

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



THÍ NGHIỆM HỆ ĐIỀU HÀNH (CO2018)

BÀI TẬP LỚN

Simple Operating System

Nhóm 8 - L08 - HK222

GV hướng dẫn:	Trần Trương Tuấn Phát	
SV thực hiện:	Trịnh Khải Toàn	2115036
	Lê Xuân Anh	2012592
	Phạm Đình Mạnh Hưng	2113614
	Ngô Văn Phương	2112070

THÀNH PHỐ HỒ CHÍ MINH, 2023



Mục lục

1 Bộ định thời (Scheduler)	5
1.1 Hiện thực	6
1.1.1 Hiện thực file queue.c:	6
1.1.2 Hiện thực file sched.c:	7
1.2 Kiểm tra code	9
1.2.1 Testcase 1	9
1.2.2 Testcase 2	11
1.2.3 Testcase 3	13
1.2.4 Testcase 4	15
1.3 Trả lời câu hỏi	17
1.3.1 Hàng đợi ưu tiên trong bộ định thời	17
2 Quản lý bộ nhớ (Memory Management)	17
2.1 Hiện thực các mục TODO	17
2.1.1 Hàm __alloc()	17
2.1.2 Hàm validate_overlap_vm_area()	18
2.1.3 Hàm alloc_pages_range()	18
2.1.4 Hàm vmmap_page_range()	19
2.1.5 Hàm __free()	19
2.1.6 Hàm find_victim_page()	20
2.1.7 Hàm pg_getpage()	20
2.1.8 Hàm MEMPHY_dump()	21
2.2 Đồng bộ hóa	21
2.3 Kết quả kiểm thử	22
2.3.1 Test case 1	22
2.3.2 Test case 2	25
2.3.3 Test case 3	28
2.4 Trả lời câu hỏi	32
2.4.1 Phân đoạn bộ nhớ	32



2.4.2	Kết hợp phân đoạn bộ nhớ và phân trang bộ nhớ	33
2.4.3	Phân trang đa cấp	34
3	Kết hợp bộ định thời và quản lý bộ nhớ (Put It All Together)	35
3.1	Kết quả kiểm thử	35
3.1.1	Test case 1	35
3.1.2	Test case 2	41
3.2	Trả lời câu hỏi	44
3.2.1	Hậu quả của việc không đồng bộ hóa	44



Danh sách hình vẽ

1	Input - Scheduler Test Case 1	9
2	Output - Scheduler Test Case 1	10
3	Biểu đồ Gantt - Scheduler Test Case 1	11
4	Input - Scheduler Test Case 2	11
5	Output - Scheduler Test Case 2	12
6	Biểu đồ Gantt - Scheduler Test Case 2	13
7	Input - Scheduler Test Case 3	13
8	Output - Scheduler Test Case 3	14
9	Biểu đồ Gantt - Scheduler Test Case 3	15
10	Input - Scheduler Test Case 4	15
11	Output - Scheduler Test Case 4	16
12	Biểu đồ Gantt - Scheduler Test Case 4	16
13	Input - Memory Test Case 1	22
14	Terminal Output - Memory Test Case 1	23
15	Trạng thái bộ nhớ RAM - Memory Test Case 1	24
16	Input - Memory Test Case 2	25
17	Output - Memory Test Case 2	26
18	Trạng thái bộ nhớ RAM - Memory Test Case 2	27
19	Input - Memory Test Case 3	29
20	Input - os_1_singleCPU_mlq_paging	36
21	Biểu đồ Gantt cho test case os_1_singleCPU_mlq_paging	41
22	Input - os_0_mlq_paging	41
23	Biểu đồ Gantt cho test case os_0_mlq_paging	44



Danh sách thành viên & Phân chia công việc

Lớp	Họ và tên	Mã số sinh viên	Công việc	Tỉ lệ công việc
1	Trịnh Khải Toàn	2115036	Scheduler	25%
2	Lê Xuân Anh	2012592	Memory & Trả lời câu hỏi	20%
3	Phạm Đình Mạnh Hưng	2113614	Memory	20%
4	Ngô Văn Phương	2112070	Memory & Scheduler	35%

1 Bộ định thời (Scheduler)

Trong các hệ thống đa nhiệm (multi-tasking) đều có những đặc điểm như:

- Tại một thời điểm trong bộ nhớ có nhiều process.
- Tại mỗi thời điểm chỉ có một process được thực thi

Do đó, cần phải giải quyết vấn đề phân loại và lựa chọn process thực thi sao cho được hiệu quả nhất. Cần có chiến lược định thời CPU.

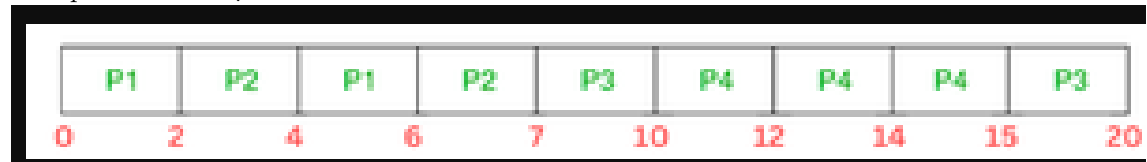
Trong thí nghiệm này chúng ta sẽ hiện thực bộ định thời theo giải thuật multilevel queue scheduling.

Giải thuật multilevel queue scheduling là giải thuật nhiều cấp, chia hàng đợi ra làm nhiều hàng đợi riêng lẻ với mỗi hàng đợi đều có giải thuật định thời của chính nó.

Ví dụ minh họa:

Process	Arrival Time	CPU Burst Time	Queue Number
P1	0	4	1
P2	0	3	1
P3	0	8	2
P4	10	5	1

Kết quả theo đồ thị Gantt:



Giải thích:

-Ta sẽ xem như Queue Number là priority của các process.

Khi arrival time là 0 thì có 3 process P1, P2 và P3 cùng lúc vào hệ thống, nhưng priority của P1 và P2 cao hơn P3 nên hệ thống sẽ ưu tiên P1 và P2, và vì P1 và P2 có chung độ ưu tiên nên ta sẽ thực hiện giải thuật Round Robin.

-Khi P1 và P2 hoàn tất thì P3 sẽ vào hệ thống theo giải thuật First-come-first-serve. Nhưng khi P4 vào hệ thống và P4 có độ ưu tiên cao hơn P3 nên hệ thống sẽ xử lý P3 và P4 theo giải thuật preemptive priority để hoàn tất P4 xong hoàn tất nốt P3.

1.1 Hiện thực

1.1.1 Hiện thực file queue.c:

- Mục đích: viết hàm để thêm proc vào queue (enqueue) và hàm để trả về proc trong queue theo quy tắc FIFO (dequeue).
- Ý tưởng và hiện thực code

– Hàm enqueue():

Về cơ bản hàm enqueue chỉ thêm lại proc được yêu cầu vào cuối ready_queue và tăng kích cỡ của queue lên 1 đơn vị.

Code:

```
void enqueue(struct queue_t * q, struct pcb_t * proc) {
    /* TODO: put a new process to queue [q] */
    if (q->size == MAX_QUEUE_SIZE) // queue full
    {
        #ifdef SCHED_DEBUG
            printf("Queue is full\n");
        #endif
        abort();
    }
    if(proc == NULL){
        #ifdef SCHED_DEBUG
            printf("Enqueue a NULL process!\n");
        #endif
    }
    q->proc[q->size] = proc;
    q->size++;
}
```

– Hàm dequeue():

Mục đích của hàm này là trả về proc có priority lớn nhất trong queue, sau đó xóa proc đó khỏi queue. Ý tưởng: vì queue là 1 kiểu array với mỗi phần tử là 1 proc nên ta sẽ dò tìm proc có priority lớn nhất trong queue để trả về từ đó dồn những proc phía sau nếu có và xóa phần tử cuối và giảm size của queue 1 đơn vị.

Code:

```
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;

    //remove process at the head (FIFO style)
    struct pcb_t* proc = q->proc[0];

    // remove process from q
    for (int i = 0; i < q->size - 1; ++i)
        q->proc[i] = q->proc[i + 1];
    q->size--;

    return proc;
}
```

Giải thích:

-Sau khi gọi hàm thì chương trình sẽ kiểm tra queue cần được lấy proc đó có phải queue rỗng không, nếu có thì hàm sẽ trả về NULL, nếu không thì sẽ tiếp tục.

-Chọn proc đầu tiên trong queue và lưu vào biến proc mới để trả về. Trước khi trả về, chương trình sẽ cập nhật lại queue, dồn các queue phía sau lên đằng trước và giảm kích thước queue xuống một đơn vị.

1.1.2 Hiện thực file sched.c:

- Mục đích: chọn ra được 1 proc từ queue để đưa vào cpu để bắt đầu chạy chương trình và quản lý timeslot của từng queue để tránh tình trạng "starving".

Dựa theo code có sẵn ta nhận thấy được chương trình có 2 hàm cần hoàn thiện trong file sched.c là hàm `get_mlq_proc()`.

- Ý tưởng và hiện thực code

– Hàm `get_mlq_proc()`:

Hàm `get_mlq_proc()` trả về 1 proc từ queue bằng cách sử dụng hàm `dequeue()` trong file `queue.c`. Nhưng phải chọn proc từ queue từ `mlq_queue` và quản lý hợp lý các slot để tránh tình trạng "starving" giữa các queue. Tiêu chí để chọn queue từ `mlq_queue` là bằng các vị trí của các queue trong `mlq_queue` (từ nhỏ đến lớn) và sử dụng lock để bảo vệ dữ liệu.

Trước hết để có thể quản lý các slot của queue thì ta cần cập nhật struct của `queue_t` trong file `queue.h` bằng cách thêm biến int gọi là `numslot` và cập nhật struct của `pcb_t` bằng cách thêm con trỏ `int* numslots` để có thể dễ dàng lưu trữ và truy cập số slot của từng queue và proc.

Tiếp theo, để gán địa chỉ cho con trỏ `numslots` của `pcb_t` ta sẽ cập nhật hàm

add_mlq_proc() trong file sched.c.

Sau đó, ta sẽ cập nhật hàm cpu_routine đúng với giải thuật.

Và cuối cùng là ta sẽ cập nhật hàm get_mlq_proc() như sau.

