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

BÁO CÁO BÀI TẬP LỚN
HỆ ĐIỀU HÀNH (CO2017)

THIẾT KẾ HỆ ĐIỀU HÀNH ĐƠN GIẢN

GVHD: Trương Tuấn Phát
Nguyễn Phương Duy

SV thực hiện: Trần Nguyễn Thái Bình – 2110051
Hoàng Đức Nguyên – 2110393
Nguyễn Phan Hoàng Phúc – 2110457
Đặng Quang Vinh – 2110667
Trương Hoàng Nguyên Vũ – 2112673

Lớp: L06

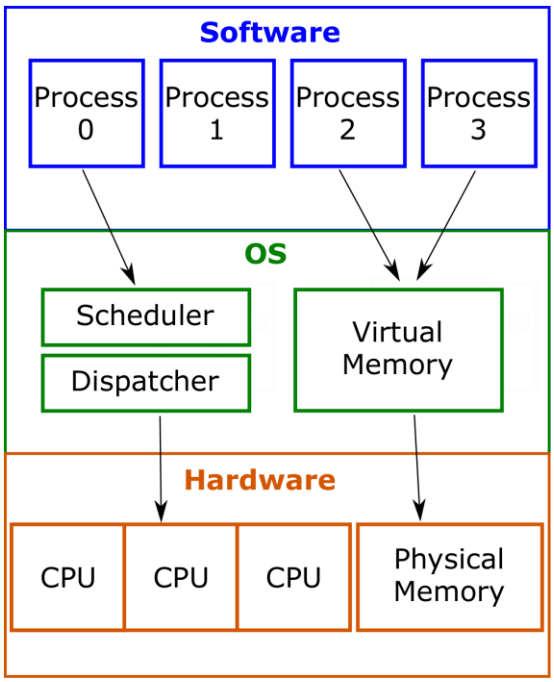
Tp. Hồ Chí Minh, Tháng 5/2023

Mục lục

1 Giới thiệu

Mục đích của bài tập lớn này là mô phỏng một hệ điều hành đơn giản, giúp sinh viên hiểu kiến thức cơ bản về định thời (Scheduling), quá trình đồng bộ (Synchronization) và quản lý bộ nhớ (Memory Management). Hình 1 thể hiển tổng quan cấu trúc của hệ điều hành mà chúng ta sẽ hiện thực. Về cơ bản, hệ điều hành sẽ quản lý 2 tài nguyên ảo: CPU(s) và RAM, sử dụng 2 thành phần:

- Scheduler (và Dispatcher): quyết định quá trình nào sẽ được thực thi trên CPU nào đó
- Virtual Memory Engine (VME): cô lập không gian bộ nhớ của mỗi quá trình khỏi những quá trình khác. Mặc dù RAM được chia sẻ bởi nhiều quá trình, mỗi quá trình không biết sự tồn tại của các quá trình khác. Điều này được thực hiện bằng cách cho phép mỗi tiến trình có không gian bộ nhớ ảo riêng và công cụ bộ nhớ ảo sẽ ánh xạ và dịch các địa chỉ logic được cung cấp bởi các quy trình sang các địa chỉ vật lý tương ứng.



Hình 1: Tổng quan về các module chính trong bài tập lớn này

Thông qua các module trên, hệ điều hành cho phép nhiều quá trình được tạo ra bởi người dùng chia sẻ và sử dụng các tài nguyên. Do đó, ở trong bài tập lớn lần này, chúng ta sẽ tiến hành hiện thực các thành phần Scheduler/Dispatcher và VME.

2 Mô tả của các process trong bài tập lớn

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. Lưu ý một vài process có chỉnh sửa để thống nhất số lệnh.

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
```

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

p2s

```
20 12
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
```

Process này thực hiện 12 lệnh calc như mô tả, và **số timeslot cần chạy để thực hiện process là 12.**

p3s

```
7 11
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
```

Process này thực hiện 11 lệnh calc như mô tả, và **số timeslot cần chạy để thực hiện process là 11.**

s0

```
12 15
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
```

Process này thực hiện 15 lệnh calc như mô tả, và **số timeslot cần chạy để thực hiện process là 15.**

s1

```
20 7
calc
calc
calc
calc
calc
calc
calc
```

Process này thực hiện 7 lệnh calc như mô tả, và **số timeslot cần chạy để thực hiện process là 7.**

s2

```
20 12
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
```

s3

```
7 11
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
```

s4

```
20 7
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
```

3 Scheduler

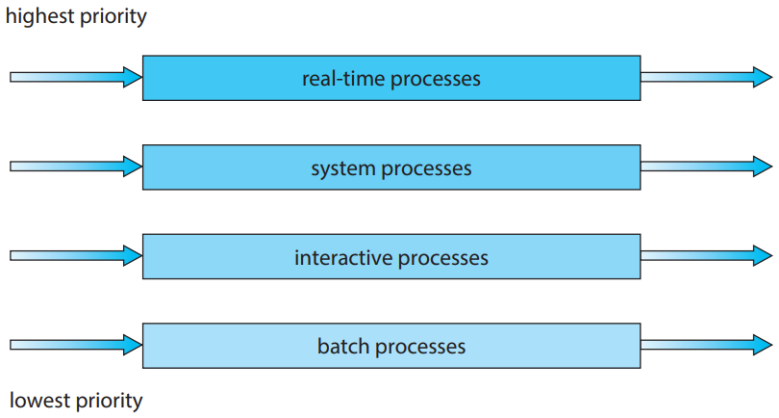
3.1 Cơ sở lý thuyết

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.



Hình 2: Ví dụ về phân nhóm các process

3.2 Trả lời câu hỏi

Câu hỏi: 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

3.3.1 Hiện thực queue.c

Hàm en_queue
Chức năng: Hàm en_queue giúp đưa process mới vào trong ready queue.
Code:

```
void enqueue(struct queue_t * q, struct pcb_t * proc) {
    /* TODO: put a new process to queue [q] */
    q->size++;
    q->proc[q->size - 1] = proc;
    return;
}
```

Hàm de_queue
Chức năng: Hàm de_queue trả về process có có priority cao nhất trong ready queue và lấy nó ra khỏi ready queue.
Code:

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

3.3.2 Hiện thực sched.c

Hàm get_mlq_proc

```
struct pcb_t * get_mlq_proc(void) {
    /*TODO: get a process from PRIORITY[ready_queue].
   * Remember to use lock to protect the queue.
   * */
    static int curr_prio = MAX_PRIO;
    static int curr_slot = 0;
    struct pcb_t * proc = NULL;
    pthread_mutex_lock(&var_lock);
    pthread_mutex_lock(&queue_lock);
    int double_check = 0;
    while (double_check < 2) {
        if (!empty(&mlq_ready_queue[MAX_PRIO - curr_prio])) {
            proc = dequeue(&mlq_ready_queue[MAX_PRIO - curr_prio]);
            curr_slot++;
            if (curr_slot >= curr_prio) { // Check curr_slot >= Thoi gian cho phep chay;
                curr_slot = 0;
                curr_prio--; // Ready to jump next queue
            }
            break;
        } else {
```

```
            // printf("%d\n", curr_prio);
            curr_slot = 0;
            curr_prio--; // Ready to jump to next queue
        }

        if (curr_prio < 1) {
            double_check += 1;
            curr_prio = MAX_PRIO;
        }
    }
    pthread_mutex_unlock(&queue_lock);
    pthread_mutex_unlock(&var_lock);
    return proc;
}
```

Hàm get_proc

```
struct pcb_t * get_proc(void) {
  struct pcb_t * proc = NULL;
  /*TODO: get a process from [ready_queue].
   * Remember to use lock to protect the queue.
   * */
    pthread_mutex_lock(&queue_lock);
    if (!empty(&ready_queue)) {
        proc = dequeue(&ready_queue);
    }
    pthread_mutex_unlock(&queue_lock);
    return proc;
}
```

3.4 Kết quả thực thi

Để hiện thực các testase scheduler, ta cần comment dòng `#define MM_PAGING` ở hai file os-cfg.h và os-mm.h. Khi test cần chú ý `max_prio` của từng file và chỉnh giá trị đó ở hai file là sched.h và os-cfg.h, các giá trị cần chỉnh bao gồm `MLQ_SCHED` và `MAX_PRIO` ở hai file

Testcase sched:

Ở testcase này ta chỉnh `MLQ_SCHED` và `MAX_PRIO` bằng 2

Input:

