

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



**Hệ điều hành(TN) - CO2018**

---

**Bài tập lớn**

**Simple Operating System**

---

Giảng viên hướng dẫn: Hoàng Lê Hải Thanh  
Nhóm: L06 - TD7

STT	Họ tên SV	MSSV	Tên lớp
1	Đặng Quốc Phú	2212576	L06
2	Lê Thanh Phú	2212581	L06
3	Trần Xuân Phú	2212601	L06
4	Hoàng Ngô Thiên Phúc	2212612	L06

TP. HỒ CHÍ MINH, THÁNG 12/2024



## Mục lục

<b>1</b>	<b>Giới thiệu</b>	<b>3</b>
<b>2</b>	<b>Trả lời câu hỏi</b>	<b>3</b>
2.1	Scheduler . . . . .	3
2.2	Memory Management . . . . .	3
2.2.1	Virtual Memory Mapping in Each Process . . . . .	3
2.2.2	The System Physical Memory . . . . .	4
2.2.3	Paging-Based Address Translation Scheme . . . . .	5
2.2.4	Multiple Memory Segments . . . . .	6
2.3	Put It All Together . . . . .	6
<b>3</b>	<b>Thiết kế scheduler</b>	<b>7</b>
3.1	Thiết kế . . . . .	7
3.2	Test case . . . . .	12
<b>4</b>	<b>Thiết kế Paging-based memory management</b>	<b>16</b>
4.1	Thiết kế . . . . .	16
4.1.1	Thiết kế alloc function . . . . .	16
4.1.2	Thiết kế free function . . . . .	18
4.2	Testcase . . . . .	18
4.2.1	Chứng minh chức năng ghi đè lên cùng vùng nhớ . . . . .	18
4.2.2	Chứng minh chức năng sử dụng paging, mỗi page 256 byte . . . . .	19
4.2.3	Chứng minh chức năng alloc một vùng nhớ (region) đã được xoá . . . . .	21
4.2.4	Chứng minh chức năng malloc một vùng nhớ . . . . .	22

# 1 Giới thiệu

## 2 Trả lời câu hỏi

### 2.1 Scheduler

**Câu hỏi:** Lợi ích của việc sử dụng thuật toán lập lịch được mô tả trong bài tập này so với các thuật toán lập lịch khác mà bạn đã học là gì?

**Trả lời:**

- Các tiến trình được phân vào các hàng đợi theo mức độ ưu tiên, cho phép các tiến trình có mức ưu tiên cao được xử lý trước các tiến trình ưu tiên thấp. Điều này đảm bảo khả năng đáp ứng cho các tác vụ quan trọng về thời gian.
- MLQ tách biệt các loại tiến trình (ví dụ: tiến trình hệ thống và tiến trình người dùng), giúp quản lý tài nguyên hiệu quả hơn.
- Thời gian xử lý được phân bổ linh hoạt cho các tiến trình trong mỗi hàng đợi, tối ưu hóa việc sử dụng CPU.
- Thiết kế của thuật toán cho phép điều chỉnh các tham số (như số lượng mức ưu tiên và thời gian lát cắt) để thích nghi với các yêu cầu khác nhau của hệ thống.
- MLQ có thể xử lý nhiều bộ xử lý và nhiều hàng đợi, phù hợp với các hệ thống có mức độ đồng thời cao.

### 2.2 Memory Management

#### 2.2.1 Virtual Memory Mapping in Each Process

**Câu hỏi:** Trong hệ điều hành đơn giản này, chúng ta triển khai thiết kế gồm nhiều đoạn bộ nhớ hoặc khu vực bộ nhớ trong khai báo mã nguồn. Lợi ích của thiết kế đề xuất với nhiều đoạn bộ nhớ là gì?

**Trả lời:**

- Các đoạn bộ nhớ (data segment và heap segment) được phân chia tách biệt, giúp dễ dàng quản lý và truy xuất bộ nhớ.
- Chiến lược "heap go down" cho phép mở rộng heap theo hướng ngược với data segment, tận dụng tối đa không gian bộ nhớ khả dụng.

- Các phân đoạn cụ thể như data segment dùng cho cấp phát tĩnh và heap segment dùng cho cấp phát động, giúp tối ưu hóa việc quản lý bộ nhớ.
- Bằng cách cách ly các khu vực sử dụng khác nhau của chương trình (dữ liệu tĩnh và dữ liệu động), thiết kế này hạn chế các lỗi do tràn bộ nhớ hoặc ghi đè không mong muốn.
- Việc tổ chức bộ nhớ thành các đoạn riêng biệt giúp dễ dàng mở rộng hoặc sửa đổi một đoạn mà không ảnh hưởng đến các đoạn khác.
- Việc chia đoạn cho phép các chiến lược quản lý khác nhau, ví dụ: data segment có thể cố định, trong khi heap segment có thể được mở rộng linh hoạt.