Code:

```
static int qid; //add current queue index/priority
struct pcb_t * get_mlq_proc(void) {
    struct pcb_t * proc = NULL;
    /*TODO: get a process from PRIORITY [ready_queue].
     * Remember to use lock to protect the queue.
     */
    pthread_mutex_lock(&queue_lock);

    int empty_queue_count = 0;
    while (mlq_ready_queue[qid].numslots <= 0 ||
           empty(&mlq_ready_queue[qid]))
    { // if current queue is out of slot or it is empty

        if (mlq_ready_queue[qid].numslots <= 0 &&
            !empty(&mlq_ready_queue[qid]))
            mlq_ready_queue[qid].numslots = MAX_PRIO - qid;
            // restore current queue's numslot (for future slots)

        ++empty_queue_count;

        if (empty_queue_count == MAX_PRIO) {
            pthread_mutex_unlock(&queue_lock);
            return NULL; // all queues are empty
        }

        // go to next queue
        if (qid == MAX_PRIO - 1)
            qid = 0;
        else
            qid++;
    }

    proc = dequeue(&mlq_ready_queue[qid]);
    pthread_mutex_unlock(&queue_lock);
}
```

```
return proc;  
}
```

Giải thích:

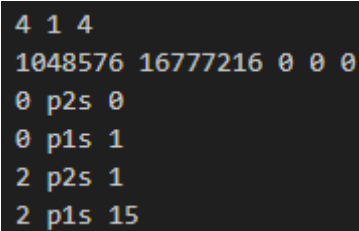
-Trước hết ta sẽ tạo 1 biến static int qid (mặc định là 0 khi bắt đầu chạy chương trình) để lưu lại vị trí trong mlq_queue mà hệ thống đang sử dụng, mục đích của biến này là để nhanh chóng truy cập vào mlq_queue mà không cần phải dò lại những vị trí trước đó.

-Ta sẽ có 1 biến int empty_queue_count để phục vụ cho mục đích dò qua các queue, nếu số queue rỗng bằng với MAX_PRIO thì ta sẽ xem như tất cả các queue trong mlq_ready_queue đều rỗng và trả về NULL. -Cuối cùng là cải thiện hàm sao cho hàm trả về proc từ các queue không rỗng, còn slot và phù hợp với giải thuật multilevel queue scheduling.

1.2 Kiểm tra code

1.2.1 Testcase 1

Input:



```
4 1 4  
1048576 16777216 0 0 0  
0 p2s 0  
0 p1s 1  
2 p2s 1  
2 p1s 15
```

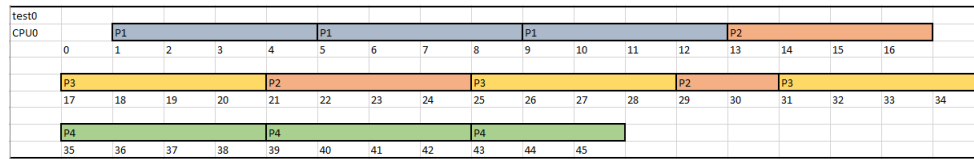
Hình 1: Input - Scheduler Test Case 1

Output:

```
Time slot 0
ld_routine
  Loaded a process at input/proc/p2s, PID: 1 PRIO: 0
Time slot 1
  CPU 0: Dispatched process 1
  Loaded a process at input/proc/p1s, PID: 2 PRIO: 1
Time slot 2
  Loaded a process at input/proc/p2s, PID: 3 PRIO: 1
Time slot 3
  Loaded a process at input/proc/p1s, PID: 4 PRIO: 15
Time slot 4
Time slot 5
  CPU 0: Put process 1 to run queue
  CPU 0: Dispatched process 1
Time slot 6
Time slot 7
Time slot 8
Time slot 9
  CPU 0: Put process 1 to run queue
  CPU 0: Dispatched process 1
Time slot 10
Time slot 11
Time slot 12
Time slot 13
  CPU 0: Processed 1 has finished
  CPU 0: Dispatched process 2
Time slot 14
Time slot 15
Time slot 16
Time slot 17
  CPU 0: Put process 2 to run queue
  CPU 0: Dispatched process 3
Time slot 18
Time slot 19
Time slot 20
Time slot 21
  CPU 0: Put process 3 to run queue
  CPU 0: Dispatched process 2
Time slot 22
Time slot 23
Time slot 24
Time slot 25
  CPU 0: Put process 2 to run queue
  CPU 0: Dispatched process 3
Time slot 26
Time slot 27
Time slot 28
Time slot 29
  CPU 0: Put process 3 to run queue
  CPU 0: Dispatched process 2
Time slot 30
Time slot 31
  CPU 0: Processed 2 has finished
  CPU 0: Dispatched process 3
Time slot 32
Time slot 33
Time slot 34
Time slot 35
  CPU 0: Processed 3 has finished
  CPU 0: Dispatched process 4
Time slot 36
Time slot 37
Time slot 38
Time slot 39
  CPU 0: Put process 4 to run queue
  CPU 0: Dispatched process 4
Time slot 40
Time slot 41
Time slot 42
Time slot 43
  CPU 0: Put process 4 to run queue
  CPU 0: Dispatched process 4
Time slot 44
Time slot 45
  CPU 0: Processed 4 has finished
  CPU 0 stopped
```

Hình 2: Output - Scheduler Test Case 1

Biểu đồ Gantt:



Hình 3: Biểu đồ Gantt - Scheduler Test Case 1

Đối với testcase 1, chương trình cần xử lý 4 proc với timeslice là 4 đơn vị thời gian bằng 1 cpu, đầu tiên vào hệ thống sẽ ưu tiên proc dựa trên giải thuật First-come First-serve, và nếu có từ 2 proc trở lên cùng đến cùng 1 thời điểm thì hệ thống ưu tiên chọn proc có độ ưu tiên cao hơn. Và khi có từ 2 proc trở lên có cùng độ ưu tiên trong queue thì hệ thống sẽ sử dụng giải thuật round-robin để thực thi xen kẽ các proc đến khi các proc hoàn thành hoặc queue ấy hết timeslot. Trường hợp hết timeslot, hệ thống sẽ kiểm tra các queue khác nếu tồn tại queue khác mà vẫn còn proc thì hệ thống sẽ chuyển qua thực thi các queue khác bỏ lại queue chưa hoàn thành, queue chưa hoàn thành sẽ được cấp lại slot và sẽ được tiếp tục thực thi khi hệ thống đi hết 1 vòng các queue.

1.2.2 Testcase 2

Input:

```
2 2 4
1048576 16777216 0 0 0
0 p2s 1
1 p1s 10
2 p2s 1
2 p1s 15
```

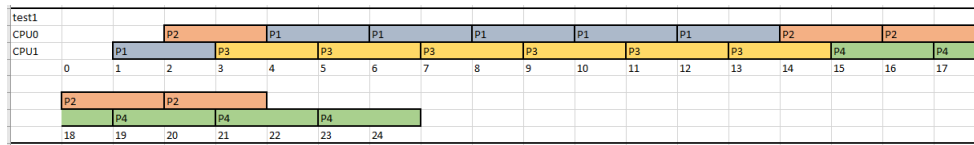
Hình 4: Input - Scheduler Test Case 2

Output:

```
Time slot 0
ld_routine
  Loaded a process at input/proc/p2s, PID: 1 PRI0: 1
Time slot 1
  CPU 1: Dispatched process 1
  Loaded a process at input/proc/p1s, PID: 2 PRI0: 10
Time slot 2
  CPU 0: Dispatched process 2
  Loaded a process at input/proc/p2s, PID: 3 PRI0: 1
Time slot 3
  CPU 1: Put process 1 to run queue
  CPU 1: Dispatched process 3
  Loaded a process at input/proc/p1s, PID: 4 PRI0: 15
Time slot 4
  CPU 0: Put process 2 to run queue
  CPU 0: Dispatched process 1
Time slot 5
  CPU 1: Put process 3 to run queue
  CPU 1: Dispatched process 3
Time slot 6
  CPU 0: Put process 1 to run queue
  CPU 0: Dispatched process 1
Time slot 7
  CPU 1: Put process 3 to run queue
  CPU 1: Dispatched process 3
Time slot 8
  CPU 0: Put process 1 to run queue
  CPU 0: Dispatched process 1
Time slot 9
  CPU 1: Put process 3 to run queue
  CPU 1: Dispatched process 3
Time slot 10
  CPU 0: Put process 1 to run queue
  CPU 0: Dispatched process 1
Time slot 11
  CPU 1: Put process 3 to run queue
  CPU 1: Dispatched process 3
Time slot 12
  CPU 0: Put process 1 to run queue
  CPU 0: Dispatched process 1
Time slot 13
  CPU 1: Put process 3 to run queue
  CPU 1: Dispatched process 3
Time slot 14
  CPU 0: Processed 1 has finished
  CPU 0: Dispatched process 2
Time slot 15
  CPU 1: Processed 3 has finished
  CPU 1: Dispatched process 4
Time slot 16
  CPU 0: Put process 2 to run queue
  CPU 0: Dispatched process 2
Time slot 17
  CPU 1: Put process 4 to run queue
  CPU 1: Dispatched process 4
Time slot 18
  CPU 0: Put process 2 to run queue
  CPU 0: Dispatched process 2
Time slot 19
  CPU 1: Put process 4 to run queue
  CPU 1: Dispatched process 4
Time slot 20
  CPU 0: Put process 2 to run queue
  CPU 0: Dispatched process 2
Time slot 21
  CPU 1: Put process 4 to run queue
  CPU 1: Dispatched process 4
Time slot 22
  CPU 0: Processed 2 has finished
  CPU 0 stopped
Time slot 23
  CPU 1: Put process 4 to run queue
  CPU 1: Dispatched process 4
Time slot 24
Time slot 25
  CPU 1: Processed 4 has finished
  CPU 1 stopped
```

Hình 5: Output - Scheduler Test Case 2

Biểu đồ Gantt:



Hình 6: Biểu đồ Gantt - Scheduler Test Case 2

Testcase 2, chạy với giải thuật tương tự testcase 1 nhưng với số lượng cpu là 2.