```
4 2 3
0 p1s 1
1 p1s 0
2 p1s 0
```

Output:

```
ld_routine
        Loaded a process at input/proc/p1s, PID: 1 PRIO: 1
        CPU 0: Dispatched process  1
==================
Time slot   0
        Loaded a process at input/proc/p1s, PID: 2 PRIO: 0
==================
Time slot   1
        CPU 1: Dispatched process  2
        Loaded a process at input/proc/p1s, PID: 3 PRIO: 0
==================
Time slot   2
==================
Time slot   3
==================
Time slot   4
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  3
==================
Time slot   5
        CPU 1: Put process  2 to run queue
        CPU 1: Dispatched process  1
==================
Time slot   6
==================
Time slot   7
==================
Time slot   8
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  2
==================
Time slot   9
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  3
==================
Time slot  10
==================
Time slot  11
==================
Time slot  12
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  1
==================
Time slot  13
        CPU 1: Put process  3 to run queue
        CPU 1: Dispatched process  2
==================
Time slot  14
        CPU 0: Processed  1 has finished
        CPU 0: Dispatched process  3
==================
Time slot  15
        CPU 1: Processed  2 has finished
        CPU 1 stopped
==================
Time slot  16
        CPU 0: Processed  3 has finished
        CPU 0 stopped
```

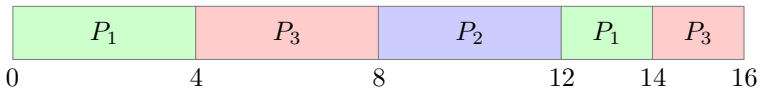
Hình 3: Output của sched

Phân tích kết quả

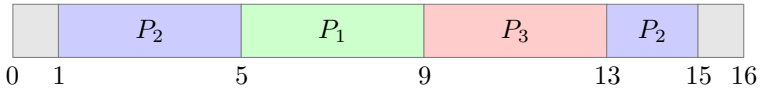
- Tại dòng đầu tiên của input, ta đọc được dãy 4 2 3, nghĩa là mỗi timeslice sẽ là 4, có 2 CPU là CPU0 và CPU1 và cuối cùng là sẽ có tổng cộng 3 process. Các process đều được định nghĩa p1s nên thời gian thực thi của mỗi process là 10s. Các process có độ ưu tiên bé nhất là 1 nên số queue ở đây là 2.
- 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).
- Tại timeslot 0, 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 1, 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 2, 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 queue 1).
- Đến timeslot 5, 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 1 được đưa vào CPU1 đang trống.
- Tiếp tục tuần tự như vậy đến timeslot 16, khi CPU0 hoàn thành process 3 thì chương trình kết thúc.

Biểu đồ Gantt

CPU 0



CPU 1



Testcase sched_0

Ở testcase này ta chỉnh `MLQ_SCHED` và `MAX_PRIO` bằng 5

Input:

```
2 1 2
0 s0 4
1 s0 0
```

Output:

```
==================
Time slot   0
ld_routine
        Loaded a process at input/proc/s0, PID: 1 PRIO: 4
        CPU 0: Dispatched process  1
==================
Time slot   1
        Loaded a process at input/proc/s0, PID: 2 PRIO: 0
==================
Time slot   2
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
==================
Time slot   3
==================
Time slot   4
        CPU 0: Put process  2 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
==================
Time slot   7
==================
Time slot   8
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
==================
Time slot   9
==================
Time slot  10
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
==================
Time slot  11
==================
Time slot  12
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  1
==================
Time slot  13
==================
Time slot  14
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
==================
Time slot  15
==================
Time slot  16
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
==================
Time slot  17
==================
Time slot  18
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
==================
Time slot  19
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  1
==================
Time slot  20
==================
Time slot  21
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
==================
Time slot  22
==================
Time slot  23
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
==================
Time slot  19
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  1
==================
Time slot  20
==================
Time slot  21
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
==================
Time slot  22
==================
Time slot  23
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
==================
Time slot  24
==================
Time slot  25
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
==================
Time slot  26
==================
Time slot  27
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
==================
Time slot  28
==================
Time slot  29
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
==================
Time slot  30
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

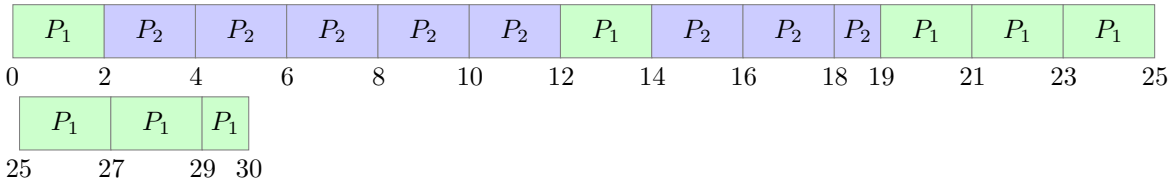
Hình 4: Output của sched_0

Phân tích kết quả

- Trong testcase này chúng ta chỉ có một CPU duy nhất nên cả 2 process sẽ chỉ có thể chạy luân phiên với nhau. Các process có độ ưu tiên thấp nhất là 4 nên số queue được sử dụng sẽ là 5.
- Đầu tiên tại timeslot 0, process 1 được load vào queue 4 sau đó được load vào CPU đang trống và thực thi. Khi process 2 xuất hiện tại timeslot 1, nó được load vào queue 0, tuy nhiên CPU đang được sử dụng bởi process 1 nên nó sẽ chờ.

- Đến timeslot 2, process 1 thực thi xong và trở về queue 4, hết 1 / 1 lần dispatch của queue 4, hệ điều hành quay ngược lên queue 0 và bắt đầu lấy process 2 vào CPU để thực thi (1/ 5 lần dispatch).
- Đến timeslot 4, hết timeslice 2s nên process 2 được lấy ra khỏi CPU và trở về queue 0, tuy nhiên chỉ mới 1 / 5 lần dispatch của queue 0 nên hệ điều hành tiếp tục lấy process 2 từ queue 0 để thực thi (2/ 5 lần dispatch).
- Cứ tiếp tục như vậy cho đến timeslot 12, lúc này queue 0 đã dispatch được 5 lần, process 2 được đưa về queue 0, hệ điều hành xuống queue 4 lấy process 1 vào CPU thực thi.
- Sau khi thực thi xong, process 1 trở về queue 4, hệ điều hành quay về queue 0 và tiếp tục dispatch nhiều lần liên tiếp ở đây.
- Đến timeslot 19, process 2 đã thực hiện xong, queue 0 trống nên hệ điều hành xuống queue 4 lấy process 1 và đưa nó thực thi đến khi xong vào timeslot 30.