### 2.2.2 The System Physical Memory

**Câu hỏi:** 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?

**Trả lời: Ưu điểm:**

- Khi sử dụng nhiều cấp, mỗi bảng trang ở cấp độ cao chỉ chứa con trỏ đến các bảng cấp thấp hơn, giảm đáng kể kích thước bộ nhớ cần thiết để lưu trữ toàn bộ bảng trang cho không gian địa chỉ lớn.
- Cấu trúc phân cấp giúp dễ dàng quản lý các không gian địa chỉ ảo lớn hơn mà không phải lưu trữ toàn bộ bảng trang trong bộ nhớ chính.
- Cấu trúc nhiều cấp chia nhỏ không gian địa chỉ thành các đơn vị nhỏ hơn, cho phép sử dụng bộ nhớ vật lý hiệu quả hơn.

**Nhược điểm:**

- Địa chỉ cần trải qua nhiều cấp phân giải (truy cập nhiều bảng) để tìm ra khung vật lý tương ứng, dẫn đến thời gian truy cập lâu hơn so với hệ thống 1 hoặc 2 cấp.
- Cần thêm cơ chế để duy trì tính toàn vẹn dữ liệu giữa các bảng phân trang ở nhiều cấp, làm tăng độ phức tạp trong cài đặt và vận hành.
- Dù giảm kích thước bảng chính, cấu trúc phân cấp yêu cầu bộ nhớ để lưu trữ các bảng ở mỗi cấp, gây tăng tổng chi phí bộ nhớ nếu không tối ưu.

### 2.2.3 Paging-Based Address Translation Scheme

**Câu hỏi: Lợi ích và bất lợi của phân đoạn kết hợp phân trang là gì?**

**Trả lời: Ưu điểm:**

- Kết hợp ưu điểm của cả phân đoạn và phân trang:
  - Phân đoạn: Cho phép chia nhỏ chương trình thành các đoạn logic (code, data, stack, heap), giúp quản lý dễ dàng hơn và giảm xung đột giữa các loại dữ liệu.
  - Phân trang: Chia nhỏ bộ nhớ vật lý thành các khung cố định (frames), giảm vấn đề phân mảnh ngoài (external fragmentation).
- Hỗ trợ không gian địa chỉ lớn hơn: Kết hợp phân đoạn với phân trang giúp tận dụng tốt hơn bộ nhớ vật lý bằng cách ánh xạ không gian địa chỉ ảo lớn vào bộ nhớ vật lý nhỏ hơn.
- Giảm phân mảnh ngoài: Nhờ phân trang trong mỗi đoạn, không cần cấp phát liên tục bộ nhớ vật lý cho toàn bộ đoạn.
- Tăng tính linh hoạt trong quản lý bộ nhớ: Cho phép mỗi đoạn sử dụng chính sách cấp phát khác nhau, phù hợp với nhu cầu cụ thể của từng đoạn.
- Dễ dàng mở rộng hoặc thay đổi: Các đoạn có thể được mở rộng hoặc thay đổi mà không ảnh hưởng đến các đoạn khác.

**Nhược điểm:**

- Độ phức tạp cao hơn: Hệ thống cần duy trì bảng đoạn (Segment Table) và bảng trang (Page Table) cho từng đoạn, làm tăng độ phức tạp trong việc triển khai và quản lý.
- Thời gian truy cập bộ nhớ tăng:
  - Tra cứu bảng đoạn để xác định địa chỉ cơ sở (base address).
  - Tra cứu bảng trang để xác định khung bộ nhớ vật lý (physical frame).
- Yêu cầu bộ nhớ cao hơn cho quản lý: Hệ thống cần thêm bộ nhớ để lưu cả bảng đoạn và bảng trang.
- Không tối ưu với các đoạn nhỏ: Đối với các đoạn rất nhỏ, việc sử dụng phân trang có thể không cần thiết, dẫn đến lãng phí tài nguyên.

### 2.2.4 Multiple Memory Segments

**Câu hỏi:** Lý do gì khiến dữ liệu được tách thành hai phân đoạn khác nhau: data và heap?

**Trả lời:**

- **Quản lý hiệu quả:** Phân đoạn dữ liệu tĩnh (data segment) để lưu trữ các giá trị cố định, trong khi heap segment dùng cho việc cấp phát và thu hồi bộ nhớ động, đảm bảo mỗi loại bộ nhớ được quản lý phù hợp với đặc điểm riêng.
- **Tránh xung đột:** Ngăn ngừa lỗi ghi đè bộ nhớ khi hai loại dữ liệu (tĩnh và động) yêu cầu mở rộng không gian.
- **Tối ưu không gian bộ nhớ:** Sử dụng các chiến lược mở rộng khác nhau: data segment mở rộng từ dưới lên (go up), còn heap segment mở rộng từ trên xuống (go down), tận dụng hiệu quả bộ nhớ có sẵn.
- **Tăng tính linh hoạt:** Dễ dàng áp dụng các cơ chế quản lý và tối ưu hóa riêng biệt cho từng loại dữ liệu, phù hợp với nhu cầu của chương trình.

