + size;
    cur_vma->sbrk += size;

    *alloc_addr = old_sbrk;

    printf("<<< ALLOC for Process %d - Region: %d - size: %d Byte >>>\n", caller->pid, rgid, size);
    printf("=====PAGE TABLE AND FREE_RG LIST CONTENT=====\\n");
    print_pgtbl(caller, 0, -1);
    print_list_vma(caller->mm->mmap);
    print_list_rg(caller->mm->mmap->vm_freerg_list);

    return 0;
} // this is statement handle case [sbrk, end] > size in vma. Alloc new region right here, this region has been mapped.
// END MY CODE

/* TODO INCREASE THE LIMIT */ // I change parameter inc_sz => size in this function call
inc_vma_limit(caller, vmaid, size);

/*Successful increase limit */
caller->mm->symrgtbl[rgid].rg_start = old_sbrk;
caller->mm->symrgtbl[rgid].rg_end = old_sbrk + size;

*alloc_addr = old_sbrk;

printf("<<< ALLOC for Process %d - Region: %d - size: %d Byte >>>\n", caller->pid, rgid, size);
printf("=====PAGE TABLE AND FREE_RG LIST CONTENT=====\\n");
print_pgtbl(caller, 0, -1);
print_list_vma(caller->mm->mmap);
print_list_rg(caller->mm->mmap->vm_freerg_list);

return 0;

```

### 3.2.1.6 Cấp phát frame được yêu cầu trong bộ nhớ RAM hoặc SWAP bằng hàm alloc\_pages\_range()

- **Ý tưởng:**

Ta sẽ tạo ra một region mới số lượng page có thể đáp ứng đủ cho size cần alloc và yêu cầu số lượng frame tương ứng. Hàm này sẽ tìm các free\_frame cho việc mapping giữa logical memory với physical memory.

Sẽ có hai trường hợp:

- Trong Ram có đủ free\_frame để dùng cho việc mapping với các page mới được tạo ra.
- Nếu Ram không đủ free\_frame thì ta sẽ tiến hành cơ chế swap out các frame trong RAM vào SWAP và lấy các free\_frame của SWAP đưa lên RAM để tiến hành mapping với các page mới.

- **code**

```

int alloc_pages_range(struct pcb_t *caller, int req_pnum, struct framephy_struct** frm_lst)
{
    int pgit, fpn;
    struct framephy_struct *newfp_str;

    for(pgit = 0; pgit < req_pnum; pgit++)
    {
        if(MEMPHY_get_freefp(caller->mram, &fpn) == 0)
        {
            // BEGIN MY CODE
            newfp_str = malloc(sizeof(struct framephy_struct));
            newfp_str->fpn = fpn;
            newfp_str->owner = caller->mm;
            newfp_str->fp_next = NULL;

            newfp_str->fp_next = *frm_lst;
            *frm_lst = newfp_str;
            // END MY CODE

        } else { // ERROR CODE of obtaining somes but not enough frames

            // BEGIN MY CODE
            int victim_pgn, victim_fp, swp_free_fp;
            uint32_t victim_pte;

            if(find_victim_page(caller->mm, &victim_pgn) != 0) return -1;
            victim_pte = caller->mm->pgd[victim_pgn];
            victim_fp = PAGING_FPN(victim_pte);

            // swap free frame in Swap to Ram = copy data fram in Ram
            // to free frame in Swap. => frame in Ram is free
            if(MEMPHY_get_freefp(caller->active_mswp, &swp_free_fp) != 0) return -3000;

            __swap_cp_page(caller->mram, victim_fp, caller->active_mswp, swp_free_fp);

            pte_set_swap(&(caller->mm->pgd[victim_pgn]), 0, swp_free_fp);

            newfp_str = malloc(sizeof(struct framephy_struct));
            newfp_str->fpn = victim_fp;
            newfp_str->owner = caller->mm;
            newfp_str->fp_next = NULL;

            newfp_str->fp_next = *frm_lst;
            *frm_lst = newfp_str;
            // END MY CODE

        }
    }
}

```

### 3.2.1.5 Gán frame vào page table bằng hàm vmap\_pages\_range()

- **Ý tưởng:**

Sau khi có được danh sách các frame có thể mapping cho page. Ta sẽ tiến gần từng frame vào page table entry tương ứng của các page bằng việc thiết lập lại các bit của pte sao cho 0-12 bit đầu của pte thể hiện frame number, sau đó ta sẽ set các page mới nào vào danh sách fifo\_pgn.

- **Code:**