1.2.3 Testcase 3

Input:

```
4 4 8
1048576 16777216 0 0 0
0 p2s 1
1 p1s 10
2 p2s 1
2 p1s 15
6 p2s 0
6 p1s 10
8 p2s 12
10 p1s 1
```

Hình 7: Input - Scheduler Test Case 3

Output:

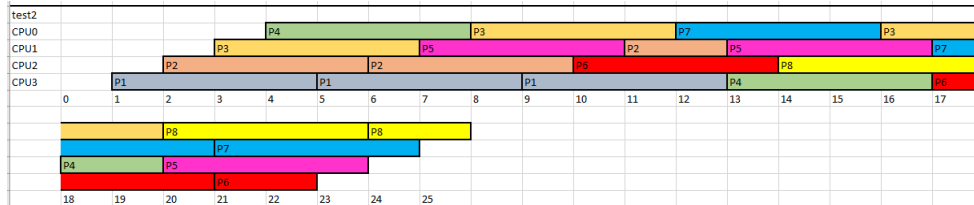
```
Time slot 0
ld_routine
  Loaded a process at input/proc/p2s, PID: 1 PRIO: 1
Time slot 1
  CPU 3: Dispatched process 1
  Loaded a process at input/proc/p1s, PID: 2 PRIO: 10
Time slot 2
  CPU 2: Dispatched process 2
  Loaded a process at input/proc/p2s, PID: 3 PRIO: 1
Time slot 3
  CPU 1: Dispatched process 3
  Loaded a process at input/proc/p1s, PID: 4 PRIO: 15
Time slot 4
  CPU 0: Dispatched process 4
Time slot 5
  CPU 3: Put process 1 to run queue
  CPU 3: Dispatched process 1
Time slot 6
  CPU 2: Put process 2 to run queue
  CPU 2: Dispatched process 2
  Loaded a process at input/proc/p2s, PID: 5 PRIO: 0
Time slot 7
  CPU 1: Put process 3 to run queue
  CPU 1: Dispatched process 5
  Loaded a process at input/proc/p1s, PID: 6 PRIO: 10
Time slot 8
  CPU 0: Put process 4 to run queue
  CPU 0: Dispatched process 3
  Loaded a process at input/proc/p2s, PID: 7 PRIO: 12
Time slot 9
  CPU 3: Put process 1 to run queue
  CPU 3: Dispatched process 1
```

```
Time slot 10
  CPU 2: Put process 2 to run queue
  CPU 2: Dispatched process 6
  Loaded a process at input/proc/p1s, PID: 8 PRIO: 1
Time slot 11
  CPU 1: Put process 5 to run queue
  CPU 1: Dispatched process 2
Time slot 12
  CPU 0: Put process 3 to run queue
  CPU 0: Dispatched process 7
Time slot 13
  CPU 3: Processed 1 has finished
  CPU 3: Dispatched process 4
  CPU 1: Processed 2 has finished
  CPU 1: Dispatched process 5
Time slot 14
  CPU 2: Put process 6 to run queue
  CPU 2: Dispatched process 8
Time slot 15
Time slot 16
  CPU 0: Put process 7 to run queue
  CPU 0: Dispatched process 3
Time slot 17
  CPU 3: Put process 4 to run queue
  CPU 3: Dispatched process 6
  CPU 1: Put process 5 to run queue
  CPU 1: Dispatched process 7
Time slot 18
  CPU 2: Put process 8 to run queue
  CPU 2: Dispatched process 4
Time slot 19
```

```
Time slot 20
  CPU 2: Processed 4 has finished
  CPU 2: Dispatched process 5
  CPU 0: Processed 3 has finished
  CPU 0: Dispatched process 8
Time slot 21
  CPU 3: Put process 6 to run queue
  CPU 3: Dispatched process 6
  CPU 1: Put process 7 to run queue
  CPU 1: Dispatched process 7
Time slot 22
Time slot 23
  CPU 3: Processed 6 has finished
  CPU 3 stopped
Time slot 24
  CPU 2: Processed 5 has finished
  CPU 2 stopped
  CPU 0: Put process 8 to run queue
  CPU 0: Dispatched process 8
Time slot 25
  CPU 1: Processed 7 has finished
  CPU 1 stopped
Time slot 26
  CPU 0: Processed 8 has finished
  CPU 0 stopped
```

Hình 8: Output - Scheduler Test Case 3

Biểu đồ Gantt:



Hình 9: Biểu đồ Gantt - Scheduler Test Case 3

Testcase 3, chạy với giải thuật tương tự testcase 1 nhưng với số lượng cpu là 4.

1.2.4 Testcase 4

Input:

```
4 2 3
1048576 16777216 0 0 0
0 p2s 137
2 p2s 135
2 p1s 135
```

Hình 10: Input - Scheduler Test Case 4

Output:


```

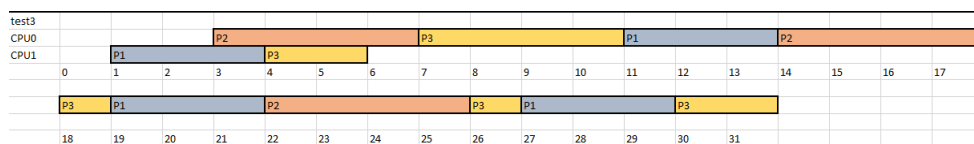
Time slot 0
ld_routine
  Loaded a process at input/proc/p2s, PID: 1 PRIO: 137
Time slot 1
  CPU 1: Dispatched process 1
Time slot 2
  Loaded a process at input/proc/p2s, PID: 2 PRIO: 135
Time slot 3
  CPU 0: Dispatched process 2
  Loaded a process at input/proc/p1s, PID: 3 PRIO: 135
Time slot 4
  CPU 1: Put process 1 to run queue
  CPU 1: Dispatched process 3
Time slot 5
Time slot 6
  CPU 1: Put process 3 to run queue
  CPU 1 stopped
Time slot 7
  CPU 0: Put process 2 to run queue
  CPU 0: Dispatched process 3
Time slot 8
Time slot 9
Time slot 10
Time slot 11
  CPU 0: Put process 3 to run queue
  CPU 0: Dispatched process 1
Time slot 12
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
Time slot 17
Time slot 18
  CPU 0: Put process 2 to run queue
  CPU 0: Dispatched process 3
Time slot 19
  CPU 0: Put process 3 to run queue
  CPU 0: Dispatched process 1
Time slot 20
Time slot 21
Time slot 22
  CPU 0: Put process 1 to run queue
  CPU 0: Dispatched process 2
Time slot 23
Time slot 24
Time slot 25
Time slot 26
  CPU 0: Processed 2 has finished
  CPU 0: Dispatched process 3
Time slot 27
  CPU 0: Put process 3 to run queue
  CPU 0: Dispatched process 1
Time slot 28
Time slot 29
Time slot 30
  CPU 0: Processed 1 has finished
  CPU 0: Dispatched process 3
Time slot 31
Time slot 32
  CPU 0: Processed 3 has finished
  CPU 0 stopped

```

Hình 11: Output - Scheduler Test Case 4

Biểu đồ Gantt:



Hình 12: Biểu đồ Gantt - Scheduler Test Case 4

-Nếu nhìn vào biểu đồ Gantt ta có thể thấy testcase 4 này có phần hơi khác so với những testcase trước đó, đặc biệt là proc 2 và 3. Timeslice trong testcase này là 4, prio của proc 2 và 3 là 135, nên cả 2 proc đều thuộc queue có prio là 135, và timeslot cho cả 2 proc là $140 - 135 = 5$ đơn vị thời gian. Chính vì lý do đó khi hệ thống chạy hết 4 đơn vị thời gian theo timeslice cho proc 2 thì queue 135 chỉ còn 1 đơn vị thời gian để chạy proc 3 trước khi đổi sang queue khác.

1.3 Trả lời câu hỏi

1.3.1 Hàng đợi ưu tiên trong bộ định thời

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

Trả lời:

Ưu điểm của priority queue:

- Tác vụ có thứ tự ưu tiên cao hơn trong Priority Queue sẽ được xếp lên trước và ưu tiên sử dụng trước giúp tăng hiệu quả xử lý.
- Dễ sử dụng và tác vụ có mức độ ưu tiên cao hơn sẽ diễn ra trước giúp tiết kiệm thời gian.
- Các tác vụ ưu tiên sẽ được giải quyết trước làm giảm nguy cơ hệ thống bị treo hoặc đóng băng vì các tác vụ quá tải.

Ví dụ: Trong OS, các tiến trình có thể có các độ ưu tiên sau đây:

- Độ ưu tiên cao: P1
- Độ ưu tiên trung bình: P2
- Độ ưu tiên thấp: P3

Tiến trình sẽ được thực hiện theo độ ưu tiên cao - trung bình - thấp.

Nếu một tiến trình mới là "P4" có độ ưu tiên là "thấp", nó sẽ được thêm vào hàng đợi sau tất cả các tiến trình đang có độ ưu tiên cao hơn nó, và chỉ khi chúng đã hoàn thành thì tiến trình mới này mới được thực hiện tiếp theo.

2 Quản lý bộ nhớ (Memory Management)

2.1 Hiện thực các mục TODO

2.1.1 Hàm `__alloc()`

- **Yêu cầu (TODO):** Hiện thực quá trình xử lý trong trường hợp không lấy được `vm_region` trống trong `vm_area` khi cấp phát bộ nhớ với hàm `__alloc()`
- **Cách hiện thực:**

Tăng giá trị của `sbrk` để chứa `vm_region` mới. Nếu sau khi tăng `sbrk > vm_end`, sử dụng hàm phụ `inc_vma_limit()` để tăng giới hạn của `vm_area` bằng cách tăng giá trị `vm_end` và thực hiện cấp phát và ánh xạ các page mới. Các bước thực hiện như sau:

1. Dùng hàm `get_vma_by_num` để lấy `vm_area` hiện tại
2. Lưu lại giá trị `sbrk` cũ.

3. Kiểm tra xem nếu tăng `sbrk` thì nó có lớn hơn `vm_end` hay không. Nếu có, gọi hàm `inc_vma_limit()` để thực hiện tăng giới hạn và cấp phát page mới. Giá trị gia tăng cần truyền vào hàm là `sbrk + size - vm_end` với `size` là kích thước `vm_region` cần cấp phát.