Biểu đồ Gantt



Testcase sched_1

Ở testcase này ta chỉnh `MLQ_SCHED` và `MAX_PRIO` bằng 5

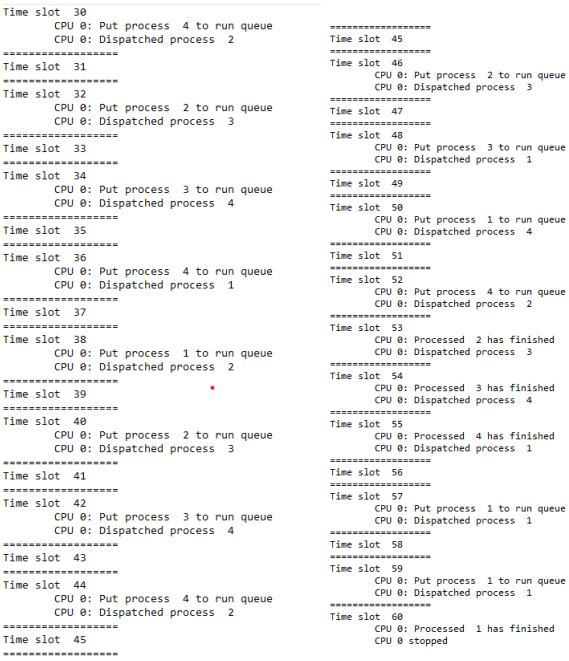
Input:

```
2 1 4
0 s0 4
1 s0 0
2 s0 0
3 s0 0
```

Output:

```
ld_routine
        Loaded a process at input/proc/s0, PID: 1 PRIO: 4
==================
Time slot   0
        CPU 0: Dispatched process  1
==================
Time slot   1
        Loaded a process at input/proc/s0, PID: 2 PRIO: 0
==================
Time slot   2
        Loaded a process at input/proc/s0, PID: 3 PRIO: 0
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
==================
Time slot   3
        Loaded a process at input/proc/s0, PID: 4 PRIO: 0
==================
Time slot   4
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  3
==================
Time slot   5
==================
Time slot   6
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
==================
Time slot   7
==================
Time slot   8
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  2
==================
Time slot   9
==================
Time slot  10
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  3
==================
Time slot  11
==================
Time slot  12
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  1
==================
Time slot  13
==================
Time slot  14
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  4
==================
Time slot  15
==================
Time slot  16
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  2
==================
Time slot  17
==================
Time slot  18
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  3
==================
Time slot  19
==================
Time slot  20
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
==================
Time slot  21
==================
Time slot  22
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  2
==================
Time slot  23
==================
Time slot  24
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  1
==================
Time slot  25
==================
Time slot  26
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  3
==================
Time slot  27
==================
Time slot  28
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
==================
Time slot  29
==================
```

Output của 30 timeslot đầu tiên sched_1



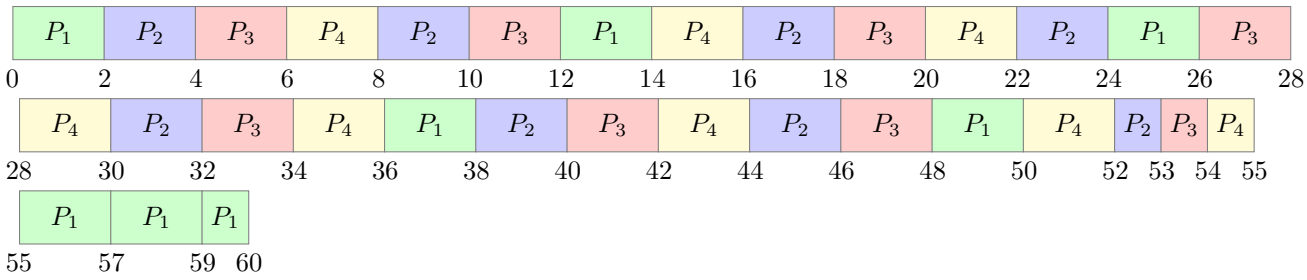
```
Time slot  30
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  2
==================
Time slot  31
==================
Time slot  32
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  3
==================
Time slot  33
==================
Time slot  34
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
==================
Time slot  35
==================
Time slot  36
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  1
==================
Time slot  37
==================
Time slot  38
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
==================
Time slot  39
==================
Time slot  40
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  3
==================
Time slot  41
==================
Time slot  42
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
==================
Time slot  43
==================
Time slot  44
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  2
==================
Time slot  45
==================
Time slot  45
==================
Time slot  46
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  3
==================
Time slot  47
==================
Time slot  48
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  1
==================
Time slot  49
==================
Time slot  50
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  4
==================
Time slot  51
==================
Time slot  52
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  2
==================
Time slot  53
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  3
==================
Time slot  54
        CPU 0: Processed  3 has finished
        CPU 0: Dispatched process  4
==================
Time slot  55
        CPU 0: Processed  4 has finished
        CPU 0: Dispatched process  1
==================
Time slot  56
==================
Time slot  57
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
==================
Time slot  58
==================
Time slot  59
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
==================
Time slot  60
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Output của 30 timeslot sau sched_1

Phân tích kết quả

- Tại dòng đầu tiên của input, ta đọc được dãy 2 1 4, nghĩa là mỗi timeslice sẽ là 2, có 1 CPU là CPU0 và cuối cùng là sẽ có 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.
- 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



4 Memory Management

4.1 Cơ sở lý thuyết

Trong Bài tập lớn này, Kỹ thuật Paging được sử dụng để quản lý bộ nhớ (Memory Management). Paging là kỹ thuật quản lý bộ nhớ (memory), trong đó máy tính lưu trữ và nhận dữ liệu từ bộ nhớ thứ cấp để sử dụng trong bộ nhớ chính.

Để hiện thực Paging trong bài tập lớn này, ta có mô tả về các thành phần sử dụng cho kỹ thuật Paging như sau:

4.1.1 Bộ nhớ ảo (Virtual Memory) cho mỗi Process

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.

Mỗi process được cấp phát một vùng nhớ **Memory Area** liên tục độc lập, trong đó có những vùng nhớ con liên tục, có thể rời rạc hoặc không, được sử dụng để lưu nội dung của process (biến, code, v.v..) được gọi là **Memory Region**. Các khối vùng nhớ độc lập liên tục này được gọi là **Page**.

Mỗi process còn có một **page address (hay logical address)** được cấp bởi CPU (trong đề bài gọi là **CPU address**) để truy cập vào một vị trí vùng nhớ nhất định, được mô tả ở hình dưới đây, cụ thể gồm:

- **Page number (p):** là chỉ số của một bảng phân trang (page table) đang lưu trữ địa chỉ cơ sở của mỗi vùng nhớ process (page) trong bộ nhớ vật lý.
- **Page offset (d)**: kết hợp với địa chỉ cơ sở để xác định địa chỉ bộ nhớ vật lý được gửi tới Khối Quản lý Bộ nhớ (Memory Management Unit).

Hình 5: CPU Address

4.1.2 Bộ nhớ chính (physical memory) của hệ thống

Tương tự như virtual memory, bộ nhớ chính được chia thành các khối nhớ nhỏ có kích thước cố định gọi là **frame** hay **page frame**. Mỗi frame trong bài tập 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:

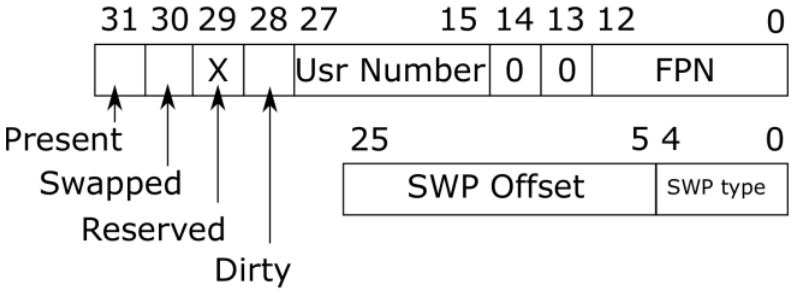
- **RAM**: có thể được truy cập trực tiếp từ CPU address bus (có thể đọc/viết với các lệnh CPU).
- **SWAP**: thiết bị nhớ thứ cấp, không thể truy cập trực tiếp từ CPU. Để thực thi các chương trình và dữ liệu lưu ở thiết bị này, chúng phải được chuyển lên bộ nhớ chính (main memory) để thực thi.

Bài tập lớn lần này được hỗ trợ một thiết bị RAM và 4 thiết bị SWAP. Trong RAM phân thành các frame gọi là MEMRAM, SWAP phân thành các frame gọi là MEMSWAP.

4.1.3 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:

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



Hình 6: Page Table Entry 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 lược 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 hiên tại đang ở trong MEMSWAP, vào các frame MEMRAM bộ nhớ chính. Cơ chế thực hiện swapping out trong mã nguồn 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: Điều gì sẽ xảy ra nếu chúng tôi chia địa chỉ thành nhiều hơn 2 cấp trong hệ thống quản lý bộ nhớ phân trang?

Trả lời: Trong hệ thống quản lý bộ nhớ phân trang, địa chỉ bộ nhớ được chia thành hai cấp: trang (page) và địa chỉ 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: Trong hệ điều hành đơn giản này, thực hiện một thiết kế của nhiều phân đoạn bộ nhớ hoặc khu vực bộ nhớ trong khai báo mã nguồn. Lợi ích của thiết kế nhiều phân đoạn được đề xuất là gì?

Trả lời: 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 Hiện thực mm-memphy.c

Hiện thực hàm MEMPHY_dump

Chức năng: dump nội dung từ mp->storage ra màn hình. Ở đây, vì kích cỡ bộ nhớ lớn, nên ta sẽ chỉ in ra những địa chỉ có nội dung không phải 0 (sẽ có trường hợp giá trị được ghi vào bộ nhớ là 0, khi đó không thể in ra được giá trị; để tránh trường hợp đó, các giá trị được ghi vào bộ nhớ sẽ khác 0 để dễ theo dõi). Ở bài tập lớn này ta chỉ quan tâm đến trạng thái của RAM nên hàm *MEMPHY_dump()* sẽ được điều chỉnh sao cho phù hợp với yêu cầu đề bài:
Code:

```
int MEMPHY_dump(struct memphy_struct * mp)
{
    /*TODO dump memphy contnt mp->storage
     *     for tracing the memory content
     */
   long int addr = 0;
   printf("------RAM CONTENT------\n");
   for(; addr < mp->maxsz; addr++) {
      if(mp->storage[addr] != 0) {
         printf("0x%08lx: %08x\n", addr, mp->storage[addr]);
      }
   }
   printf("\n");
   return 0;
}
```

4.3.2 Hiện thực mm-vm.c

Hàm __alloc
Chức năng:: cấp phát một vùng nhớ cho process. Trước tiên, nó thử tìm một vùng nhớ trống có đủ kích thước để cấp phát cho tiến trình bằng cách gọi hàm **get_free_vmrg_area()**. Nếu không tìm thấy vùng nhớ trống đủ lớn, hàm sẽ cố gắng tăng kích thước của vùng nhớ hiện có bằng cách gọi hàm **inc_vma_limit()**. Hàm sẽ cấp phát vùng nhớ tại địa chỉ đã được cấp phát trước đó; nếu xảy ra lỗi, hàm sẽ xử lý việc quản lý vùng nhớ và trả về giá trị khác không để báo lỗi. Giá trị vùng nhớ được cấp phát dư ra sẽ được đưa vào danh sách các vùng nhớ trống của process: **Code:**

```
int __alloc(struct pcb_t *caller, int vmaid, int rgid, int size, int *alloc_addr)
{
  /* Allocating invalid size */
  if(size <= 0) {
    return -1;
  }

  /* Set allocated value to 1 */
  caller->mm->allocated[rgid] = 1;

  /*Allocate at the toproof */
  struct vm_rg_struct rgnode;

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

    *alloc_addr = rgnode.rg_start;
    return 0;
  }

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

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

  old_sbrk = cur_vma->sbrk;

  /* TODO INCREASE THE LIMIT
   * inc_vma_limit(caller, vmaid, inc_sz)
   */
  inc_vma_limit(caller, vmaid, inc_sz);

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

  *alloc_addr = old_sbrk;

  if(old_sbrk + size < cur_vma->sbrk) {
    struct vm_rg_struct* free_rg = malloc(sizeof(struct vm_rg_struct));
    free_rg->rg_start = old_sbrk + size;
    free_rg->rg_end = cur_vma->sbrk;
    free_rg->rg_next = NULL;
    enlist_vm_freerg_list(caller->mm, free_rg);
  }
  return 0;
}
```

Hàm __free
Chức năng:: giải phóng không gian lưu trữ được liên kết với một khu vực nhất định (được xác định bởi **rgid**) trong vùng nhớ ảo có id là (**vmaid**) 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;

  /* Double free */
  if(caller->mm->allocated[rgid] == 0) {
    return -1;
  }

  /* TODO: Manage the collect freed region to freerg_list */
  rgnode->rg_start = caller->mm->symrgtbl[rgid].rg_start;
  rgnode->rg_end = caller->mm->symrgtbl[rgid].rg_end;

  /* Reset allocated value */
  caller->mm->allocated[rgid] = 0;

  /*enlist the obsoleted memory region */
  enlist_vm_freerg_list(caller->mm, rgnode);

  return 0;
}
```

Hàm pg_getpage

Chức năng: lấy giá trị số trang từ bộ nhớ RAM (**physical memory**) thông qua bảng phân trang. Nếu trang chưa được load vào bộ nhớ RAM, hàm sẽ thực hiện việc swap nội dung của một trang nạn nhân (được chọn theo chiến lược FIFO) hiện tại trong RAM vào bộ nhớ thứ cấp và swap ngược lại giá trị cần truy cập ở bộ nhớ thứ cấp vào trang mới bị swap trong bộ nhớ RAM. Sau khi hoàn thành, hàm cập nhật bảng trang (**page table**) để đánh dấu trang hiện tại đã được load vào bộ nhớ RAM và trả về giá trị số trang cần tìm:

Code:

```
int pg_getpage(struct mm_struct *mm, int pgn, int *fpn, struct pcb_t *caller)
{
  uint32_t pte = mm->pgd[pgn];

  if (PAGING_PAGE_SWAPPED(pte))
  { /* Page is not online, make it actively living */
    int vicpgn;
    int vicfpn;
    int tgtfpn;
    int swpfpn;
    uint32_t vicpte;

    /* Find victim page */
    find_victim_page(caller->mm, &vicpgn);

    /* Find victim frame */
    vicfpn = PAGING_GET_FPN(mm->pgd[vicpgn]);

    /* The target frame storing our variable */
    tgtfpn = PAGING_GET_SWPOFF(pte);

    /* Get free frame in MEMSWP */
    MEMPHY_get_freefp(caller->active_mswp, &swpfpn);

    /* Do swap frame from MEMRAM to MEMSWP and vice versa*/
    /* Copy victim frame to swap */
    __swap_cp_page(caller->mram, vicfpn, caller->active_mswp, swpfpn);

    /* Copy target frame from swap to mem */
    __swap_cp_page(caller->active_mswp, tgtfpn, caller->mram, vicfpn);
    MEMPHY_put_freefp(caller->active_mswp, tgtfpn);

    /* Update page table */
    vicpte = 0;
    init_pte(&vicpte, 1, 0, 0, 1, 0, swpfpn);
    mm->pgd[vicpgn] = vicpte;

```

```
    pte = 0;
    init_pte(&pte, 1, vicfpn, 0, 0, 0, 0);
    mm->pgd[pgn] = pte;

    enlist_pgn_node(&caller->mm->fifo_pgn, &caller->mm->fifo_pgn_tail, pgn);
  }

  *fpn = PAGING_GET_FPN(pte);

  return 0;
}
```

Hàm validate_overlap_vm_area
Chức năng: Kiểm tra xem có sự chồng chéo của các vùng nhớ ảo hay không.
Code:

```
int validate_overlap_vm_area(struct pcb_t *caller, int vmaid, int vmastart, int vmaend)
{
  struct vm_area_struct *vma = caller->mm->mmap;

  /* TODO validate the planned memory area is not overlapped */
  return 0;

  while (vma != NULL) {
    if (vma->vm_id != vmaid && vmaend > vma->vm_start && vmastart < vma->vm_end) {
      return -1; // overlap detected
    }
    vma = vma->vm_next;
  }

  return 0;
}
```

Hàm find_victim_page
Chức năng: Tìm page victim bằng thuật toán FIFO (First in first out) để thực hiện swap.
Code:

```
int find_victim_page(struct mm_struct *mm, int *retpgn)
{
  struct pgn_t *pg = mm->fifo_pgn;

  /* TODO: Implement the theorical mechanism to find the victim page */
  if (pg == NULL) {
    return -1;
  }

  *retpgn = pg->pgn;
  mm->fifo_pgn = pg->pg_next;

  free(pg);

  return 0;
}
```