```

int vmap_page_range(struct pcb_t *caller, // process call
                    int addr, // start address which is aligned to pagesz
                    int pnum, // num of mapping page
                    struct framephy_struct *frames, // list of the mapped frames
                    struct vm_rg_struct *ret_rg) // return mapped region, the real mapped fp
{
    // no guarantee all given pages are mapped
    // uint32_t * pte = malloc(sizeof(uint32_t));
    struct framephy_struct *fpit = NULL;
    // int fpn;
    int pgit = 0;
    int pgn = PAGING_PGN(addr);

    ret_rg->rg_end = ret_rg->rg_start = addr; // at least the very first space is usable

    fpit = frames;

    /* TODO map range of frame to address space
     *      [addr to addr + pnum*PAGING_PAGESZ
     *      in page table caller->mm->pgd[]
     */
    /* Tracking for later page replacement activities (if needed)
     * Enqueue new usage page */

    // BEGIN MY CODE
    for (pgit = 0; pgit < pnum; pgit++) {
        pte_set_fp(&(caller->mm->pgd[pgn + pgit]), fpit->fpn);
        fpit = fpit->fp_next;
        enlist_pgn_node(&caller->mm->fifo_pgn, pgn + pgit);
    }

    ret_rg->rg_end += pgit*PAGING_PAGESZ;
    // END MY CODE

    return 0;
}

```

### 3.2.1.4 Thu hồi memory bằng hàm \_\_free()

- **Ý tưởng:**

Đầu tiên kiểm tra xem có tồn tại vùng nhớ có vmaid và rgid đó không

Ta sẽ lấy ra được vùng region cần thu hồi từ rgid, sau đó sau đó lưu vùng region này vào con trỏ rgnode. Và tiến hành thu hồi bằng cách thiết lập vùng nhớ start và end của vùng nhớ đó bằng 0. Ở đây theo hướng dẫn của bài tập lớn để đơn giản, chúng ta sẽ không cần tác động gì đến vùng nhớ vật lý.

Ta sẽ đưa vùng region vừa mới thu hồi vào danh sách các free\_region của process.

- **Code:**

```

int __free(struct pcb_t *caller, int vmaid, int rgid)
{
    struct vm_rg_struct *rgnode=malloc(sizeof(struct vm_rg_struct));

    if(rgid < 0 || rgid > PAGING_MAX_SYMTBL_SZ)
        return -1;
    /* TODO: Manage the collect freed region to freerg_list */

    // BEGIN MY CODE
    struct vm_rg_struct *currng = get_symrg_byid(caller->mm, rgid);
    struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);

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

    // assign values for rgnode so that it can enlist to freeList
    rgnode->rg_start = caller->mm->symrgtbl[rgid].rg_start;
    rgnode->rg_end = caller->mm->symrgtbl[rgid].rg_end;
    rgnode->size = rgnode->rg_end - rgnode->rg_start;

    // free the freed region
    caller->mm->symrgtbl[rgid].rg_start = 0;
    caller->mm->symrgtbl[rgid].rg_end = 0;
    caller->mm->symrgtbl[rgid].rg_next = NULL;

    enlist_vm_freerg_list(caller->mm, rgnode);

    printf("<<< FREE for Process %d: - Region: %ld free range [%ld -%ld] >>>\n",
           caller->pid, rgid, rgnode->rg_start, rgnode->rg_end);
    printf("=====PAGE TABLE AND FREE_RG LIST CONTENT=====\\n");
    print_pgtbl(caller, 0, -1);
    print_list_vma(caller->mm->mmap);
    print_list_rg(caller->mm->mmap->vm_freerg_list);
    // END MY CODE

    return 0;
}

```

### 3.2.2 Kết quả kiểm thử mô tả các quá trình được thực thi bởi CPU

#### 3.2.2.1 Kiểm thử test case os\_0\_mlq.paging

- **Test case os\_0\_mlq.paging:**

1	2	2	2		
2	1048576	16777216	0	0	0
3	0	p0s	0		
4	2	p1s	15		

- **Kết quả kiểm thử:**

```

Time slot 0
ld_routine
Loaded a process at input/proc/p0s, PID: 1 PRIO: 0
Time slot 1
CPU 1: Dispatched process 1
Time slot 2
Loaded a process at input/proc/p1s, PID: 2 PRIO: 15
<<< ALLOC for Process 1 - Region: 0 - size: 300 Byte >>>
=====PAGE TABLE AND FREE_RG LIST CONTENT=====
print_pgtbl: 0 - 512
00000000: 80000001
00000004: 80000000
print_list_vma:
va[0->512]
print_list_rg:
rg[0->0]
CPU 0: Dispatched process 2
Time slot 3
CPU 1: Put process 1 to run queue
CPU 1: Dispatched process 1
<<< ALLOC for Process 1 - Region: 4 - size: 300 Byte >>>
=====PAGE TABLE AND FREE_RG LIST CONTENT=====
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
print_list_vma:
va[0->1024]
print_list_rg:
rg[0->0]
Time slot 4
<<< FREE for Process 1: - Region: 0 free range [0 -300] >>>
=====PAGE TABLE AND FREE_RG LIST CONTENT=====
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
print_list_vma:
va[0->1024]
print_list_rg:
rg[0->0]
rg[0->300]
CPU 0: Put process 2 to run queue
CPU 0: Dispatched process 2
Time slot 5
CPU 1: Put process 1 to run queue
CPU 1: Dispatched process 1
<<< ALLOC for Process 1 - Region: 1 - size: 100 Byte >>>
=====PAGE TABLE AND FREE_RG LIST CONTENT=====
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
print_list_vma:
va[0->1024]
print_list_rg:
rg[0->0]
```