Lưu ý: Trong hàm `__alloc()`, nhóm đã bổ sung thêm các lệnh để khởi tạo giá trị cho toàn bộ vùng nhớ mới được cấp phát, cụ thể là "0".

- **Code:** Xem thêm trong source code, đường dẫn `src/mm-vm.c`

2.1.2 Hàm `validate_overlap_vm_area()`

- **Yêu cầu (TODO):** Kiểm tra xem `vm_area` được cho có bị chồng lên các `vm_area` khác hay không.
- **Cách hiện thực:** Ta thực hiện duyệt qua toàn bộ các `vm_area` của process hiện tại. Tại mỗi bước duyệt, dùng macro `OVERLAP` để kiểm tra sự chồng chéo lẫn nhau của hai vùng nhớ, được định nghĩa trong `include/mm.h` như sau:

```
#define OVERLAP(x1,x2,y1,y2) (((y2-x1)*(x2-y1)>0)?1:0)
```

- **Code:** Xem thêm trong source code, đường dẫn `src/mm-vm.c`

2.1.3 Hàm `alloc_pages_range()`

- **Yêu cầu (TODO):** Hiện thực quá trình tìm frame trống trong RAM để ánh xạ với các page chứa vùng nhớ cần cấp phát.
- **Cách hiện thực:** Đối với trường hợp lấy được frame trống từ RAM, ta chỉ cần thêm nó vào danh sách frame (tham số `frm_lst`). Tuy nhiên, nếu không thể lấy được frame trống từ RAM, ta cần thực hiện swap để lấy frame trống trước khi thêm vào danh sách frame. Các bước xử lý khi không lấy được frame trống như sau:
 1. Gọi hàm `find_victim_page()` để chọn ra page bị thay thế (victim page).
 2. Tra bảng phân trang của process, lấy frame number của victim page (victim frame) bằng macro `PAGING_PTE_FPN`.
 3. Lấy frame trống từ MEMSWAP (swap frame). Nếu như MEMSWAP hiện tại (`active_mswp`) đã đầy thì cần chuyển qua các MEMSWAP khác để tìm. Nếu tất cả các MEMSWAP đều đầy thì trả về lỗi.
 4. Gọi hàm `__swap_cp_page()` để swap-out victim page trong victim frame vào swap frame. Sau bước này ta có được frame trống là victim frame.

5. Gọi hàm `pte_set_swap()` để cập nhật lại bảng phân trang, vị trí ứng với victim page.

- **Code:** Xem thêm trong source code, đường dẫn `src/mm.c`

2.1.4 Hàm `vmap_page_range()`

- **Yêu cầu (TODO):** Ánh xạ các frame trống lấy được từ hàm `alloc_pages_range` vào các page chứa vùng nhớ mới được cấp phát và trả về vùng nhớ được ánh xạ.
- **Cách hiện thực:** Vùng nhớ được ánh xạ sẽ nằm trong khoảng `[addr, addr + pgnum * PAGING_PAGESZ]` với `addr` là địa chỉ khởi đầu của vùng ánh xạ, `pgnum` là số page chứa vùng nhớ mới được cấp phát và `PAGING_PAGESZ` là kích thước page. Các bước thực hiện:

1. Khai báo các iterator `pgit` và `fpit`.
2. Duyệt qua các page và frame trống tương ứng. Ở mỗi lần duyệt:
 - Khởi tạo một page table entry mới bằng hàm `init_pte()`, truyền các tham số cần thiết.
 - Cập nhật lại bảng phân trang, vị trí page hiện tại.
 - Thêm page được ánh xạ vào FIFO queue để lưu vết các lần truy xuất page.

- **Code:** Xem thêm trong source code, đường dẫn `src/mm.c`

2.1.5 Hàm `__free()`

- **Yêu cầu (TODO):** Quản lý vùng nhớ `vm_region` sau khi nó được giải phóng.

- **Cách hiện thực:**

Các bước thực hiện:

1. Kiểm tra tính hợp lệ của `vm_region` ID. Sau đó dùng ID đó để lấy `vm_region` tương ứng cần giải phóng.
2. Tạo một `vm_region` node mới, deep copy `vm_region` đó vào node này.
3. Xóa `vm_region` đó khỏi symbol table.
4. Thêm node chứa thông tin của `vm_region` đó vào danh sách `vm_region` trống.

- **Code:** Xem thêm trong source code, đường dẫn `src/mm-vm.c`

- **Lưu ý:** Trong hàm có sử dụng thêm lệnh để gán kí hiệu rỗng `"_"` cho vùng nhớ được giải phóng, nhằm mục đích dễ quan sát và gỡ lỗi.

2.1.6 Hàm `find_victim_page()`

- **Yêu cầu (TODO):** Hiện thực cơ chế chọn page để thay thế (victim page) theo lý thuyết.
- **Cách hiện thực:** Ta sử dụng một giải thuật thay trang đơn giản là FIFO (First-In-First-Out). Theo đó, ta sử dụng FIFO queue (`fifo_pgn`) như một chuỗi tham chiếu (reference string). Vì chuỗi này luôn được thêm vào ở đầu danh sách nên theo FIFO ta sẽ chọn phần tử cuối để làm victim page. Sau khi chọn ta xóa nó khỏi FIFO queue.
- **Code:** Xem thêm trong source code, đường dẫn `src/mm-vm.c`

2.1.7 Hàm `pg_getpage()`

- **Yêu cầu (TODO):** Hiện thực quá trình lấy page từ bộ nhớ RAM hoặc SWAP để truy xuất dữ liệu.
- **Cách hiện thực:**
 1. Lấy page table entry (PTE) của page cần truy xuất (target page) trong bảng phân trang.
 2. Dựa vào PTE xác định xem target page có nằm trong RAM không. Nếu có, truy xuất thông tin frame number của page trong PTE và kết thúc. Nếu không thì target page sẽ nằm trong MEMSWAP, ta thực hiện các bước tiếp theo.
 3. Dựa vào PTE ta lấy được swap frame của target page (target frame) từ trường dữ liệu swap offset sử dụng macro `PAGING_PTE_SWPOFF`.
 4. Dựa vào PTE ta xác định được MEMSWAP chứa target page từ trường dữ liệu swap type sử dụng macro `PAGING_PTE_SWPTYP`.
 5. Gọi hàm `MEMPHY_get_freelfp()` để lấy frame trống từ RAM với mục đích đưa target page vào để truy xuất. Ta có 2 trường hợp sau:
 - Nếu lấy được frame trống từ RAM: Swap-in target page vào frame trống bằng hàm `__swap_cp_page()`, sau đó cập nhật PTE của target page bằng hàm `pte_set_fpn()` và cập nhật vào bảng phân trang. Cuối cùng là thêm target page vào FIFO queue.
 - Nếu không lấy được frame trống từ RAM: Ta chọn victim page bằng hàm `find_victim_page()`, truy xuất thông tin frame (victim frame) từ PTE của nó. Sau đó, tìm frame trống từ các MEMSWAP (swap frame) nếu có. Thực hiện swap-out victim page vào swap frame, swap-in target page vào victim frame và cập nhật các PTE của các page đó rồi đưa vào bảng phân trang. Cuối cùng là thêm target page vào FIFO queue.
 6. Sau bước trên ta thu được frame chứa target page là victim frame.
- **Code:** Xem thêm trong source code, đường dẫn `src/mm-vm.c`

2.1.8 Hàm MEMPHY_dump()

- **Yêu cầu (TODO):** Hiện thị trạng thái của bộ nhớ RAM/SWAP để phục vụ cho việc theo dõi quá trình thay đổi dữ liệu.
- **Cách hiện thực:** Vì kích thước RAM/SWAP có thể rất lớn nên cần dump dữ liệu của nó vào các file riêng (`ram.txt`, `swp0.txt`, `swp1.txt`, `swp2.txt`, `swp3.txt...`). Bố cục dữ liệu của RAM/SWAP sẽ được phân thành các frame và được chú thích bằng số frame và khoảng địa chỉ. Dữ liệu trong mỗi frame sẽ được in ra theo dạng bảng.
- **Code:** Xem thêm trong source code, đường dẫn `src/mm-memphy.c`
- **Lưu ý:** Dữ liệu đặc biệt trong RAM/SWAP bao gồm
 - `_`: Dữ liệu trống (NULL) hoặc không sử dụng
 - `0`: Dữ liệu đã được cấp phát (giá trị ban đầu)

2.2 Đồng bộ hóa

Liên quan đến quản lý bộ nhớ, các process có sử dụng chung các shared resource là MEMRAM và MEMSWAP. Do đó các thao tác liên quan đến việc truy xuất bộ nhớ của MEMRAM và MEMSWAP cần được bảo vệ để tránh hiện tượng race condition. Các hàm chứa các thao tác đó là:

- `MEMPHY_mv_csr()`
- `MEMPHY_seq_read()` và `MEMPHY_read()`
- `MEMPHY_seq_write()` và `MEMPHY_write()`
- `MEMPHY_format()`
- `MEMPHY_get_freefp()`
- `MEMPHY_put_freefp()`
- `MEMPHY_dump()`

Ta thực hiện đồng bộ hóa như sau:

1. Khai báo thêm khóa mutex (kiểu dữ liệu `pthread_mutex_t`) bên trong `memphy_struct`.
2. Gọi hàm khởi tạo mutex `pthread_mutex_init(&mp->mutex, NULL)` trong hàm `init_memphy()`.
3. Đặt các lệnh `pthread_mutex_lock(&mp->mutex)` và `pthread_mutex_unlock(&mp->mutex)` tại đầu và cuối critical section của các hàm trên.