4.3.3 Hiện thực mm.c

Hàm vmap_page_range

Chức năng: Ánh xạ (map) các khối trang vật lý (physical frames) tới một không gian địa chỉ ảo (virtual address space) cho process.
Code:

```
int vmap_page_range(struct pcb_t *caller, // process call
                                int addr, // start address which is aligned to pagesz
                               int pgnum, // 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)); // page table entry
  struct framephy_struct *fpit = malloc(sizeof(struct framephy_struct));
```

```
  int  fpn; // frame page number
  int pgit = 0;
  int pgn = PAGING_PGN(addr); // Extract page number from address

  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 + pgnum*PAGING_PAGESZ
   *      in page table caller->mm->pgd[]
   */
  for (pgit = 0; pgit < pgnum  && fpit != NULL; pgit++, fpit = fpit->fp_next){
    //The pte pointer is set to point to the page table entry for the current page number
     by adding pgn + pgit to
    // the base address of the page table stored in caller->mm->pgd
    pte = caller->mm->pgd + (pgn + pgit);
    fpn = fpit->fpn;
    pte_set_fpn(pte, fpn);
   // ret_rg->rg_end += PAGING_PAGESZ;

    /* Tracking for later page replacement activities (if needed)
    * Enqueue new usage page */
    enlist_pgn_node(&caller->mm->fifo_pgn, pgn+pgit);
  }

  return 0;
}
```

4.3.4 Kết quả thực thi

Để kiểm tra hoạt động của bộ nhớ, nhóm chúng em có thêm một đặc tả process là *m2s* như sau:

m2s

```
1 10
alloc 200 0
alloc 200 1
write 12 0 13
write 11 1 11
read 0 13
read 1 11
write 13 0 0
read 0 0
free 0
free 1
```

Với đặc tả process này, nhóm chúng em cũng tạo một testcase để kiểm tra hoạt động của bộ nhớ tên là *test_ mem*, nội dung của *test_ mem* như sau:

test_ mem

```
2 1 1
1048576 16777216 0 0 0
0 m2s 0
```

Kết quả sau khi chạy testcase *test_ mem* như sau:

Output

```
==================
Time slot   0
ld_routine
        Loaded a process at input/proc/m2s, PID: 1 PRIO: 0
        CPU 0: Dispatched process  1
allocate region=200 reg=0
------PAGE TABLE CONTENT------
print_pgtbl: 0 - 256
00000000: 80000000

------FREE REGION CONTENT------
print_list_rg:
rg[200->256]

==================
Time slot   1
allocate region=200 reg=1
------PAGE TABLE CONTENT------
print_pgtbl: 0 - 512
00000000: 80000000
00000001: 80000001

------FREE REGION CONTENT------
print_list_rg:
rg[200->256]
rg[456->512]

==================
Time slot   2
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
write region=0 offset=13 value=12
------RAM CONTENT------
0x0000000d: 0000000c

==================
Time slot   3
write region=1 offset=11 value=11
------RAM CONTENT------
0x0000000d: 0000000c
0x0000010b: 0000000b

==================
Time slot   4
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
read region=0 offset=13 value=12
==================
Time slot   5
read region=1 offset=11 value=11
==================
Time slot   6
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
write region=0 offset=0 value=13
------RAM CONTENT------
0x00000000: 0000000d
0x0000000d: 0000000c
0x0000010b: 0000000b

==================
Time slot   7
read region=0 offset=0 value=13
==================
Time slot   8
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
free reg=0
------PAGE TABLE CONTENT------
print_pgtbl: 0 - 512
00000000: 80000000
00000001: 80000001

------FREE REGION CONTENT------
print_list_rg:
rg[0->256]
rg[456->512]

==================
Time slot   9
free reg=1
------PAGE TABLE CONTENT------
print_pgtbl: 0 - 512
00000000: 80000000
00000001: 80000001

------FREE REGION CONTENT------
print_list_rg:
rg[0->512]

==================
Time slot  10
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Hình 7: Output của test_mem

Giải thích kết quả:
Ta sẽ chỉ quan tâm đến việc bộ nhớ được hoạt động thế nào trong testcase này mà không quan tâm đến phần định thời scheduler, do phần định thời ở testcase này là tương đồng với phần định thời mà phần trước đã đề cập và giải thích. Bây giờ ta sẽ giải thích kết quả in ra màn hình qua từng timeslot một của process:

- **Time slot 0:** Ở timeslot này, process này được load và được dispatch vào CPU để bắt đầu chạy. Như có thể thấy ở đặc tả của *m2s* và *test_mem* ở trên, khi này process sẽ yêu cầu cấp phát một vùng nhớ có size là 200 (lưu ý mỗi page trong simple os này có độ lớn là 256), khi này page table rỗng và không có region nào trống do đó ta cần tăng độ rộng của vùng nhớ và tạo một vùng trống được căn chỉnh bằng độ lớn một page. Sau khi đã tạo vùng nhớ như vậy, ta cần map page vừa tạo với một frame trong bộ nhớ thực (RAM), khi đó giá trị page table sẽ được thay đổi. Cuối cùng ta thêm vùng nhớ dư khi cấp phát vào danh sách các vùng nhớ rỗng của proces. Giá trị bên trái của page table là giá trị page number và giá trị bên phải là giá trị được chứa trong page table có index là page number thường được gọi là *pte*.

- **Time slot 1:** Ta tiếp tục cấp phát một vùng nhớ có độ lớn là 200. Do không có vùng nhớ trống nào đủ rộng nên ta tiếp tục tăng kích cỡ vùng nhớ ảo lên và map page vừa được thêm vào một frame của bộ nhớ RAM. Cuối cùng vùng nhớ còn dư khi cấp phát sẽ được đưa vào danh sách các vùng nhớ trống của process.

- **Time slot 2:** Ta viết giá trị 12 vào region 0 trong symbol table tại offset 13. Ta đã biết region 0 nằm trong page 0 bắt đầu ở 0 và page 0 theo page table được map tới frame 0 của bộ nhớ thực (RAM). Do đó trong bộ nhớ thực tại địa chỉ 0x0000000d sẽ có giá trị là 0000000c.

- **Time slot 3:** Ta viết giá trị 11 vào region 1 trong symbol table tại offset 11. Ta đã biết region 1 nằm trong page 1 bắt đầu ở 256 và page 1 theo page table được map tới frame 1 của bộ nhớ thực (RAM). Do đó trong bộ nhớ thực tại địa chỉ 0x0000010b sẽ có giá trị là 0000000b.

- **Time slot 4:** Ta đọc giá trị tại region 0 offset là 13; như đã được ghi ở trên, ta thấy kết quả đọc được là 12.

- **Time slot 5:** Ta đọc giá trị tại region 1 offset là 11; như đã được ghi ở trên, ta thấy kết quả đọc được là 11.
- **Time slot 6:** Ta viết giá trị 13 vào region 0 trong symbol table tại offset 0. Ta đã biết region 0 nằm trong page 0 bắt đầu ở 0 và page 0 theo page table được map tới frame 0 của bộ nhớ thực (RAM). Do đó trong bộ nhớ thực tại địa chỉ 0x00000000 sẽ có giá trị là 0000000d.
- **Time slot 7:** Ta đọc giá trị tại region 0 offset là 0; như đã được ghi ở trên, ta thấy kết quả đọc được là 13.
- **Time slot 8:** Ta giải phóng vùng nhớ region 0; khi này vùng nhớ region 0 sẽ được thêm vào danh sách các vùng nhớ trống và được combine với các vùng nhớ trống liền kề nếu có. Ta có thể thấy vùng nhớ region 0 đã được combine với một vùng nhớ trống trong danh sách các vùng nhớ trống trong kết quả ở trên để tạo nên một vùng nhớ lớn hơn.
- **Time slot 9:** Ta giải phóng vùng nhớ region 1; khi này vùng nhớ region 1 sẽ được thêm vào danh sách các vùng nhớ trống và được combine với các vùng nhớ trống liền kề nếu có. Ta có thể thấy vùng nhớ region 1 đã được combine với một vùng nhớ trống trong danh sách các vùng nhớ trống để tạo ra một vùng nhớ lớn hơn.
- **Time slot 10:** Kết thúc process, dừng CPU.

Bây giờ, nhóm của chúng em sẽ tạo ra một testcase mới tên là *test_ mem_ swapping* để kiểm tra hoạt động của bộ nhớ khi xuất hiện trường hợp bộ nhớ RAM hết dung lượng và phải thực hiện swap. Nội dung của testcase như sau:
test_mem_swapping