```
rg[100->300]
Time slot 6
<<< Process 1 - WRITE region=1 offset=20 value=100 >>>
=====PAGE TABLE=====
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
=====MEMORY CONTENT=====
Address: Content
0x00000114: 00000064
CPU 0: Put process 2 to run queue
CPU 0: Dispatched process 2
Time slot 7
CPU 1: Put process 1 to run queue
CPU 1: Dispatched process 1
<<< Process 1 - READ region=1 offset=20 value=100 >>>
=====PAGE TABLE=====
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
=====MEMORY CONTENT=====
Address: Content
0x00000114: 00000064
Time slot 8
CPU 0: Put process 2 to run queue
CPU 0: Dispatched process 2
<<< Process 1 - WRITE region=4 offset=20 value=102 >>>
=====PAGE TABLE=====
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
=====MEMORY CONTENT=====
Address: Content
0x00000040: 00000066
0x00000114: 00000064
Time slot 9
CPU 1: Put process 1 to run queue
CPU 1: Dispatched process 1
<<< Process 1 - READ region=4 offset=20 value=102 >>>
=====PAGE TABLE=====
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
=====MEMORY CONTENT=====
Address: Content
0x00000040: 00000066
0x00000114: 00000064
Time slot 10
<<< Process 1 - WRITE region=4 offset=30 value=103 >>>
=====PAGE TABLE=====
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
=====MEMORY CONTENT=====
```

```

Address: Content
0x00000040: 00000066
0x0000004a: 00000067
0x00000114: 00000064
CPU 0: Put process 2 to run queue
CPU 0: Dispatched process 2
Time slot 11
CPU 1: Put process 1 to run queue
CPU 1: Dispatched process 1
<<< Process 1 - READ region=4 offset=30 value=103 >>>
=====PAGE TABLE=====
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
=====MEMORY CONTENT=====
Address: Content
0x00000040: 00000066
0x0000004a: 00000067
0x00000114: 00000064
Time slot 12
CPU 0: Processed 2 has finished
CPU 0 stopped
Time slot 13
CPU 1: Put process 1 to run queue
CPU 1: Dispatched process 1
<<< FREE for Process 1: - Region: 4 free range [300 -600] >>>
=====PAGE TABLE AND FREE_RG LIST CONTENT=====
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
print_list_vma:
va[0->1024]
print_list_rg:
rg[0->0]
rg[300->600]
rg[100->300]
Time slot 14
Time slot 15
CPU 1: Processed 1 has finished
CPU 1 stopped

```

### 3.2.2.2 Kiểm thử test case os\_1\_mlq.paging\_small\_4K

- Test case os\_1\_mlq.paging\_small\_4K:

1	2	4	8
2	4096	16777216	0 0 0
3	1	p0s	130
4	2	s3	39
5	4	m1s	15
6	6	s2	120
7	7	m0s	120
8	9	p1s	15
9	11	s0	38
<b>10</b>	<b>16</b>	<b>s1</b>	<b>0</b>

- **Kết quả kiểm thử:**

```

Time slot 0
Id_routine
Time slot 1
Loaded a process at input/proc/p0s, PID: 1 PRIO: 130
CPU 2: Dispatched process 1
Time slot 2
<<< ALLOC for Process 1 - Region: 0 - size: 300 Byte >>>
=====PAGE TABLE AND FREE_RG LIST CONTENT=====
print_pgtbl: 0 - 512
00000000: 80000001
00000004: 80000000
print_list_vma:
va[0->512]
print_list_rg:
rg[0->0]
Time slot 3
CPU 2: Put process 1 to run queue
CPU 2: Dispatched process 1
<<< ALLOC for Process 1 - Region: 4 - size: 300 Byte >>>
=====PAGE TABLE AND FREE_RG LIST CONTENT=====
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
print_list_vma:
va[0->1024]
print_list_rg:
rg[0->0]
Time slot 4
Loaded a process at input/proc/m1s, PID: 2 PRIO: 15
<<< FREE for Process 1: - Region: 0 free range [0 -300] >>>
=====PAGE TABLE AND FREE_RG LIST CONTENT=====
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
print_list_vma:
va[0->1024]
print_list_rg:
rg[0->0]
rg[0->300]
Time slot 5
CPU 1: Dispatched process 2
<<< ALLOC for Process 2 - Region: 0 - size: 300 Byte >>>
=====PAGE TABLE AND FREE_RG LIST CONTENT=====
print_pgtbl: 0 - 512
00000000: 80000005
00000004: 80000004
print_list_vma:
va[0->512]
print_list_rg:
rg[0->0]
CPU 2: Put process 1 to run queue
CPU 2: Dispatched process 1
<<< ALLOC for Process 1 - Region: 1 - size: 100 Byte >>>
=====PAGE TABLE AND FREE_RG LIST CONTENT=====
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003

```