## 2.3 Put It All Together

**Câu hỏi:** Điều gì sẽ xảy ra nếu synchronization không được xử lý trong hệ điều hành đơn giản của bạn? Minh họa vấn đề của hệ điều hành đơn giản (kết quả bài tập) bằng ví dụ nếu có.

**Trả lời:** Nếu synchronization (đồng bộ hóa) không được xử lý trong hệ điều hành đơn giản (simple OS), sẽ dẫn đến các vấn đề sau:

- **Xung đột tài nguyên (Resource Conflict):** Nhiều tiến trình có thể truy cập và thay đổi tài nguyên chung (như hàng đợi hoặc bộ nhớ) đồng thời, dẫn đến dữ liệu bị ghi đè hoặc không nhất quán.
- **Điều kiện tranh chấp (Race Condition):** Thứ tự thực thi không được đảm bảo, khiến các tiến trình cạnh tranh thay đổi tài nguyên, dẫn đến kết quả không xác định hoặc sai lệch.
- **Deadlock:** Hai hoặc nhiều tiến trình có thể chờ nhau mãi mãi do không có cơ chế đồng bộ để xử lý việc truy cập tài nguyên tuần tự.

- **Suy giảm hiệu năng:** Các tiến trình có thể bị treo hoặc thực thi sai, gây gián đoạn hoặc làm chậm hệ thống.

**Ví dụ minh họa:**

- **Với hàng đợi tiến trình (Ready Queue):**
  - Không có đồng bộ hóa: Hai tiến trình A và B truy cập đồng thời hàm `enqueue()` để thêm PCB vào hàng đợi.
  - Kết quả:
    - \* Tiến trình A đang thêm vào hàng đợi nhưng bị gián đoạn bởi tiến trình B.
    - \* Cả hai tiến trình ghi đè lên trạng thái hàng đợi, dẫn đến mất PCB hoặc dữ liệu không chính xác.
  - Kết quả cuối: CPU có thể chọn một PCB sai, gây lỗi hoặc hành vi bất thường.

## 3 Thiết kế scheduler

### 3.1 Thiết kế

Hệ thống vận hành dựa trên cơ chế lập lịch đa cấp hàng đợi (Multilevel Queue Scheduling), phân chia các tiến trình vào 140 hàng đợi ưu tiên khác nhau, từ cấp độ 0 (ưu tiên cao nhất) đến cấp độ 139 (ưu tiên thấp nhất). Các tiến trình trong hàng đợi có cấp độ thấp hơn sẽ được xử lý sau.

```
struct queue_t {  
    struct pcb_t * proc[MAX_QUEUE_SIZE];  
    int size;  
};
```

- `proc[MAX_QUEUE_SIZE]`: Khai báo một mảng các con trỏ `pcb_t`. Mảng này có kích thước cố định là `MAX_QUEUE_SIZE`, tức là hàng đợi có thể chứa tối đa `MAX_QUEUE_SIZE` tiến trình.
- `size`: Biến nguyên để lưu trữ số lượng tiến trình hiện có trong hàng đợi.

Ở file `queue.c` ta sẽ viết các hàm `enqueue()` và `dequeue()`: các phương thức này tuân thủ đúng quy tắc FIFO (vào trước lấy ra trước).

```
int empty(struct queue_t * q) {
    if (q == NULL) return 1;
    return (q->size == 0);
}

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

struct pcb_t * dequeue(struct queue_t * q) {
    /* TODO: return a pcb whose priority is the highest
     * in the queue [q] and remember to remove it from q
     */
    if (empty(q)) {
        return NULL; // Return NULL if the queue is empty
    }

    struct pcb_t *ret = q->proc[0];
    for (int i = 0; i < (q->size - 1); i++)
        q->proc[i] = q->proc[i + 1];
    q->proc[q->size - 1] = NULL;
    q->size--;
    return ret;
}
```

1. enqueue:

- Thêm tiến trình mới vào cuối hàng đợi bằng cách gán nó vào chỉ số `q->size` và sau đó tăng `q->size` lên 1.

2. dequeue:

3. Kiểm tra xem hàng đợi có rỗng không. Nếu rỗng, trả về NULL.



4. Lấy tiến trình ở đầu hàng đợi (chỉ số 0).
5. Dịch chuyển tất cả các phần tử trong hàng đợi một vị trí sang trái để xóa phần tử đầu tiên.
6. Giảm  $q \rightarrow \text{size}$  xuống 1 để phản ánh việc xóa.
7. Trả về tiến trình đã được lấy ra.