2.3 Kết quả kiểm thử

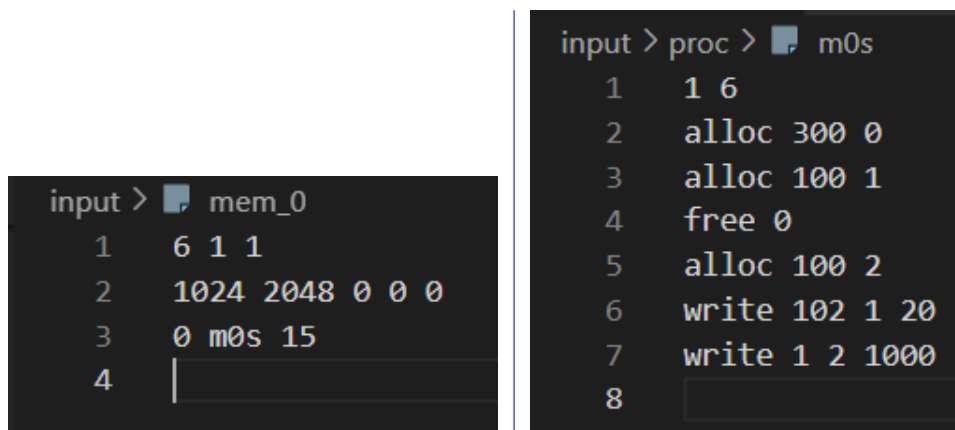
Sau đây là một số test case mà nhóm đã sử dụng để kiểm tra các hoạt động của Virtual Memory Engine. Các test case này chỉ sử dụng một CPU và chỉ chạy duy nhất một process không chứa lệnh `calc` (tránh đụng chạm nhiều tới scheduler).

Lưu ý: Để dễ quan sát, nhóm đã sử dụng ký tự "_" để biểu thị ô nhớ trống chưa được cấp phát và ký tự "0" để biểu thị ô nhớ đã được cấp phát.

2.3.1 Test case 1

Đây là một test case đơn giản để kiểm tra các thao tác đơn giản của việc quản lý bộ nhớ. Test case này chỉ làm việc với RAM với kích thước 1024 byte. Process được chọn để thực thi là `m0s` (có sẵn trong source code ban đầu).

- **Input:** `input/mem_0` và `input/proc/m0s`



```
input > mem_0
1 6 1 1
2 1024 2048 0 0 0
3 0 m0s 15
4

input > proc > m0s
1 1 6
2 alloc 300 0
3 alloc 100 1
4 free 0
5 alloc 100 2
6 write 102 1 20
7 write 1 2 1000
8
```

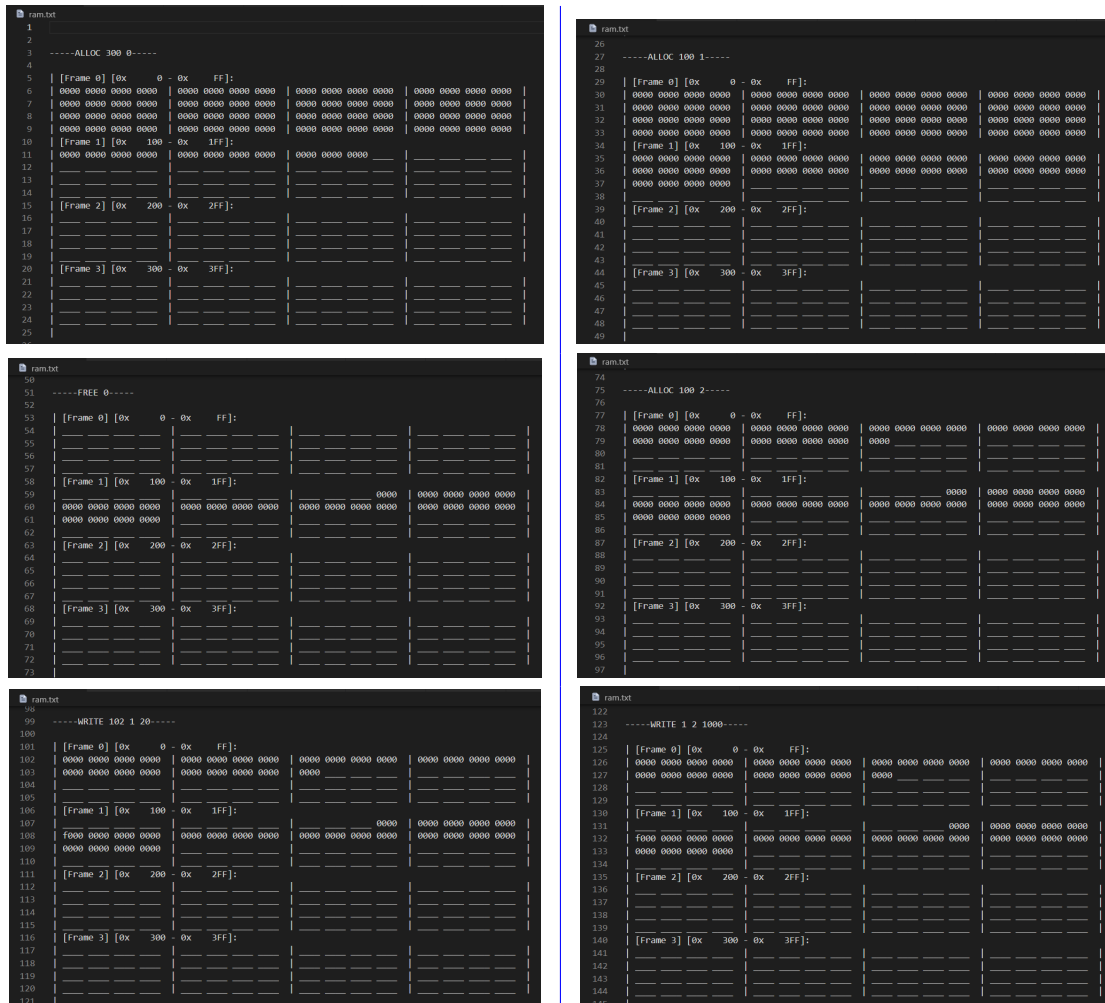
Hình 13: Input - Memory Test Case 1

- **Output trong terminal**

```
bash - ossim_test + v [icon] [icon]
• phuong@admin:~/OSAssignment/ossim_test$ ./os mem_0
ld_routine
Time slot 0
    Loaded a process at input/proc/m0s, PID: 1 PRIO: 15
Time slot 1
    CPU 0: Dispatched process 1
print_pgtbl: 0 - 512
00000000: 80000000
00000004: 80000001
Time slot 2
print_pgtbl: 0 - 512
00000000: 80000000
00000004: 80000001
Time slot 3
print_pgtbl: 0 - 512
00000000: 80000000
00000004: 80000001
Time slot 4
print_pgtbl: 0 - 512
00000000: 80000000
00000004: 80000001
Time slot 5
write region=1 offset=20 value=102
print_pgtbl: 0 - 512
00000000: 80000000
00000004: 80000001
Time slot 6
write region=2 offset=1000 value=1
print_pgtbl: 0 - 512
00000000: 80000000
00000004: 80000001
Time slot 7
    CPU 0: Processed 1 has finished
    CPU 0 stopped
○ phuong@admin:~/OSAssignment/ossim_test$
```

Hình 14: Terminal Output - Memory Test Case 1

- Trạng thái của bộ nhớ RAM sau từng lệnh: ram.txt



Hình 15: Trang thái bộ nhớ RAM - Memory Test Case 1

- Giải thích về output:

- **alloc 300 0**: Cấp phát 300 byte cho `vm_region 0`. Vì hiện tại chưa có `vm_region` trống nên chương trình sẽ tăng `sbrk`. Vì lượng cần tăng là 300 - trải dài qua 2 page nên `vm_end` cũng cần phải tăng tương ứng là 512. Chương trình thực hiện ánh xạ 2 page mới được cấp: Page 0 - Frame 0 và Page 1 - Frame 1. Từ đó ta cập nhật bảng phân trang:

* PTE page 0: 0x80000000 tương ứng với bit present = 1 và frame number = 0

* PTE page 1: 0x80000001 tương ứng với bit present = 1 và frame number = 1

Reference string hiện tại là: 1, 0

Sau khi ánh xạ xong thì 300 byte sẽ được cấp phát tại page 0, 1 trong bộ nhớ ảo và tương ứng là frame 0, 1 trong bộ nhớ vật lý RAM. Để thấy trong RAM các ô nhớ được cấp phát có địa chỉ từ 0x000 - 0x12B.

- **alloc 100 1:** Cấp phát 100 byte cho `vm_region 1`. Thực hiện tăng `sbrk` tương tự. Tuy nhiên, vì page 1 còn đủ bộ nhớ (212 byte) nên 100 ô nhớ tiếp theo sẽ được cấp phát mà không cần phải tăng giới hạn và cấp phát page mới. Khoảng địa chỉ được cấp phát như trong file dump là 0x12C - 0x18F.
- **free 0:** Giải phóng vùng nhớ tại `vm_region 0`. 300 byte được cấp phát ở lệnh trên sẽ được giải phóng và `vm_region 0` được đưa vào danh sách frame trống.
- **alloc 100 2:** Cấp phát 100 byte cho `vm_region 2`. Ta hiện có `vm_region` trống là 300 byte của `vm_region 0` đã bị giải phóng trước đó, nên ta sẽ sử dụng 100 byte trong vùng này để cấp phát. Khoảng địa chỉ được cấp phát là 0x000 - 0x063.
- **write 102 1 20:** Ghi dữ liệu 102 vào `vm_region 1`, tại ô nhớ thứ 20. Ta tính được địa chỉ của ô nhớ cần ghi là $0x12C + 20 = 0x140$. Dữ liệu cần ghi chính là kí hiệu "f" tương ứng với mã ASCII 102.
- **write 1 2 1000:** Ghi dữ liệu 1 vào `vm_region 2`, tại ô nhớ thứ 1000. Ta tính được địa chỉ ô nhớ cần ghi là $0x000 + 1000 = 0x3E8$. Vì ô nhớ này vượt quá giới hạn của `vm_region 2` là 0x063 nên lệnh này sẽ bị lỗi và không làm gì cả.