```
00000012: 80000002
print_list_vma:
va[0->1024]
print_list_rg:
rg[0->0]
rg[100->300]
Time slot 6
Loaded a process at input/proc/s2, PID: 3 PRIO: 120
<<< Process 1 - WRITE region=1 offset=20 value=100 >>>
=====PAGE TABLE=====
CPU 0: Dispatched process 3
<<< ALLOC for Process 2 - Region: 1 - size: 100 Byte >>>
=====PAGE TABLE AND FREE_RG LIST CONTENT=====
print_pgtbl: 0 - 512
00000000: 80000005
00000004: 80000004
print_list_vma:
va[0->512]
print_list_rg:
rg[0->0]
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
=====MEMORY CONTENT=====
Address: Content
0x00000114: 00000064
Time slot 7
CPU 1: Put process 2 to run queue
CPU 1: Dispatched process 2
<<< FREE for Process 2: - Region: 0 free range [0 -300] >>>
=====PAGE TABLE AND FREE_RG LIST CONTENT=====
print_pgtbl: 0 - 512
00000000: 80000005
00000004: 80000004
print_list_vma:
va[0->512]
print_list_rg:
rg[0->0]
rg[0->300]
CPU 2: Put process 1 to run queue
CPU 2: Dispatched process 1
<<< Process 1 - READ region=1 offset=20 value=100 >>>
=====PAGE TABLE=====
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
=====MEMORY CONTENT=====
Address: Content
0x00000114: 00000064
Loaded a process at input/proc/m0s, PID: 4 PRIO: 120
Time slot 8
<<< ALLOC for Process 2 - Region: 2 - size: 100 Byte >>>
=====PAGE TABLE AND FREE_RG LIST CONTENT=====nprint_pgtbl: 0 -
512
00000000: 80000005
00000004: 80000004
print_list_vma:
va[0->512]
print_list_rg:
rg[0->0]
```

```
rg[100->300]
CPU 0: Put process 3 to run queue
CPU 0: Dispatched process 4
<<< Process 1 - WRITE region=4 offset=20 value=102 >>>
=====PAGE TABLE=====
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
=====MEMORY CONTENT=====
<<< ALLOC for Process 4 - Region: 0 - size: 300 Byte >>>
=====PAGE TABLE AND FREE_RG LIST CONTENT=====
print_pgtbl: 0 - 512
00000000: 80000007
00000004: 80000006
print_list_vma:
va[0->512]
print_list_rg:
rg[0->0]
Address: Content
0x00000040: 00000066
0x00000114: 00000064
Time slot 9
Loaded a process at input/proc/p1s, PID: 5 PRIO: 15
CPU 2: Put process 1 to run queue
CPU 2: Dispatched process 5
CPU 1: Put process 2 to run queue
CPU 1: Dispatched process 2
<<< FREE for Process 2: - Region: 2 free range [0 -100] >>>
=====PAGE TABLE AND FREE_RG LIST CONTENT=====
print_pgtbl: 0 - 512
00000000: 80000005
00000004: 80000004
print_list_vma:
va[0->512]
print_list_rg:
rg[0->0]
rg[0->100]
rg[100->300]
<<< ALLOC for Process 4 - Region: 1 - size: 100 Byte >>>
=====PAGE TABLE AND FREE_RG LIST CONTENT=====
print_pgtbl: 0 - 512
00000000: 80000007
00000004: 80000006
print_list_vma:
va[0->512]
print_list_rg:
rg[0->0]
Time slot 10
<<< FREE for Process 2: - Region: 1 free range [300 -400] >>>
=====PAGE TABLE AND FREE_RG LIST CONTENT=====
print_pgtbl: 0 - 512
00000000: 80000005
00000004: 80000004
print_list_vma:
va[0->512]
print_list_rg:
rg[0->0]
rg[300->400]
rg[0->100]
rg[100->300]
```

CPU 0: Put process 4 to run queue  
CPU 0: Dispatched process 3