```
2 1 1
256 16777216 0 0 0
0 m2s 0
```

Kết quả sau khi chạy xong testcase *test_ mem_ swapping* như sau:
Output

```
ld_routine
        Loaded a process at input/proc/m2s, PID: 1 PRIO: 0
==================
Time slot   0
        CPU 0: Dispatched process  1
allocate region=200 reg=0
------PAGE TABLE CONTENT------
print_pgtbl: 0 - 256
00000000: 80000000

------FREE REGION CONTENT------
print_list_rg:
rg[200->256]

==================
Time slot   1
allocate region=200 reg=1
------PAGE TABLE CONTENT------
print_pgtbl: 0 - 512
00000000: 80000000
00000001: c0000000

------FREE REGION CONTENT------
print_list_rg:
rg[200->256]
rg[456->512]

==================
Time slot   2
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
write region=0 offset=13 value=12
------PAGE TABLE CONTENT------
print_pgtbl: 0 - 512
00000000: 80000000
00000001: c0000000

------RAM CONTENT------
0x0000000d: 0000000c

==================
Time slot   3
write region=1 offset=11 value=11
==================
Time slot   3
write region=1 offset=11 value=11
------PAGE TABLE CONTENT------
print_pgtbl: 0 - 512
00000000: c0000020
00000001: 80000000

------RAM CONTENT------
0x0000000b: 0000000b

==================
Time slot   4
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
read region=0 offset=13 value=12
------PAGE TABLE CONTENT------
print_pgtbl: 0 - 512
00000000: 80000000
00000001: c0000000

==================
Time slot   5
read region=1 offset=11 value=11
------PAGE TABLE CONTENT------
print_pgtbl: 0 - 512
00000000: c0000020
00000001: 80000000

==================
Time slot   6
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
write region=0 offset=0 value=13
------PAGE TABLE CONTENT------
print_pgtbl: 0 - 512
00000000: 80000000
00000001: c0000000

------RAM CONTENT------
0x00000000: 0000000d
0x0000000d: 0000000c

==================
Time slot   7
read region=0 offset=0 value=13
------PAGE TABLE CONTENT------
print_pgtbl: 0 - 512
00000000: 80000000
00000001: c0000000

==================
Time slot   8
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
free reg=0
------PAGE TABLE CONTENT------
print_pgtbl: 0 - 512
00000000: 80000000
00000001: c0000000

------FREE REGION CONTENT------
print_list_rg:
rg[0->256]
rg[456->512]

==================
Time slot   9
free reg=1
------PAGE TABLE CONTENT------
print_pgtbl: 0 - 512
00000000: 80000000
00000001: c0000000

------FREE REGION CONTENT------
print_list_rg:
rg[0->512]

==================
Time slot  10
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Hình 8: Output của test_mem_swapping

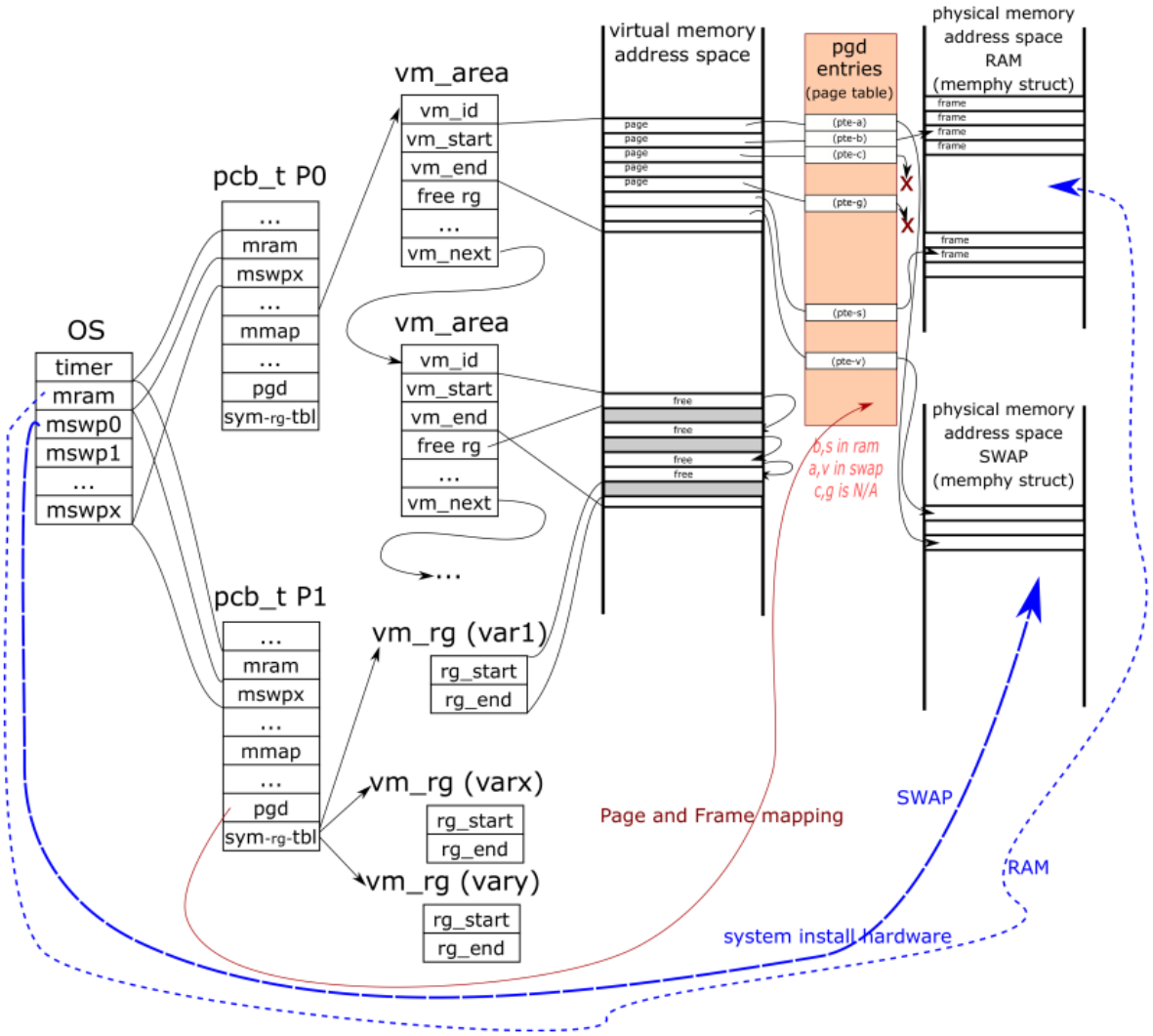
Giải thích kết quả: Tương tự như với testcase ở trên, ta chỉ quan tâm đến việc bộ nhớ hoạt động thế nào mà không quan tâm đến bộ định thời.