Ở file sched.c ta sẽ cập nhật các phần sau:

- Khai báo mảng static int slot[MAX\_PRIO] dùng để lưu trữ số slot tương ứng với mỗi mức độ ưu tiên trong thuật toán lập lịch Multi-Level Queue (MLQ).

```
14 static int slot[MAX_PRIO];
```

Hình 3.1: Khai báo mảng tĩnh slot

- Hàm init\_scheduler() sao cho mỗi queue thứ i ban đầu sẽ được cấp một lượng slot = MAX\_PRIO - i.

```
27 void init_scheduler(void) {
28     #ifdef MLQ_SCHED
29         int i ;
30
31         for (i = 0; i < MAX_PRIO; i ++){
32             mlq_ready_queue[i].size = 0;
33             slot[i] = MAX_PRIO - i;
34         }
35     #endif
36     ready_queue.size = 0;
37     run_queue.size = 0;
38     pthread_mutex_init(&queue_lock, NULL);
39 }
40
```

Hình 3.2: Cập nhật hàm init\_scheduler()

- Thuật toán MLQ của ta sẽ được thực hiện ở hàm `get_mlq_proc()` như sau:

```

48  struct pcb_t * get_mlq_proc(void) {
49      struct pcb_t * proc = NULL;
50      /*TODO: get a process from PRIORITY [ready_queue].
51       * Remember to use lock to protect the queue.
52       */
53      pthread_mutex_lock(&queue_lock);
54      int prio;
55      for(prio = 0; prio < MAX_PRIO; prio++){
56          if (empty(&mlq_ready_queue[prio]) ) {
57              slot[prio] = MAX_PRIO - prio;
58              continue;
59          }
60
61          proc = dequeue(&mlq_ready_queue[prio]);
62          slot[prio]--;
63          if(slot[prio] <= 0){
64              slot[prio] = MAX_PRIO - prio;
65          }
66          break;
67      }
68      pthread_mutex_unlock(&queue_lock);
69      return proc;
70  }

```

Hình 3.3: Hàm `get_mlq_proc()`

### 1. Khởi tạo biến `proc`:

```
struct pcb_t * proc = NULL;
```

Biến `proc` thuộc kiểu con trỏ `struct pcb_t *` và được khởi tạo bằng `NULL`. Biến này sẽ lưu trữ PCB (Process Control Block) của tiến trình được chọn từ hàng đợi.

### 2. Bảo vệ hàng đợi bằng khóa (mutex):

```
pthread_mutex_lock(&queue_lock);
```

Hàm `pthread_mutex_lock` được sử dụng để khóa mutex `queue_lock`. Điều này đảm bảo rằng chỉ có một luồng có thể truy cập và sửa đổi hàng đợi tại

một thời điểm, tránh xung đột dữ liệu khi nhiều luồng cùng thao tác trên hàng đợi.

### 3. Duyệt qua các mức ưu tiên:

```
int prio;  
for(prio = 0; prio < MAX_PRIO; prio++){  
    // ...  
}
```

Vòng lặp for duyệt qua tất cả các mức ưu tiên từ 0 đến  $\text{MAX\_PRIO} - 1$ .

### 4. Kiểm tra hàng đợi rỗng:

```
if (empty(&mlq_ready_queue[prio]) ) {  
    slot[prio] = MAX_PRIO - prio;  
    continue;  
}
```

- Hàm `empty` kiểm tra xem hàng đợi `mlq_ready_queue[prio]` tương ứng với mức ưu tiên `prio` có rỗng hay không.
- Nếu hàng đợi rỗng, số slot `slot[prio]` được đặt lại bằng  $\text{MAX\_PRIO} - \text{prio}$  và vòng lặp tiếp tục với mức ưu tiên tiếp theo.

### 5. Lấy tiến trình từ hàng đợi:

```
proc = dequeue(&mlq_ready_queue[prio]);
```

Nếu hàng đợi không rỗng, hàm `dequeue` được gọi để lấy PCB của tiến trình đầu tiên trong hàng đợi `mlq_ready_queue[prio]` và gán nó cho biến `proc`.

### 6. Giảm số slot và thiết lập lại nếu cần:

```
slot[prio]--;  
if(slot[prio] <= 0){  
    slot[prio] = MAX_PRIO - prio;  
}
```

- Sau khi lấy tiến trình, số slot `slot[prio]` tương ứng với mức ưu tiên đó sẽ bị giảm đi 1.
- Nếu `slot[prio]` nhỏ hơn hoặc bằng 0, nghĩa là đã sử dụng hết slot cho mức ưu tiên này, số slot sẽ được đặt lại bằng  $\text{MAX\_PRIO} - \text{prio}$ .