```
Time slot 11
CPU 1: Processed 2 has finished
CPU 1: Dispatched process 4
<<< FREE for Process 4: - Region: 0 free range [0 -300] >>>
=====PAGE TABLE AND FREE_RG LIST CONTENT=====
print_pgtbl: 0 - 512
00000000: 80000007
00000004: 80000006
print_list_vma:
va[0->512]
print_list_rg:
rg[0->0]
rg[0->300]
CPU 2: Put process 5 to run queue
CPU 2: Dispatched process 5
Time slot 12
<<< ALLOC for Process 4 - Region: 2 - size: 100 Byte >>>
=====PAGE TABLE AND FREE_RG LIST CONTENT=====nprint_pgtbl: 0 -
512
00000000: 80000007
00000004: 80000006
print_list_vma:
va[0->512]
print_list_rg:
rg[0->0]
rg[100->300]
CPU 0: Put process 3 to run queue
CPU 0: Dispatched process 3
Time slot 13
CPU 1: Put process 4 to run queue
CPU 1: Dispatched process 4
<<< Process 4 - WRITE region=1 offset=20 value=102 >>>
=====PAGE TABLE=====
print_pgtbl: 0 - 512
00000000: 80000007
00000004: 80000006
=====MEMORY CONTENT=====
Address: Content
0x00000040: 00000066
0x00000114: 00000064
0x00000640: 00000066
CPU 2: Put process 5 to run queue
CPU 2: Dispatched process 5
Time slot 14
<<< Process 4 - WRITE region=2 offset=1000 value=1 >>>
=====PAGE TABLE=====
print_pgtbl: 0 - 512
00000000: 80000007
00000004: 80000006
=====MEMORY CONTENT=====
Address: Content
0x00000040: 00000066
0x000000e8: 00000001
0x00000114: 00000064
0x00000640: 00000066
CPU 0: Put process 3 to run queue
CPU 0: Dispatched process 3
Time slot 15
CPU 1: Processed 4 has finished
CPU 1: Dispatched process 1
<<< Process 1 - READ region=4 offset=20 value=102 >>>
=====PAGE TABLE=====
print_pgtbl: 0 - 1024
00000000: 80000001
```

```
00000004: 80000000
00000008: 80000003
00000012: 80000002
=====MEMORY CONTENT=====
Address: Content
0x00000040: 00000066
0x000000e8: 00000001
0x00000114: 00000064
0x00000640: 00000066
CPU 2: Put process 5 to run queue
CPU 2: Dispatched process 5
Time slot 16
<<< Process 1 - WRITE region=4 offset=30 value=103 >>>
=====PAGE TABLE=====
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
=====MEMORY CONTENT=====
Address: Content
0x00000040: 00000066
0x0000004a: 00000067
0x000000e8: 00000001
0x00000114: 00000064
0x00000640: 00000066
CPU 0: Put process 3 to run queue
CPU 0: Dispatched process 3
Time slot 17
CPU 1: Put process 1 to run queue
CPU 1: Dispatched process 1
<<< Process 1 - READ region=4 offset=30 value=103 >>>
=====PAGE TABLE=====
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
=====MEMORY CONTENT=====
Address: Content
0x00000040: 00000066
0x0000004a: 00000067
0x000000e8: 00000001
0x00000114: 00000064
0x00000640: 00000066
CPU 2: Put process 5 to run queue
CPU 2: Dispatched process 5
Time slot 18
CPU 0: Put process 3 to run queue
CPU 0: Dispatched process 3
Time slot 19
CPU 1: Put process 1 to run queue
CPU 1: Dispatched process 1
<<< FREE for Process 1: - Region: 4 free range [300 -600] >>>
=====PAGE TABLE AND FREE_RG LIST CONTENT=====
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
print_list_vma:
va[0->1024]
print_list_rg:
rg[0->0]
```