2.3.2 Test case 2

- **Input** `mem_1` và `input/proc/m1s`

```
input > mem_1
1 6 1 1
2 1024 2048 0 0 0
3 0 m1s 15
4 |
```

```
input > proc > m1s
1 1 6
2 alloc 300 0
3 alloc 100 1
4 free 0
5 alloc 100 2
6 free 2
7 free 1
8
```

Hình 16: Input - Memory Test Case 2

- **Output:**

```
bash - ossim_test + v
phuong@admin:~/OSAssignment/ossim_test$ ./os mem_1
Time slot 0
ld_routine
    Loaded a process at input/proc/m1s, PID: 1 PRIO: 15
Time slot 1
    CPU 0: Dispatched process 1
print_pgtbl: 0 - 512
00000000: 80000000
00000004: 80000001
Time slot 2
print_pgtbl: 0 - 512
00000000: 80000000
00000004: 80000001
Time slot 3
print_pgtbl: 0 - 512
00000000: 80000000
00000004: 80000001
Time slot 4
print_pgtbl: 0 - 512
00000000: 80000000
00000004: 80000001
Time slot 5
print_pgtbl: 0 - 512
00000000: 80000000
00000004: 80000001
Time slot 6
print_pgtbl: 0 - 512
00000000: 80000000
00000004: 80000001
Time slot 7
    CPU 0: Processed 1 has finished
    CPU 0 stopped
phuong@admin:~/OSAssignment/ossim_test$
```

Hình 17: Output - Memory Test Case 2

- Trạng thái bộ nhớ RAM sau mỗi lệnh:



Hình 18: Trạng thái bộ nhớ RAM - Memory Test Case 2

• Giải thích cho output:

- **alloc 300 0:** Cấp phát 300 byte tại **vm_region 0**. Tăng giới hạn **sbrk** (**sbrk = 300**). Vì giới hạn này tràn qua **vm_end** 1 khoảng 2 page nên **vm_end** được tăng tương ứng (**vm_end = 512**). Các frame trống trong RAM được lựa chọn là 0 và 1 và được ánh xạ như sau:

- * Page 1 ánh xạ với Frame 1, PTE: 0x80000001 (present = 1, frame = 1)
- * Page 0 ánh xạ với Frame 0, PTE: 0x80000001 (present = 1, frame = 0)

Reference string: 1, 0

- **alloc 100 1:** Cấp phát 100 byte tại **vm_region 1**. Tăng giới hạn để lấy vùng nhớ mới (**sbrk = 400**). Vì **sbrk** không bị tràn qua page mới nên việc cấp phát diễn ra bình thường mà không cần ánh xạ thêm.

- **free 0:** Giải phóng 300 byte của `vm_region 0`. Đưa nó vào danh sách `vm_region` trống.
- **alloc 100 2:** Cấp phát 200 byte tại `vm_region 2`. Hiện tại ta có vùng nhớ trống đủ để cấp phát cho nó là 300 byte đã được giải phóng trước đó. Như vậy, ta sử dụng 200 byte đầu tiên của vùng đó để cấp phát, không cần tăng giới hạn.
- **free 2:** Giải phóng 200 byte của `vm_region 0`. Đưa nó vào danh sách `vm_region` trống.
- **free 1:** Giải phóng 100 byte của `vm_region 0`. Đưa nó vào danh sách `vm_region` trống.

2.3.3 Test case 3

Dưới đây, ta sẽ sử dụng một testcase phức tạp hơn để kiểm tra hoạt động swapping giữa RAM và SWAP. Ở đây ta sử dụng một process mới là `m2s` (chưa có sẵn trong source code). Kích thước RAM là 1024 byte, SWAP 0 là 2048 byte.

- **Input:** `input/mem_2` và `input/proc/m2s`

```

input > mem_2
1 6 1 1
2 1024 2048 0 0 0
3 0 m2s 15
4

```

```

input > proc > m2s
1 1 25
2 alloc 384 0
3 alloc 128 1
4 alloc 64 2
5 alloc 320 3
6 free 0
7 free 2
8 write 49 1 0
9 write 49 1 127
10 write 51 3 0
11 write 51 3 319
12 alloc 576 6
13 alloc 448 4
14 alloc 128 5
15 alloc 384 0
16 alloc 64 2
17 write 52 4 0
18 write 52 4 447
19 write 54 6 0
20 write 54 6 575
21 write 50 2 0
22 write 50 2 63
23 write 53 5 0
24 write 53 5 127
25 write 48 0 0
26 write 48 0 383
27

```

Hình 19: Input - Memory Test Case 3

- Output trong Terminal:

Vì output quá dài nên không thể đưa vào báo cáo, xem thêm trong đường link <https://drive.google.com/drive/folders/10cJX-Tdb6ehenUQV3--fZCoWLNvdGseE?usp=sharing>

- Trạng thái của bộ nhớ RAM và SWAP 0:

Vì dữ liệu dump của RAM và SWAP 0 khá lớn nên không thể đưa vào báo cáo, xem thêm trong đường link

<https://drive.google.com/drive/folders/10cJX-Tdb6ehenUQV3--fZCoWLNvdGseE?usp=sharing>

- **Giải thích về output:**

- **alloc 384 0:** Cấp phát 384 byte cho `vm_region 0`. Thực hiện tăng `sbrk` lên 384 đơn vị (`sbrk = 384`). Vì 384 ô nhớ trải dài qua 2 page nên `vm_end` được tăng tương ứng (`vm_end = 512`). Chương trình chọn ra frame trống và ánh xạ với page cần cấp phát, bảng phân trang như sau:

- * PTE page 0: 0x80000000 (present = 1, frame = 0)

- * PTE page 1: 0x80000001 (present = 1, frame = 1)

Reference string: 1, 0

Khoảng địa chỉ được cấp phát sẽ là 0x000 - 0x17F, nằm trong frame 0 và 1.

- **alloc 128 1:** Cấp phát 128 byte cho `vm_region 1`. Tăng `sbrk` 128 đơn vị (`sbrk = 512`). Vì sau khi tăng `sbrk` chưa vượt quá `vm_end` nên không cần cấp phát thêm page. Khoảng địa chỉ được cấp phát trong RAM là 0x180 - 0x1FF, nằm trong frame 1.

- **alloc 64 2:** Cấp phát 64 byte cho `vm_region 2`. Tăng `sbrk` lên 64 đơn vị (`sbrk = 576`). Hiện tại `sbrk` mới đã vượt qua giới hạn, tràn qua page 2 nên cần cấp phát page đó (`vm_end = 768`). Chương trình chọn ra frame trống là 2 và ánh xạ với page 2. PTE của page 2 sẽ là: 0x80000002 (present = 1, frame = 2).

Reference string: 1, 0, 2

Khoảng địa chỉ được cấp phát sẽ là 0x200 - 0x23F, nằm trong frame 2.

- **alloc 320 3:** Cấp phát 320 byte cho `vm_region 3`. Tăng `sbrk` lên 320 đơn vị (`sbrk = 896`). Vì `sbrk` tràn qua page 3 nên tăng giới hạn tương ứng (`vm_end = 1024`). Frame trống được chọn là frame 3, ánh xạ với page 3. PTE của page 3 sẽ là 0x80000003 (present = 1, frame = 3)

Reference string: 1, 0, 2, 3

Khoảng địa chỉ được cấp phát trong RAM sẽ là 0x240 - 0x37F, nằm trong frame 2 và 3.

- **free 0:** Giải phóng vùng nhớ tại `vm_region 0`. Khoảng địa chỉ 0x000 - 0x17F của nó sẽ được giải phóng và sử dụng cho các lần cấp phát tiếp theo.
- **free 2:** Giải phóng vùng nhớ tại `vm_region 2`. Khoảng địa chỉ 0x200 - 0x23F của nó sẽ được giải phóng và sử dụng cho các lần cấp phát tiếp theo.
- **write 49 1 0:** Ghi dữ liệu 49 (ký tự "1") vào `vm_region 1`, offset = 0. Vì ô nhớ ở page 1 hiện có trong RAM nên việc ghi diễn ra bình thường.

- **write 49 1 127:** Ghi dữ liệu 49 (ký tự "1") vào `vm_region 1`, `offset = 127`. Vì ô nhớ ở page 1 hiện có trong RAM nên việc ghi diễn ra bình thường.
- **write 51 3 0:** Ghi dữ liệu 51 (ký tự "3") vào `vm_region 3`, `offset = 0`. Vì ô nhớ ở page 2 hiện có trong RAM nên việc ghi diễn ra bình thường.
- **write 51 3 319:** Ghi dữ liệu 51 (ký tự "3") vào `vm_region 3`, `offset = 127`. Vì ô nhớ ở page 3 hiện có trong RAM nên việc ghi diễn ra bình thường.
- **alloc 576 6:** Cấp phát 576 byte cho `vm_region 6`. Hiện tại ta đã có 2 frame trống nhưng kích thước nhỏ hơn nên không thể sử dụng. Tăng `sbrk` lên 576 đơn vị (`sbrk = 1472`). Dữ liệu được cấp phát hiện đã tràn ra 2 page mới, tăng `vm_end` tương ứng (`vm_end = 1536`). Vì RAM hiện tại không còn frame trống nên cần thực hiện thao tác swap:

- * Chọn victim page là 1 và 0 theo FIFO, swap-out vào frame 0 và 1 của SWAP 0 tương ứng.

PTE của page 1: 0xC0000000 (`swapped = 1`, `swap offset = 0`, `swap type = 0`)

PTE của page 0: 0xC0002000 (`swapped = 1`, `swap offset = 256`, `swap type = 0`)

- * Ánh xạ page 5 vào frame 0 của RAM, PTE: 0x80000000 (`present = 1`, `frame = 0`)
- * Ánh xạ page 4 vào frame 1 của RAM, PTE: 0x80000001 (`present = 1`, `frame = 1`)

Reference string: 2, 3, 5, 4

- **alloc 448 4:** Cấp phát 448 byte cho `vm_region 4`. Các `vm_region` trống vẫn không đủ nên lại phải tăng giới hạn 448 đơn vị (`sbrk = 1920`). Vì nó vượt quá giới hạn `vm_end` một khoảng trải dài 2 page nên `vm_end` được tăng tương ứng (`vm_end = 2048`). Thực hiện swap để lấy frame trống:

- * Chọn victim page là 2 và 3 theo FIFO, swap-out vào frame 2 và 3 của SWAP 0.