### 7. Thoát vòng lặp:

```
break;
```

Sau khi lấy được tiến trình, vòng lặp **for** sẽ bị thoát bằng lệnh **break**.

### 8. Mở khóa mutex:

```
pthread_mutex_unlock(&queue_lock);
```

Hàm `pthread_mutex_unlock` được gọi để mở khóa mutex `queue_lock`, cho phép các luồng khác truy cập hàng đợi.

### 9. Trả về PCB của tiến trình:

```
return proc;
```

Hàm trả về con trỏ `proc` trỏ đến PCB của tiến trình được chọn.

## 3.2 Test case

File đầu vào chứa các thông tin sau:

- Time slice: 2
- Số lượng CPU: 1
- Số lượng tiến trình: 6

File tiến trình:

- File s4: có 30 lệnh (chỉ có các lệnh calc) và live priority = 4, arrive time = 1.
- File s3: có 17 lệnh (chỉ có các lệnh calc) và live priority = 3, arrive time = 2.
- File s2: có 13 instructions (chỉ có các lệnh calc) và prio = 3, arrive time = 6.
- File p1s: có 10 lệnh (chỉ có các lệnh calc) và live priority = 2, arrive time = 9.
- File s0: có 15 lệnh (chỉ có các lệnh calc) và live priority = 1, arrive time = 11.
- File s1: có 7 lệnh (chỉ có các lệnh calc) và live priority = 0, arrive time = 16.



Arrival Time	Process Name	Live Priority	Number of Instructions
1	s4(p1)	4	30
2	s3(p2)	3	17
6	s2(p3)	3	13
9	p1s(p4)	2	10
11	s0(p5)	1	15
16	s1(p6)	0	7

Kết quả:

```
Time slot    0
ld_routine
Time slot    1
    Loaded a process at input/proc/s4, PID: 1 PRI0: 4
Time slot    2
    CPU 0: Dispatched process 1
    Loaded a process at input/proc/s3, PID: 2 PRI0: 3
Time slot    3
Time slot    4
    CPU 0: Put process 1 to run queue
    CPU 0: Dispatched process 2
Time slot    5
Time slot    6
    CPU 0: Put process 2 to run queue
    CPU 0: Dispatched process 2
    Loaded a process at input/proc/s2, PID: 3 PRI0: 3
Time slot    7
Time slot    8
    CPU 0: Put process 2 to run queue
    CPU 0: Dispatched process 3
Time slot    9
    Loaded a process at input/proc/p1s, PID: 4 PRI0: 2
Time slot   10
    CPU 0: Put process 3 to run queue
    CPU 0: Dispatched process 4
Time slot   11
```



```
        Loaded a process at input/proc/s0, PID: 5 PRI0: 1
Time slot 12
        CPU 0: Put process 4 to run queue
        CPU 0: Dispatched process 5
Time slot 13
Time slot 14
        CPU 0: Put process 5 to run queue
        CPU 0: Dispatched process 5
Time slot 15
Time slot 16
        Loaded a process at input/proc/s1, PID: 6 PRI0: 0
        CPU 0: Put process 5 to run queue
        CPU 0: Dispatched process 6
Time slot 17
Time slot 18
        CPU 0: Put process 6 to run queue
        CPU 0: Dispatched process 6
Time slot 19
Time slot 20
        CPU 0: Put process 6 to run queue
        CPU 0: Dispatched process 6
Time slot 21
Time slot 22
        CPU 0: Put process 6 to run queue
        CPU 0: Dispatched process 6
Time slot 23
        CPU 0: Processed 6 has finished
        CPU 0: Dispatched process 5
Time slot 24
Time slot 25
        CPU 0: Put process 5 to run queue
        CPU 0: Dispatched process 5
...
Time slot 94
        CPU 0: Processed 1 has finished
```



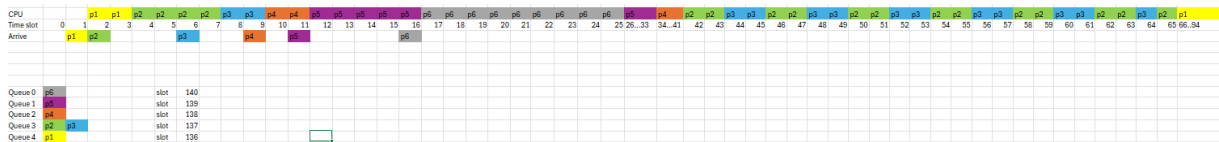
CPU 0 stopped

Giải thích kết quả có được:

Time Slot	Hành động	PID	Priority	Giải thích
0	ld_routine	-	-	Khởi động hệ thống.
1	Loaded a process at input / proc / s4	1	4	Nạp tiến trình s4 với PID 1 và priority 4.
2	CPU 0: Dispatched process 1	1	4	CPU 0 bắt đầu xử lý tiến trình s4.
2	Loaded a process at input / proc / s3	2	3	Tiến trình s3 được nạp khi s4 đang được xử lý.
4	CPU 0: Put process 1 to run queue	1	4	Sau 2 time slice, s4 bị đưa vào hàng đợi run queue.
4	CPU 0: Dispatched process 2	2	3	CPU 0 chuyển sang xử lý tiến trình s3.
6	CPU 0: Put process 2 to run queue	2	3	Sau 2 time slice, s3 bị đưa vào hàng đợi run queue.
6	CPU 0: Dispatched process 2	2	3	CPU 0 tiếp tục xử lý s3.
6	Loaded a process at input / proc / s2	3	3	Tiến trình s2 được nạp.
8	CPU 0: Put process 2 to run queue	2	3	Sau 2 time slice, s3 lại bị đưa vào hàng đợi run queue.
8	CPU 0: Dispatched process 3	3	3	CPU 0 chuyển sang xử lý s2.
9	Loaded a process at input / proc / p1s	4	2	Tiến trình p1s được nạp.
10	CPU 0: Put process 3 to run queue	3	3	Sau 2 time slice, s2 bị đưa vào hàng đợi run queue.
10	CPU 0: Dispatched process 4	4	2	CPU 0 chuyển sang xử lý p1s.

11	Loaded a process at input / proc / s0	5	1	Tiến trình s0 được nạp.
12	CPU 0: Put process 4 to run queue	4	2	Sau 2 time slice, p1s bị đưa vào hàng đợi run queue.
12	CPU 0: Dispatched process 5	5	1	CPU 0 chuyển sang xử lý s0.
14	CPU 0: Put process 5 to run queue	5	1	Sau 2 time slice, s0 bị đưa vào hàng đợi run queue.
14	CPU 0: Dispatched process 5	5	1	CPU 0 tiếp tục xử lý s0.
...	...	...	...	...
94	CPU 0: Processed 1 has finished	1	4	Tiến trình s4 kết thúc sau khi được xử lý xong.

Phác họa biểu diễn như sau:



Hình 3.4: Phác họa MLQ

Xem chi tiết tại link: [MY EXCEL](#)

## 4 Thiết kế Paging-based memory management

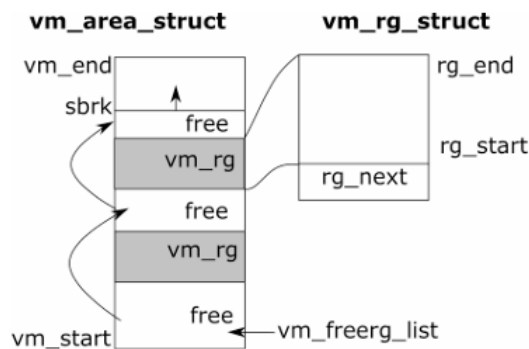
## 4.1 Thiết kế

### 4.1.1 Thiết kế alloc function

Thực tế một process có thể có nhiều segmentation với nhiều kích thước khác nhau, và mỗi process sẽ có một vùng không gian bộ nhớ ảo riêng của mình. Ở Bài tập lớn hiện tại chỉ giới hạn mỗi process chỉ cần 1 segment hay vm-area-struct với vmaid = 0 (id của vm-area-struct, duy nhất với mỗi vm-area-struct). Trong vm-area-struct sẽ là một vùng nhớ có kích thước theo bội số của kích thước một page, hay nói cách khác vm-area-struct



là một segmentation với nhiều trang bên trong. Bên trong vm-area-struct sẽ có cấu trúc như đã đề cập trong mô tả BTL (gồm nhiều vm-rg-struct và các region free được liên kết với nhau bằng linked list). Ta có hình vẽ minh họa sau:



Hình 4.1: Sơ đồ khối tổng quát

Ta sẽ thực hiện cấp phát vùng nhớ theo các bước sau:

- Đầu tiên, sẽ kiểm tra lượng vùng nhớ cần cấp + mốc sbrk có vượt ngưỡng vm-end (điểm cuối của vm-area-struct, đối ngược là vm-start là mốc đầu tiên của vm-area-struct). Nếu vượt ngưỡng, ta sẽ phải nâng vm-end theo size là bội số của kích thước 1 page, nếu không thì ta chỉ cần đơn thuần nâng sbrk theo size gốc (không theo bội số của kích thước page) và kết thúc quá trình cấp phát.