```

rg[300->600]
rg[100->300]
CPU 2: Processed 5 has finished
CPU 2 stopped
Time slot 20
CPU 0: Processed 3 has finished
CPU 0 stopped
Time slot 21
CPU 1: Processed 1 has finished
CPU 1 stopped

```

### 3.2.3 Trả lời câu hỏi

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

**Trả lời:**

Ưu điểm của thiết kế đa đoạn vùng nhớ trong hệ điều hành đơn giản là:

- Quản lý hiệu quả tài nguyên: Với việc chia nhỏ bộ nhớ thành nhiều đoạn, ta có thể quản lý tài nguyên bộ nhớ một cách hiệu quả hơn. Mỗi đoạn nhớ có thể được dùng cho mục đích cụ thể như lưu trữ mã chương trình, dữ liệu, bảng tham số, vùng nhớ stack và heap. Điều này giúp tăng cường sự sắp xếp, tập trung và tối ưu hóa việc sử dụng bộ nhớ.
- Bảo vệ dữ liệu và an toàn hơn: Bằng cách chia nhỏ bộ nhớ thành các đoạn riêng biệt, ta có thể xác định quyền truy cập của các quy trình và ngăn chặn việc ghi đè hoặc ghi vào bộ nhớ của quy trình khác. Điều này giúp bảo vệ dữ liệu khỏi sự ghi đè không mong muốn và giảm thiểu xung đột hoặc lỗi liên quan đến việc truy cập bộ nhớ.
- Hỗ trợ đa nhiệm: Thiết kế nhiều đoạn nhớ cung cấp khả năng hỗ trợ đa nhiệm, cho phép nhiều quy trình chạy cùng một lúc. Mỗi quy trình có thể được gán một đoạn nhớ riêng, giúp ngăn chặn việc truy cập hoặc ảnh hưởng đến bộ nhớ của các quy trình khác. Điều này tạo điều kiện cho việc chạy song song và chia sẻ tài nguyên một cách an toàn và hiệu quả.
- Tính linh hoạt và mở rộng: Với thiết kế nhiều đoạn nhớ, ta có thể linh hoạt mở rộng hệ thống bằng cách thêm hoặc xóa các đoạn nhớ tùy theo nhu cầu. Điều này cho phép bạn tăng cường khả năng lưu trữ và xử lý dữ liệu mà không ảnh hưởng đến hoạt động của các đoạn nhớ khác.

Tóm lại, thiết kế đa đoạn vùng nhớ trong hệ điều hành đơn giản mang lại nhiều lợi ích như quản lý tài nguyên hiệu quả, bảo vệ dữ liệu, hỗ trợ đa nhiệm và tính linh hoạt mở rộng.

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

**Trả lời:**

**Ưu điểm:**

- Giúp truy xuất bảng phân trang nhanh hơn
- Tính linh hoạt cao hơn trong cách tổ chức không gian bộ nhớ. Điều này đặc biệt hữu ích trong các hệ thống yêu cầu nhiều bộ nhớ khác nhau, vì nó cho phép điều chỉnh các bảng phân trang một cách linh hoạt để đáp ứng các yêu cầu

**Nhược điểm:**

- Gây khó khăn trong việc thiết kế vì phân trang đa cấp làm tăng độ phức tạp cho bộ nhớ
- Tăng chi phí do phải truy xuất nhiều bảng phân trang
- Gây ra phân mảnh không gian bộ nhớ, làm giảm hiệu suất của hệ thống.

**Câu hỏi 3:** What will happen if the synchronization is not handled in your simple OS? Illustrate by example the problem of your simple OS if you have any.

### Trả lời:

Nếu việc đồng bộ hóa không được xử lý trong bài tập lớn hệ điều hành này thì có thể xảy ra các vấn đề liên quan đến đồng bộ hóa dữ liệu và tài nguyên. Dưới đây là một ví dụ minh họa về vấn đề có thể xảy ra trong hệ điều hành nếu không có việc đồng bộ hóa.

Trong phần Memory Management: MEMPHY\_read(), MEMPHY\_write() sử dụng cơ chế synchronization bằng việc áp dụng pthread\_mutex\_lock

Giả sử hệ điều hành cho phép nhiều tiến trình chạy đồng thời trên nhiều cores. Cả hai tiến trình A và B đều muốn thực hiện chức năng Write và Read dữ liệu trên 1 biến dùng chung là 'total' => A và B truy xuất cùng địa chỉ vật lý biến này

Tiến trình A:

1. Đọc giá trị hiện tại của "total".
2. Tăng giá trị đọc được lên một đơn vị.
3. Ghi giá trị mới vào "total".

Tiến trình B:

1. Đọc giá trị hiện tại của "total".
2. Tăng giá trị đọc được lên một đơn vị.
3. Ghi giá trị mới vào "total".

Với việc không có việc đồng bộ hóa, các bước trên có thể xảy ra đồng thời và song song giữa hai tiến trình. Giả sử ban đầu giá trị của "total" là 0.

Trong một tình huống, hai tiến trình A và B đều cùng thực hiện các bước đầu tiên của quá trình của mình đồng thời:

Tiến trình A đọc giá trị hiện tại của "total" (0) và tăng giá trị đọc được lên một đơn vị, trở thành 1. Trong khi đó, tiến trình B cũng đọc giá trị hiện tại của "total" (0) và tăng giá trị đọc được lên một đơn vị, trở thành 1. Sau đó, cả hai tiến trình đều ghi giá trị mới (1) vào "total". Do đó, kết quả cuối cùng của "total" là 1, trong khi đúng lẽ ra nó nên là 2.

Vấn đề này xảy ra do việc không có quy tắc đồng bộ hóa giữa các tiến trình. Khi hai tiến trình đọc và chỉnh sửa cùng một biến toàn cục mà không có sự đồng bộ hóa, sự tương đồng và tính nhất quán của dữ liệu không được đảm bảo.