PTE của page 2: 0xC0004000 (`swapped = 1`, `swap offset = 512`, `swap type = 0`)

PTE của page 3: 0xC0006000 (`swapped = 1`, `swap offset = 768`, `swap type = 0`)

- * Ánh xạ page 7 vào frame 3 của RAM, PTE: 0x80000003 (`present = 1`, `frame = 3`)
- * Ánh xạ page 6 vào frame 2 của RAM, PTE: 0x80000002 (`present = 1`, `frame = 2`)

Reference string: 5 4 7 6

- **alloc 128 5:** Cấp phát 128 byte cho `vm_region 5`. Ta chọn được `vm_region` trống đủ chỗ để cấp phát - 384 byte của `vm_region 0` trước đó. Vì khoảng dữ liệu 128 byte ở đây hiện đang ở page 0 mà nó nằm trong SWAP nên cần swap lại:

- * Chọn victim page là 5 theo FIFO, nằm trong frame 0 của RAM

- * Swap-out page 5 vào frame 4 của SWAP 0, PTE: 0xC0008000 (swapped = 1, swap offset = 1024, swap type = 0)
- * Swap-in page 0 từ frame 1 của SWAP vào frame 0 của RAM, PTE: 0x80002000 (swapped = 0, frame = 0)

Reference string: 4 7 6 0

- **alloc 384 0**: Cấp phát 384 byte cho `vm_region` 0. Hiện tại không có `vm_region` trống để chứa nên cần phải tăng giới hạn (`sbrk = 2304`). Giới hạn mới tràn ra khỏi `vm_end` một page nên `vm_end` cũng tăng (`vm_end = 2304`). Thực hiện swap để lấy frame trống:

- * Chọn victim page là 4 theo FIFO (nằm trong frame 1 của RAM), swap-out nó vào frame 1 của SWAP 0.

PTE của page 4: 0xC0002000 (swapped = 1, swap offset = 256, swap type = 0)

- * Ánh xạ page 8 với frame 1 của RAM, PTE: 0x80000001 (present = 1, frame = 1)

Reference string: 7 6 0 8

- ... (các lệnh còn lại giải thích tương tự)

2.4 Trả lời câu hỏi

2.4.1 Phân đoạn bộ nhớ

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

Trả lời:

- Phân đoạn cung cấp mức độ linh hoạt vì các phân đoạn có kích thước thay đổi và các quy trình có thể được thiết kế để có nhiều phân đoạn cho phép cấp phát bộ nhớ chi tiết hơn.
- Hệ thống tối ưu hóa việc sử dụng và phân bổ bộ nhớ theo nhu cầu cụ thể của các thành phần và quy trình khác nhau.
- Phân đoạn cũng cho phép chia sẻ các phân đoạn bộ nhớ giữa các quy trình, rất hữu ích cho giao tiếp giữa các quy trình hoặc để chia sẻ thư viện mã.
- Tách riêng các phần khác nhau trong OS trong các phân đoạn giúp các phân đoạn bị lỗi thì toàn hệ thống không bị gặp lỗi.

Ví dụ:

- Stack segmentation: là 1 phần bộ nhớ trong kiến trúc bộ nhớ của máy tính được dành riêng để lưu trữ kết quả trả về hàm hoặc biến. Được quản lý bằng cơ chế "LIFO"

- Heap segmentation: lưu trữ dữ liệu động. Thường được cấp phát tĩnh hoặc động và nó được sử dụng để lưu trữ các cấu trúc dữ liệu có kích thước không xác định hoặc các đối tượng được cấp phát bộ nhớ trong khi chương trình đang thực hiện.
- Data segmentation: là quá trình chia nhỏ dữ liệu để tăng tốc độ truy cập và tìm kiếm giúp giảm thiểu tài nguyên và cải thiện hiệu suất của hệ thống.
- Code segmentation: là chia nhỏ chương trình thành các đoạn nhỏ hơn để dễ quản lý. Khi code được phân đoạn, chương trình có thể được nạp và thực thi một cách nhanh chóng.

2.4.2 Kết hợp phân đoạn bộ nhớ và phân trang bộ nhớ

Câu hỏi: What is the advantage and disadvantage of segmentation with paging?

Trả lời

- **Ưu điểm:**

- Segmentation with paging quản lý bộ nhớ hiệu quả, cung cấp bộ nhớ động. Chia bộ nhớ thành các trang nhỏ làm giảm bộ nhớ. Giảm sự phân mảnh và đơn giản hóa việc cấp phát.
- Segmentation with paging hỗ trợ tăng thêm số lượng địa chỉ bộ nhớ ảo lớn. Điều này làm tăng bộ nhớ ảo.
- Kết hợp được hai ưu điểm là bảo vệ bộ nhớ nhưng vẫn giữ được tính chia sẻ dữ liệu.

- **Nhược điểm:**

- Segmentation with paging là kỹ thuật bộ nhớ có độ phức tạp cao.
- Chưa giải quyết được vấn đề phân mảnh nội làm giảm hiệu suất của hệ thống.
- Làm tăng thêm chi phí hệ thống, mỗi lần truy cập bộ nhớ đòi hỏi thêm thời gian xử lý.
- Page table lưu trữ liên tục trên hệ thống.

Ví dụ: Một process được chia thành nhiều phân đoạn, mỗi phân đoạn này lại được chia làm nhiều trang có thước giống nhau. Khi process này chạy, OS sẽ tạo một bản đơn cho bộ nhớ ảo. Với mỗi phân đoạn thì OS sẽ cấp phát một vùng dữ liệu liên tục trên bộ nhớ ảo, rồi phân đoạn này sẽ chia thành các trang có kích thước giống nhau. Khi có yêu cầu truy cập vào 1 phân đoạn nào đó, OS sẽ sử dụng Page Table để dịch địa chỉ ảo sang địa chỉ vật lý tương ứng trong bộ nhớ RAM. Sau khi dịch, OS kiểm tra trang tương ứng đã được tải lên bộ nhớ RAM chưa, nếu chưa thì sẽ tải từ đĩa vào bộ nhớ RAM. Việc này khá tốn tài nguyên cho việc quản lý Page Table và có thể mang lại hiện tượng phân mảnh vì một số trang có thể bị đặt trong các vùng ảo không liên tục. Nhưng cách này quản lý bộ nhớ hiệu quả, làm giảm xung đột bộ nhớ.

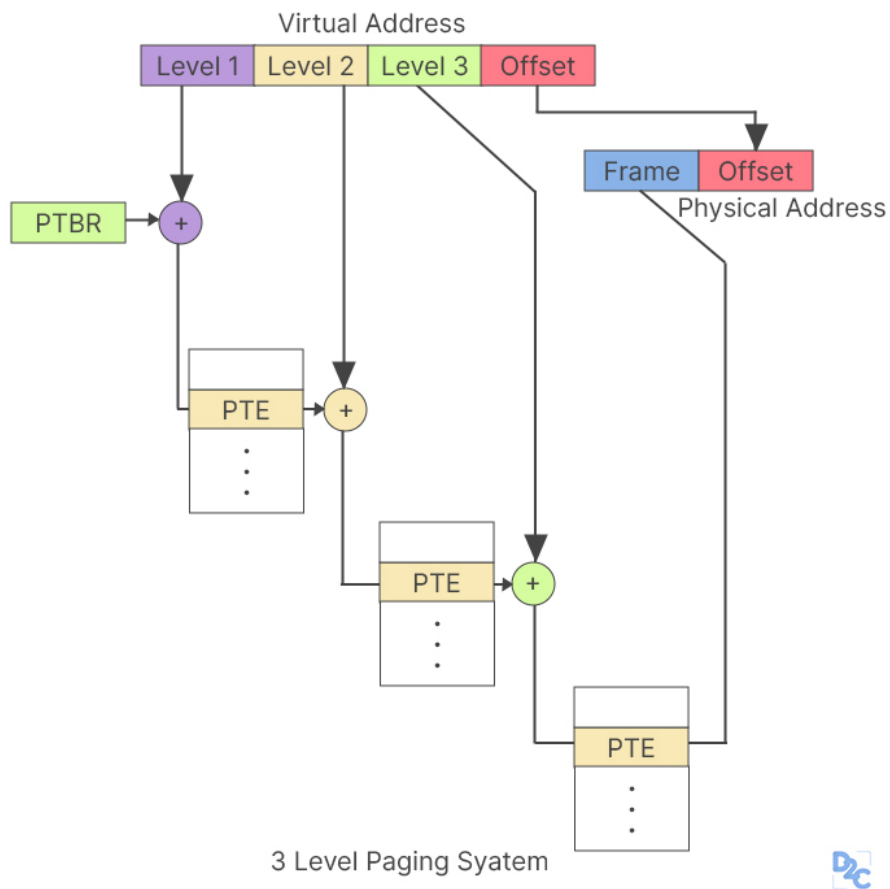
2.4.3 Phân trang đa cấp

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

Trả lời:

- Việc chia địa chỉ thành nhiều hơn 2 cấp sẽ tạo ra sơ đồ phân trang nhiều cấp.
- Page table được chia thành nhiều cấp độ.
- Địa chỉ cơ sở của mục nhập trang cấp một sẽ là địa chỉ cơ sở của mục nhập bảng trang cấp hai và mục nhập bảng trang cấp hai sẽ là địa chỉ cơ sở của mục nhập bảng trang cấp ba, v.v.
- Thông tin trang thực tế sẽ được lưu trữ trong các mục của Page Table cấp cuối cùng
- Ưu điểm:
 - Giảm chi phí bộ nhớ của Page Table bằng cách chỉ cấp phát bộ nhớ cho các Page Table thực sự được sử dụng.
 - Chia bộ nhớ vật lý thành các khung có kích thước cố định. Có thể chia các trang thành các đơn vị nhỏ hơn. Làm giảm lãng phí không gian và phân mảnh nội.
- Nhược điểm:
 - Tăng thời gian dịch địa chỉ dịch bộ nhớ ảo sang địa chỉ vật lý vì phải truy cập nhiều mức của Page Table.
 - Tăng độ phức tạp của hệ thống.