- Với trường hợp cần nâng vm-end, ta sẽ tính toán số lượng page cần cấp phát. Ta tính được số page cần cấp theo công thức sau, với size là số lượng byte vùng nhớ cần cấp:  

$$\text{SỐ PAGE} = (\text{PAGING-PAGE-ALIGNSZ}(\text{sbrk} + \text{size}) - \text{PAGING-PAGE-ALIGNSZ}(\text{sbrk})) / \text{PAGING-PAGESZ}$$

với macro PAGING-PAGE-ALIGNSZ sẽ tính toán số page tương ứng với kích thước truyền vào và macro PAGING-PAGESZ là kích thước một page. Công việc tính toán này sẽ được thực hiện trong hàm inc-vma-limit().

- Kế tiếp ta tăng vm-end theo đúng lượng kích thước ứng với số page đã tính ở trên và tiến hành ánh xạ các page vào frame trên RAM. ta thực hiện nó trong hàm vm-map-ram(). Trong vm-map-ram() ta sẽ thực hiện 2 việc, việc đầu tiên ta gọi hàm alloc-pages-range() để trả về danh sách những frame trống trong RAM hoặc SWAP trong trường hợp RAM thiếu frame trống. Ở đây ta loại bỏ trường hợp cả RAM và SWAP đều không đủ frame. Việc thứ hai sau khi có được danh sách frame trống, ta gọi hàm vmap-page-range() để tiến hành ánh xạ những page cần ánh xạ với các frame trống, sau đó cập nhật page table của process và thêm page number vào danh sách những page vừa tham chiếu trong mm->fifo-pgn bằng hàm enlist-pgn-node().

Sau khi hoàn tất các bước ánh xạ, chương trình quay về lại hàm `alloc()` để cập nhật `sbrk` theo đúng size truyền vào và cập nhật các thông số `rg-start` và `rg-end` của region vừa `alloc`. Với thiết kế này ta có thể tránh được tình trạng phân mảnh nội của phương pháp phân trang, tuy nhiên vẫn gây ra hiện tượng phân mảnh ngoại của phương pháp phân đoạn.

#### 4.1.2 Thiết kế free function

Với hàm `free`, ta chỉ cần đơn thuần xoá đi cập nhật lại thông tin `rg-start` và `rg-end` của `symtblrg` tại region tương ứng.

### 4.2 Testcase

- Time slice: 6
- Số lượng CPU: 1
- Số lượng tiến trình: 1
- RAM: 1048576 (1MB)
- SWAP1: 16777216 (16MB)
- SWAP2: 0 (0MB)
- SWAP3: 0 (0MB)
- SWAP4: 0 (0MB)
- VMEMSZ: 4194304 (4MB)

#### 4.2.1 Chứng minh chức năng ghi đè lên cùng vùng nhớ

Phân tích tiến trình:

```
1 11 alloc 256 2
alloc 256 1
alloc 256 3
alloc 512 0
write 20 2 200
read 2 200 20
write 30 2 200
read 2 200 20
```

free 1  
free 2  
free 3  
free 0

Page table tại thời điểm sau khi thực hiện xong các lệnh alloc có dạng:

```
Time slot 3
print_pgtbl: 0 - 768
00000000: 80000000
00000004: 80000001
00000008: 80000002
Alloc: 0x75ab271ffe34
In address extended: 768
In address extended: 1280
```

Hình 4.2: Page table

p thực hiện 2 lệnh write 20 và 30 cùng vào vị trí có offset 200 của region 2, tuy nhiên chỉ có giá trị 30 được thực hiện sau là ghi. Kết quả được kiểm chứng tại time slot 7, p thực hiện lệnh read trả về kết quả là 30.

```
Time slot 7
read region=2 offset=200 value=30
print_pgtbl: 0 - 1280
00000000: c0000000
00000004: 80000001
00000008: 80000002
00000012: 80000000
00000016: 80000000
0000001e
```

Hình 4.3: Result

#### 4.2.2 Chứng minh chức năng sử dụng paging, mỗi page 256 byte

Phân tích tiến trình:

1 8  
alloc 256 2

alloc 256 1

alloc 256 3

alloc 256 0

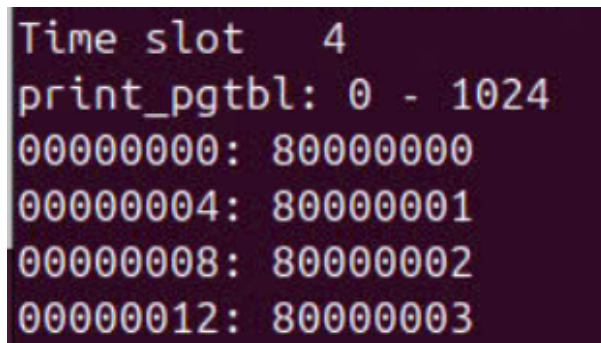
free 0

free 1

free 2

free 3

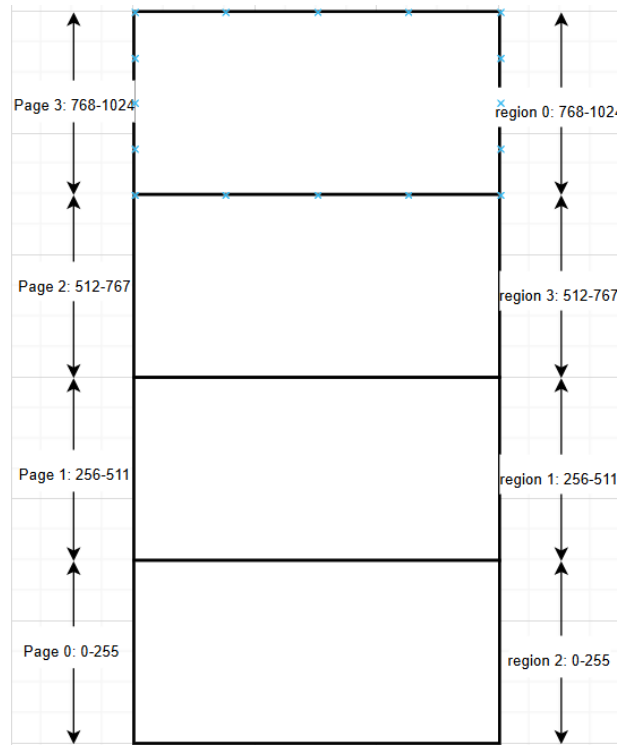
Page table tại thời điểm sau khi thực hiện xong các lệnh alloc có dạng:



```
Time slot 4
print_pgtbl: 0 - 1024
00000000: 80000000
00000004: 80000001
00000008: 80000002
00000012: 80000003
```

Hình 4.4: Page table

Trong đó, vùng không gian địa chỉ ảo của tiến trình là 0 – 1024 (0x00000000 – 0x00000400) với không gian của từng region là: region 1 là 256 – 511, region 2 là 0 – 255, region 3 512 – 767, region 0 768 – 1024. Hình ảnh của từng vùng được mô tả như sau:



#### 4.2.3 Chứng minh chức năng alloc một vùng nhớ (region) đã được xoá

Phân tích tiến trình:

```
1 9
alloc 100 2
alloc 100 3
write 20 3 100
free 3
alloc 200 1
write 30 1 100
read 1 100
free 1
free 2
```

Page table tại thời điểm sau khi thực hiện xong các lệnh alloc có dạng:

```
Time slot 2
print_pgtbl: 0 - 256
00000000: 80000000
Alloc: 0x7c5ca9dffe34
In address: 100
In address: 200
Time slot 3
write region=3 offset=100 value=20
print_pgtbl: 0 - 256
00000000: 80000000
00000014
Time slot 4
print_pgtbl: 0 - 256
00000000: 80000000
free
Time slot 5
print_pgtbl: 0 - 256
00000000: 80000000
Alloc: 0x7c5ca9dffe34
In address extended: 100
In address extended: 300
Time slot 6
write region=1 offset=100 value=30
print_pgtbl: 0 - 512
00000000: 80000000
00000004: 80000001
00000014
0000001e
```

Sau khi được cấp phát region 3 chiếm vùng không gian từ 100 đến 199. Sau khi hoàn tất, region 3 được xóa và trả lại vùng nhớ để cấp phát cho lần sau, region 3 không chiếm dụng vùng nhớ khi đã xóa. Điều này được thể hiện thông qua việc region 1 yêu cầu vùng nhớ có kích thước 200, hệ điều hành cấp phát từ điểm bắt đầu của region 3 trước đó (là 100).

#### 4.2.4 Chứng minh chức năng malloc một vùng nhớ

Phân tích tiến trình:

1 4

malloc 256 0

free 0

alloc 200 0

free 0

```
ld_routine
    Loaded a process at input/proc/pos, PID: 1 PRIO: 0
    CPU 0: Dispatched process 1
print_pgtbl: 0 - 0
Alloc: 0x7e49a8bffe34
In address: 768
In address: 1024
print_pgtbl: 0 - 0
free
Time slot 1
Time slot 2
print_pgtbl: 0 - 0
Alloc: 0x7e49a8bffe34
In address extended: 0
In address extended: 200
Time slot 3
print_pgtbl: 0 - 256
00000000: 80000000
free
Time slot 4
    CPU 0: Processed 1 has finished
    CPU 0 stopped
```

Khi một lệnh malloc được gọi, vùng nhớ cho nó được cấp phát từ trên xuống ("go down"). Như trong kiểm thử ở trên, khi malloc một vùng nhớ có kích thước 256, hệ điều hành cung cấp một vùng nhớ từ trên xuống (tức từ 1024 xuống 768). Điều này phân biệt với lệnh alloc như trong ví dụ (từ 0 đến 256).