Để giải quyết vấn đề này, hệ điều hành phải cung cấp các cơ chế đồng bộ hóa như mutex, semaphore hoặc các phương pháp khác để đảm bảo rằng chỉ một tiến trình có thể truy cập vào biến toàn cục tại một thời điểm và đảm bảo tính đúng đắn của kết quả.

## 3.3 Module 3 - Put It All Together

### 3.3.1 Mutex lock

#### 3.3.1.1 Mutex lock trong Scheduler

CPU thực hiện tranh chấp trong hàng đợi ready\_queue để lấy process ra từ hàng đợi và đưa vào thực thi

- **Tranh chấp ở hàm get\_mlq\_proc() và hàm decrease\_slot():** Khi các CPU đồng thời gọi hàm này sẽ xảy ra tranh chấp process lấy ra để thực thi trên CPU, vì thế để bảo vệ ta dùng mutex lock ở đầu và cuối 2 hàm.
- **Tranh chấp ở 2 hàm put\_mlq\_proc() và add\_mlq\_proc():** Hai hàm này đã được sử dụng sẵn mutex lock trong source code.

#### 3.3.1.2 Mutex lock trong Memory management

Các CPU sẽ thực hiện tranh chấp nhau chủ yếu ở các phần có liên quan đến memory. Các hàm có khả năng sẽ có tranh chấp là MEMPHY\_mv\_csr(), MEMPHY\_seq\_read(), MEMPHY\_read(), MEMPHY\_seq\_write(), MEMPHY\_write(), MEMPHY\_get\_freefp(), MEMPHY\_put\_freefp(). Do đó ta dùng mutex lock ở đầu và cuối để bảo vệ các hàm.

**CHECKLIST** We leverage some materials in guidelines, but as usual, this section has a lot of material intensively. Try to keep it not too long

Checklists	Y/N
What was done? Address question such as:  With the available resource, we design this, implement that  With the meeting, we minus/refine the work target  Describe the activities supported under the project  Describe the timeline or scheduling	Y Y Y Y
What was learned about the implementation and management of the projects' activities?	Y
Design and implementation (3-4 pages, try to keep it less than 6 pages)	Y
Simulation, testing or evaluation (3-4 pages should be balance with the previous section)  Does this experiment consist of objective statements? Double check.	Y

Did it determine participants' role and contributions  Who is coordinator, design the plan, integrated need to verify and design test/evaluation method  Who is the architecture design should provide proof-of-concept or at least prototype working code  Who is the main developer to implement the details and fix bug  Who is quality analyst design the sanity test, the module test or self-test and test scenario	Y
---	---

# 4 PARTICIPANT AND PROJECT OUTPUTS

## 4.1 Role and contribution of each member 1

Nhóm trưởng: Nguyễn Phạm Thiên Phúc

Đóng góp của các thành viên:

- Nguyễn Phạm Thiên Phúc: thiết kế phần định thời, hiện thực các cơ chế cấp phát và thu hồi của memory
- Trần Minh Hiếu: thiết kế phần định thời, hiện thực các cơ chế cấp phát và thu hồi của memory
- Trương Thuận Hưng: thiết kế phần định thời, giải thuật thay trang, hàm pg\_getpage(), viết báo cáo
- Nguyễn Hoàng Kim: thiết kế phần định thời, giải thuật thay trang, hàm pg\_getpage(), viết báo cáo

Qua các buổi họp hàng tuần, nhóm đã giải quyết được các vấn đề đã đề ra ở buổi họp đầu tiên, làm rõ được các điểm lí thuyết liên quan như bộ định thời, quản lý bộ nhớ cũng như cơ chế mutex lock, hiểu rõ code được thực thi như thế nào, đồng thời phát triển thêm các hàm cần thiết đối với quá trình hiện thực. Qua đó, từng thành viên trong nhóm đã hiện thực được bài tập lớn và chạy thử kết quả, đối chiếu với output được thầy cung cấp.

Checklists	Y/N
What was done? Address question such as:  With the available resource, we design this, implement that  With the meeting, we minus/refine the work target  Describe the activities supported under the project  Describe the timeline or scheduling	Y Y Y Y
What was learned about the implementation and management of the projects' activities?	Y
Design and implementation (3-4 pages, try to keep it less than 6 pages)	Y
Simulation, testing or evaluation (3-4 pages should be balance with the previous section)  Does this experiment consist of objective statements? Double check.	Y

Did it determine participants' role and contributions	Y
Who is coordinator, design the plan, integrated need to verify and design test/evaluation method	Y
Who is the architecture design should provide proof-of-concept or at least prototype working code	Y