- **Time slot 0:** Tương tự với testcase ở trên, ta cũng cung cấp một vùng nhớ có độ lớn là 200 và map page vừa được tạo vào frame trong bộ nhớ RAM. Cuối cùng là thêm vùng nhớ phần dư vào danh sách các vùng nhớ trống của process.
- **Time slot 1:** Tương tự với testcase ở trên, ta cũng cung cấp thêm một vùng nhớ có độ lớn 200 và map page vừa được tạo vào frame trong bộ nhớ RAM. Tuy vậy trong testcase này, bộ nhớ RAM chỉ có độ lớn là 256 (kích cỡ 1 frame) nên ta tìm kiếm trong bộ nhớ thứ cấp một frame trống và map page vừa được thêm vào đó. Khi này page table cũng sẽ khác so với testcase ở trên khi giá trị pte ở page number 1 đã được kích hoạt bit swapped.
- **Time slot 2:** Ta viết giá trị 12 vào region 0 ở offset là 13. Ta đã biết region 0 nằm trong page 0 bắt đầu ở 0 và theo page table page 0 được map với frame 0 của bộ nhớ RAM do đó đang hiện diện ở bộ nhớ RAM, khi này ta viết giá trị vào địa chỉ 0x0000000d và có giá trị là 0000000c.
- **Time slot 3:** Ta viết giá trị 11 vào region 1 ở offset là 11. Ta đã biết region 1 nằm trong page 1 bắt đầu ở 256 và theo page table page 1 được map với một frame nằm trong bộ nhớ thứ cấp và đang không hiện diện ở bộ nhớ RAM, do đó ta thực hiện swap với victim page là page 0 và victim frame là frame 0 ở bộ nhớ RAM. Khi này ta điều chỉnh lại page table và map page 1 vào frame 0 của bộ nhớ RAM, sau đó ta ghi giá trị 0000000b vào địa chỉ 0x0000000b.
- **Time slot 4:** Ta đọc giá trị tại region 0 offset là 13. Khi này theo page table thì page 0 chứa region 0 đang map vào một frame ở bộ nhớ thứ cấp và đang không hiện diện ở RAM, do đó ta cần thực hiện swap với victim page là page 1 và victim frame là frame 0 ở bộ nhớ RAM. Khi này ta điều chỉnh lại page table và map page 0 vào frame 0 và đọc giá trị theo địa chỉ là 12.
- **Time slot 5:** Ta đọc giá trị tại region 1 offset là 11. Khi này theo page table thì page 1 chứa region 1 đang map vào một frame ở bộ nhớ thứ cấp và đang không hiện diện ở RAM, do đó ta cần thực hiện swap với victim page là page 0 và victim frame là frame 0 ở bộ nhớ RAM. Khi này ta điều chỉnh lại page table và map page 1 vào frame 0 và đọc giá trị theo địa chỉ là 11.
- **Time slot 6:** Ta viết giá trị 13 vào region 0 ở offset là 0. Khi này page 0 chứa region 0 được map với một frame nằm trong bộ nhớ thứ cấp và đang không hiện diện ở bộ nhớ RAM, do đó ta thực hiện swap với victim page là page 1 và victim frame là frame 0 ở bộ nhớ RAM. Khi này ta điều chỉnh lại page table và map page 0 vào frame 0 của bộ nhớ RAM, sau đó ta ghi giá trị 0000000d vào địa chỉ 0x00000000.
- **Time slot 7:** Ta đọc giá trị tại region 0 offset là 0. Khi này theo page table thì page 0 được map vào frame 0 của bộ nhớ RAM và đang hiện diện ở bộ nhớ RAM, do đó ta đọc giá trị theo địa chỉ là 13.
- **Time slot 8:** Ta giải phóng vùng nhớ region 0; khi này vùng nhớ region 0 sẽ được thêm vào danh sách các vùng nhớ trống và được combine với các vùng nhớ trống liền kề nếu có. Ta có thể thấy vùng nhớ region 0 đã được combine với một vùng nhớ trống trong danh sách các vùng nhớ trống trong kết quả ở trên để tạo nên một vùng nhớ lớn hơn.
- **Time slot 9:** Ta giải phóng vùng nhớ region 1; khi này vùng nhớ region 1 sẽ được thêm vào danh sách các vùng nhớ trống và được combine với các vùng nhớ trống liền kề nếu có. Ta có thể thấy vùng nhớ region 1 đã được combine với một vùng nhớ trống trong danh sách các vùng nhớ trống để tạo ra một vùng nhớ lớn hơn.
- **Time slot 10:** Kết thúc process, dừng CPU.

5 Put It All Together

5.1 Cơ sở lý thuyết

Sau khi tổng hợp hai phần trên, ta đã có được một hệ điều hành đơn giản với một góc nhìn khái quát qua biểu đồ sau.



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 của bạn? Minh họa bằng ví dụ vấn đề của hệ điều hành đơn giản của bạn nếu có?

Trả lời:
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.
Ví dụ: Nếu hai tác vụ cố gắng truy cập đồng thời cùng một tài nguyên mà không đồng bộ hóa phù hợp, điều đó có thể dẫn đến tình trạng chạy đua trong đó hành vi của hệ thống trở nên khó đoán. Hãy xem xét một tình huống trong đó hai process, process 1 và process 2 đều cùng cố gắng truy cập vào một tài nguyên được chia sẻ, chẳng hạn như một biến toàn cục, mà không có sự đồng bộ hóa thích hợp. Cả hai process đều đọc giá trị của biến và tăng giá trị đó, sau đó ghi giá trị mới trở lại biến. Nếu process 1 và

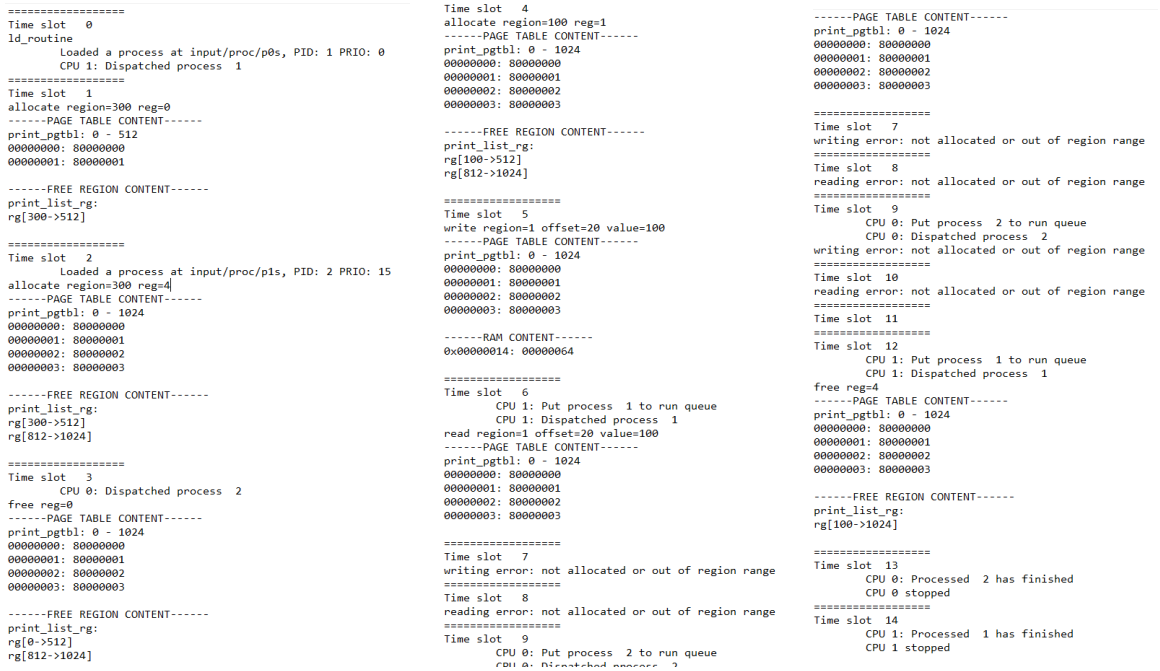
process 2 thực hiện đồng thời và cả hai đều đọc giá trị của biến cùng một lúc, cả hai sẽ nhận được cùng một giá trị. Sau đó, cả hai sẽ tăng giá trị này và ghi lại vào biến. Tuy nhiên, vì cả hai tác vụ đều tăng giá trị lên 1, nên giá trị cuối cùng của biến lẽ ra phải được tăng lên 2, nhưng nó chỉ được tăng lên 1 vì mỗi tác vụ chỉ nhìn thấy giá trị trước đó.