Who is the main developer to implement the details and fix bug	Y
Who is quality analyst design the sanity test, the module test or self-test and test scenario	Y

## 4.2 Project output

Các output của OS được đánh giá là đúng theo với lý thuyết của yêu cầu và nhóm đề ra.

### 4.1.1 Sản phẩm

Hệ điều hành nhận vào các process, sau đó chọn process thông qua bộ định thời theo cơ chế multilevel queue, sau đó cấp phát, thay đổi và thu hồi các vùng nhớ nếu cần thiết.

### 4.1.2 API

#### 4.1.2.1 Schedule

- enqueue(struct queue\_t \* q, struct pcb\_t \* proc): thêm proc vào đầu hàng chờ q
- dequeue(struct queue\_t \* q): lấy ra proc đầu tiên của hàng chờ q
- get\_proc\_mlq(void): lấy ra process cần thực hiện tiếp theo theo cơ chế multilevel queue

#### 4.1.2.2 Memory management

- \_\_alloc(struct pcb\_t \*caller, int vmaid, int rgid, int size, int \*alloc\_addr): cấp phát vùng nhớ tại virtual memory
- validate\_overlap\_vm\_area(struct pcb\_t \*caller, int vmaid, int vmastart, int vmaend): kiểm tra nếu có xung đột giữa vm\_area\_struct hiện tại với các vm\_area\_struct đã tạo
- alloc\_pages\_range(struct pcb\_t \*caller, int incpgnum, struct framephy\_struct \*\*frm\_lst): cấp phát trang được yêu cầu của frame trong RAM
- vmap\_page\_range(struct pcb\_t \*caller, int addr, int pgnum, struct framephy\_struct \*frames, struct vm\_rg\_struct \*ret\_rg): gắn frame vào page table
- \_\_free(struct pcb\_t \*caller, int vmaid, int rgid): thu hồi vùng nhớ cấp phát
- pg\_getpage(struct mm\_struct \*mm, int pgn, int \*fpn, struct pcb\_t \*caller): trả về frame number tương ứng với page

Checklists	Y/N
Directly achievable product of project's completed activities  Standalone standards: Protocol, rule, procedure, algorithm, guidelines  Software product: program/application, libraries, APIs	Y Y
Verify the matching of <u>these outputs</u> with the project's <u>completed</u> activities?  Is it an over claim?  Is there any completed activities without nothing (output), if yes, re-evaluate: we did it unfinished or /  lost focus on the objectives or problematic planning?	N N
What were the <b>main output</b> of the project?  Identify any output that were planned but which has not materialised  Specify when these output will be completed, if it falls into out-of-scope including plan for any future publications (if possible)	Y Y
What were the main specific <b>achievements</b> or <b>practice influence</b>  What was learned about the production or realisation of the project.  What contributed to these output and what lessons you may draw from your experiences	Y Y
If appropriate, highlight any unique or innovative outputs  If appropriate, explain why output were not completed or were poor of quality	Y

### 4.3 Project outcome

Các chuẩn đầu ra mà bài tập lớn của nhóm đã đạt được là: L.O.1.1, L.O.2.1, L.O.2.2, L.O.3.1



[www.agriterra.org](http://www.agriterra.org)

<b>L.O.1</b>	<b>Describe on how to apply fundamental knowledge of computing and mathematics in an operating system</b>
L.O.1.1	Define the functionality and structures that a modern operating system must deliver to meet a particular need.
L.O.1.2	Explain virtual memory and its realisation in hardware and software.
<b>L.O.2</b>	<b>Able to report the tradeoffs between the performance and the resource and technology constraints in a design of an operating system.</b>
L.O.2.1	Compare and contrast common algorithms used for both preemptive and non-preemptive scheduling of tasks in operating systems
L.O.2.2	Compare and contrast different approaches to file organisation, recognizing the strengths and weaknesses of each.
<b>L.O.3</b>	<b>Describe main operating system concepts and their aspects that are useful to realise concurrent systems and describe the benefits of each.</b>
L.O.3.1	Compare and contrast different methods for process synchronisation.

# ANNEXES

## 1. Terms of Reference

- <https://www.kernel.org/doc/Documentation/vm/pagemap.txt>
- <https://www.geeksforgeeks.org/advantages-and-disadvantages-of-various-cpu-scheduling-algorithms/>
- <https://data-flair.training/blogs/priority-scheduling-algorithm-in-operating-system/>
- <https://www.scaler.com/topics/operating-system/priority-scheduling-algorithm/>