5.3 Kết quả thực thi

Ta sẽ mô phỏng hệ điều hành đơn giản thực thi chương trình bằng cách kiểm thử testcase **os_0_mlq_paging**:

```
6 2 2
1048576 16777216 0 0 0
0 p0s 0
2 p1s 15
```

Output



Hình 9: Output của os_0_mlq_paging

Giải thích kết quả:

- **Time slot 0:** Loader load process 1 vào **ready_queue** và sau đó được đưa vào **CPU 1** tiến hành thực thi.
- **Time slot 1:** Ta thực thi lệnh (*alloc* 300 0) cấp phát vùng nhớ có size là 300 lưu vào region 0. Vì size mỗi page là 256, nên ta cần hai page để lưu. Page table lúc này đang rỗng và không có region nào trống do đó ta cần tăng độ rộng vùng nhớ ảo và tạo hai vùng nhớ trống, mỗi vùng có độ rộng bằng một page. Sau khi tạo, ta cần map page vừa tạo với một frame trong bộ nhớ thực RAM. Trong output ta có thể thấy **page table** tại thời điểm này đã mapping 2 page tương ứng với 2 frame với RAM. Vì hai page có kích thước tổng là 512, ta chỉ cấp phát 300 nên sẽ có region bị dư không sử dụng và ta có thể thấy nó được in ra trong FREE REGION CONTENT.
- **Time slot 2:** Tại thời điểm này, process 2 đã được load lên **ready_queue** và tiếp tục thực hiện lệnh (*alloc* 300 4) cấp phát vùng nhớ có size là 300 lưu vào region 4. Tương tự như cách phân tích ở trên, ta có được bảng page table và các region dư được in ra.

- **Time slot 3:** Process 2 được đưa vào CPU 0 để thực thi, đồng thời ở CPU 1 thực hiện lệnh (**free** 0) giải phóng vùng nhớ tại region 0. Khi đó, rg[0->300] (vùng nhớ ảo từ 0 tới 300) sẽ được đưa vào danh sách các region không sử dụng và sẽ hợp lại với rg[301->512] tạo thành rg[0->512] như output.
- **Time slot 4:** Thực hiện lệnh (*allocate* 100 1) của process 1. Khi đó rg[0->512] (virtual memory area của process 1) đang trống được sử dụng một phần để cấp phát. Output của FREE REGION CONTENT được thay đổi như hình.
- **Time slot 5:** Lệnh (*write* 100 1 20) từ process 1 viết giá trị 100 vào vị trí trong region 1 có offset 20. Khi đó trong RAM tại địa chỉ 0x00000014 đang lưu giá trị 00000064 (giá trị hệ 16 của số 100 hệ 10).
- **Time slot 6:** Process 1 được đưa vào lại hàng đợi và tiếp tục được đưa lên CPU 1 tiếp tục thực thi. Sau đó, thực thi lệnh (*read* 1 20 20) đọc nội dung tương ứng trong vùng nhớ vật lý của region 1 với offset là 20, và ta đọc được giá trị 100 như output.
- **Time slot 7:** Thực thi lệnh (*write* 102 2 20) viết giá trị 102 vào vị trí trong region 2 với offset 20. Tuy nhiên, vì region 2 chưa được cấp phát vùng nhớ, nên không thể thực hiện lệnh này và in ra lỗi writing error: not allocated or out of region range.
- **Time slot 8:** Thực thi lệnh (*read* 2 20 20) đọc nội dung tương ứng trong vùng nhớ vật lý của region 2 với offset là 20. Tuy nhiên vì region 2 chưa được cấp phát vùng nhớ, nên không thể đọc nội dung và in ra lỗi reading error: not allocated or out of region range.
- **Time slot 9:** Process 2 thực thi đủ timeslice quay trở lại hàng đợi, sau đó tiếp tục được đưa vào CPU 0 tiếp tục thực thi. Lệnh (*write* 103 3 20) viết giá trị 103 vào vị trí trong region 3 với offset 20. Tuy nhiên, vì region 3 chưa được cấp phát vùng nhớ, nên không thể thực hiện và in ra lỗi.
- **Time slot 10:** Thực thi lệnh (*read* 3 20 20) đọc nội dung tương ứng trong vùng nhớ vật lý của region 3 với offset là 20. Tuy nhiên, vì region 3 chưa được cấp phát vùng nhớ, nên không thể thực hiện và in ra lỗi.
- **Time slot 12:** Process 1 hoàn thành một timeslice và tiếp tục được đưa vào CPU 1 để tiếp tục thực thi. Lệnh (*free* 4) giải phóng vùng nhớ ở region 4. Vùng nhớ region 4 là rg[512->812] được đưa vào danh sách các region trống không được sử dụng và gộp chung lại với các vùng nhớ có sẵn in ra như output.
- **Time slot 13:** Process 2 hoàn thành thực thi và không còn process nào nên CPU 0 dừng.
- **Time slot 14:** Process 1 hoàn thành thực thi và không còn process nào nên CPU 1 dừng.

Testcase os_1_mlq_paging
Testcase os_1_mlq_paging_small_1K
Testcase os_1_mlq_paging_small_4K
Testcase os_1_singleCPU_mlq
Testcase os_1_singleCPU_mlq_paging
Vì output của các testcase khác rất dài có thể gây loãng báo cáo nên output sẽ được upload riêng trên drive

Toàn bộ các output của phần memory tại đây

5.4 Vấn đề về đồng bộ (Synchronization)

Trong hệ điều hành đơn giản này, ta có sử dụng cơ chế multi-thread để mô phỏng hệ điều hành thực tế chạy chương trình như thế nào. Do đó không thể không xảy ra các trường hợp làm phát sinh race condition, có thể đến từ sự bất đồng bộ của lệnh write, lệnh read hay lênh alloc vì trong hệ điều hành đơn giản này, mọi thao tác trên bộ nhớ ta sử dụng chung một bộ nhớ RAM và dùng chung các bộ nhớ thứ cấp với nhau. Vì thế để đơn giản hóa và tránh đi trường hợp race condition này, nhóm chúng em đã thêm một khóa mutex trước khi và sau khi thực thi một câu lệnh của process, khóa mutex được khai báo như biến tĩnh và được sử dụng chung giữa các thread, được thêm vào như sau:

```
/*
* OTHER IMPLEMENTATIONS HERE
*/
static pthread_mutex_t process_mtx;
/*
* OTHER IMPLEMENTATIONS HERE
*/
static void * cpu_routine(void * args) {
    /*
    * OTHER IMPLEMENTATIONS HERE
    */
    while(1) {
        /*
        * OTHER IMPLEMENTATIONS HERE
        */
        /* Run current process */
        pthread_mutex_lock(&process_mtx);
        run(proc);
        pthread_mutex_unlock(&process_mtx);
        /*
        * OTHER IMPLEMENTATIONS HERE
        */
    }
    /*
    * OTHER IMPLEMENTATIONS HERE
    */
}
/*
* OTHER IMPLEMENTATIONS HERE
*/
int main(int argc, char * argv[]) {
    /*
    * OTHER IMPLEMENTATIONS HERE
    */
    /* Init process mutex */
    pthread_mutex_init(&process_mtx, NULL);
    /*
    * OTHER IMPLEMENTATIONS HERE
    */
}
```

6 Phân công

Họ và tên	Công việc	Hoàn thiện
Trần Nguyễn Thái Bình	Hoàn thiện các hàm virtual machine	100%
Hoàng Đức Nguyên	Cơ sở lý thuyết và trả lời câu hỏi	100%
Nguyễn Phan Hoàng Phúc	Viết báo cáo và kiểm thử các testcase	100%
Đặng Quang Vinh	Hoàn thiện các hàm scheduler	100%
Trương Hoàng Nguyên Vũ	Hoàn thiện các hàm memphy và memory mapping	100%