Ví dụ: Chia địa chỉ thành 3 cấp trong hệ thống quản lý bộ nhớ phân trang



3 Kết hợp bộ định thời và quản lý bộ nhớ (Put It All Together)

3.1 Kết quả kiểm thử

3.1.1 Test case 1

- **Input:** Đường dẫn `input/os_1_singleCPU_mfq_paging`

```
input > os_1_singleCPU_mlq_paging
1 2 1 8
2 1048576 16777216 0 0 0
3 1 s4 4
4 2 s3 3
5 4 m1s 2
6 6 s2 3
7 7 m0s 3
8 9 p1s 2
9 11 s0 1
10 16 s1 0
11 |
```

Hình 20: Input - os_1_singleCPU_mlq_paging

• Output

```
phuong@admin:~/OSAssignment/ossim_test$ ./os os_1_singleCPU_mlq_paging
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
    Loaded a process at input/proc/m1s, PID: 3 PRI0: 2
    CPU 0: Put process 1 to run queue
    CPU 0: Dispatched process 1
Time slot 5
    Loaded a process at input/proc/s2, PID: 4 PRI0: 3
Time slot 6
    CPU 0: Put process 1 to run queue
    CPU 0: Dispatched process 1
Time slot 7
    Loaded a process at input/proc/m0s, PID: 5 PRI0: 3
```

```
Time slot 8
    CPU 0: Put process 1 to run queue
    CPU 0: Dispatched process 1
    Loaded a process at input/proc/p1s, PID: 6 PRI0: 2
Time slot 9
    CPU 0: Processed 1 has finished
    CPU 0: Dispatched process 3
Time slot 10
Time slot 11
    Loaded a process at input/proc/s0, PID: 7 PRI0: 1
    CPU 0: Put process 3 to run queue
    CPU 0: Dispatched process 6
Time slot 12
Time slot 13
    CPU 0: Put process 6 to run queue
    CPU 0: Dispatched process 3
Time slot 14
Time slot 15
    CPU 0: Put process 3 to run queue
    CPU 0: Dispatched process 6
    Loaded a process at input/proc/s1, PID: 8 PRI0: 0
Time slot 16
Time slot 17
    CPU 0: Put process 6 to run queue
    CPU 0: Dispatched process 3
Time slot 18
Time slot 19
    CPU 0: Processed 3 has finished
    CPU 0: Dispatched process 6
Time slot 20
Time slot 21
    CPU 0: Put process 6 to run queue
    CPU 0: Dispatched process 6
Time slot 22
Time slot 23
    CPU 0: Put process 6 to run queue
    CPU 0: Dispatched process 6
Time slot 24
Time slot 25
```



```
CPU 0: Processed 6 has finished
CPU 0: Dispatched process 2
Time slot 26
Time slot 27
CPU 0: Put process 2 to run queue
CPU 0: Dispatched process 4
Time slot 28
Time slot 29
CPU 0: Put process 4 to run queue
CPU 0: Dispatched process 5
Time slot 30
Time slot 31
CPU 0: Put process 5 to run queue
CPU 0: Dispatched process 2
Time slot 32
Time slot 33
CPU 0: Put process 2 to run queue
CPU 0: Dispatched process 4
Time slot 34
Time slot 35
CPU 0: Put process 4 to run queue
CPU 0: Dispatched process 5
Time slot 36
Time slot 37
CPU 0: Put process 5 to run queue
CPU 0: Dispatched process 2
Time slot 38
Time slot 39
CPU 0: Put process 2 to run queue
CPU 0: Dispatched process 4
Time slot 40
Time slot 41
CPU 0: Put process 4 to run queue
CPU 0: Dispatched process 5
write region=1 offset=20 value=102
print_pgtbl: 0 - 512
00000000: 80000002
00000004: 80000003
Time slot 42
```

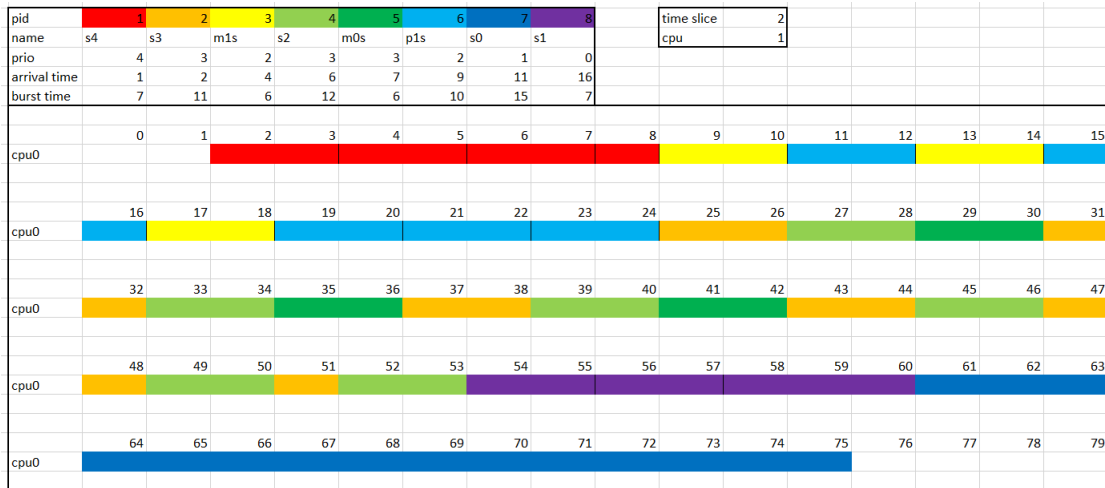
```
write region=2 offset=1000 value=1
print_pgtbl: 0 - 512
00000000: 80000002
00000004: 80000003
Time slot 43
    CPU 0: Processed 5 has finished
    CPU 0: Dispatched process 2
Time slot 44
Time slot 45
    CPU 0: Put process 2 to run queue
    CPU 0: Dispatched process 4
Time slot 46
Time slot 47
    CPU 0: Put process 4 to run queue
    CPU 0: Dispatched process 2
Time slot 48
Time slot 49
    CPU 0: Put process 2 to run queue
    CPU 0: Dispatched process 4
Time slot 50
Time slot 51
    CPU 0: Put process 4 to run queue
    CPU 0: Dispatched process 2
Time slot 52
    CPU 0: Processed 2 has finished
    CPU 0: Dispatched process 4
Time slot 53
Time slot 54
    CPU 0: Processed 4 has finished
    CPU 0: Dispatched process 8
Time slot 55
Time slot 56
    CPU 0: Put process 8 to run queue
    CPU 0: Dispatched process 8
Time slot 57
Time slot 58
    CPU 0: Put process 8 to run queue
    CPU 0: Dispatched process 8
Time slot 59
```




```
Time slot 60
    CPU 0: Put process 8 to run queue
    CPU 0: Dispatched process 8
Time slot 61
    CPU 0: Processed 8 has finished
    CPU 0: Dispatched process 7
Time slot 62
Time slot 63
    CPU 0: Put process 7 to run queue
    CPU 0: Dispatched process 7
Time slot 64
Time slot 65
    CPU 0: Put process 7 to run queue
    CPU 0: Dispatched process 7
Time slot 66
Time slot 67
    CPU 0: Put process 7 to run queue
    CPU 0: Dispatched process 7
Time slot 68
Time slot 69
    CPU 0: Put process 7 to run queue
    CPU 0: Dispatched process 7
Time slot 70
Time slot 71
    CPU 0: Put process 7 to run queue
    CPU 0: Dispatched process 7
Time slot 72
Time slot 73
    CPU 0: Put process 7 to run queue
    CPU 0: Dispatched process 7
Time slot 74
Time slot 75
    CPU 0: Put process 7 to run queue
    CPU 0: Dispatched process 7
Time slot 76
    CPU 0: Processed 7 has finished
    CPU 0 stopped
phuong@admin:~/OSAssignment/ossim_te
```

```
phuong@admin:~/OSAssignment/ossim_test$
```

- Biểu đồ Gantt



Hình 21: Biểu đồ Gantt cho test case `os_1_singleCPU_mfq_paging`

- Bộ nhớ RAM/SWAP0:

Xem thêm trong đường dẫn <https://drive.google.com/drive/folders/10cJX-Tdb6ehenUQV3--fZCoWLNvDgusp=sharing>

3.1.2 Test case 2

Input: Đường dẫn input/os_0_m1q_paging

```
input > os_0_mfq_paging
1      6 2 4
2    1048576 16777216 0 0 0
3      0 p0s 0
4      2 p1s 15
5      3 p1s 0
6      4 p1s 0
7
```

Hình 22: Input - os_0_m1q_paging

Output:



```
phuong@admin:~/OSAssignment/ossim_test$ ./os os_0_mlq_paging
Time slot 0
ld_routine
    Loaded a process at input/proc/p0s, PID: 1 PRI0: 0
    CPU 1: Dispatched process 1
Time slot 1
    Loaded a process at input/proc/p1s, PID: 2 PRI0: 15
print_pgtbl: 0 - 512
00000000: 80000000
00000004: 80000001
Time slot 2
    CPU 0: Dispatched process 2
Time slot 3
    Loaded a process at input/proc/p1s, PID: 3 PRI0: 0
print_pgtbl: 0 - 768
00000000: 80000000
00000004: 80000001
00000008: 80000002
Time slot 4
print_pgtbl: 0 - 768
00000000: 80000000
00000004: 80000001
00000008: 80000002
    Loaded a process at input/proc/p1s, PID: 4 PRI0: 0
Time slot 5
print_pgtbl: 0 - 768
00000000: 80000000
00000004: 80000001
00000008: 80000002
write_region=1 offset=20 value=100
print_pgtbl: 0 - 768
00000000: 80000000
00000004: 80000001
00000008: 80000002
Time slot 6
    CPU 1: Put process 1 to run queue
    CPU 1: Dispatched process 3
Time slot 7
```



```
Time slot 8
    CPU 0: Put process 2 to run queue
    CPU 0: Dispatched process 4
Time slot 9
Time slot 10
Time slot 11
Time slot 12
    CPU 1: Put process 3 to run queue
    CPU 1: Dispatched process 1
read region=1 offset=20 value=100
print_pgtbl: 0 - 768
00000000: 80000000
00000004: 80000001
00000008: 80000002
Time slot 13
write region=2 offset=20 value=102
print_pgtbl: 0 - 768
00000000: 80000000
00000004: 80000001
00000008: 80000002
Time slot 14
    CPU 0: Put process 4 to run queue
    CPU 0: Dispatched process 3
Time slot 15
read region=2 offset=20 value=11
print_pgtbl: 0 - 768
00000000: 80000000
00000004: 80000001
00000008: 80000002
write region=3 offset=20 value=103
print_pgtbl: 0 - 768
00000000: 80000000
00000004: 80000001
00000008: 80000002
Time slot 16
Time slot 17
    CPU 1: Processed 1 has finished
    CPU 1: Dispatched process 4
Time slot 18
```

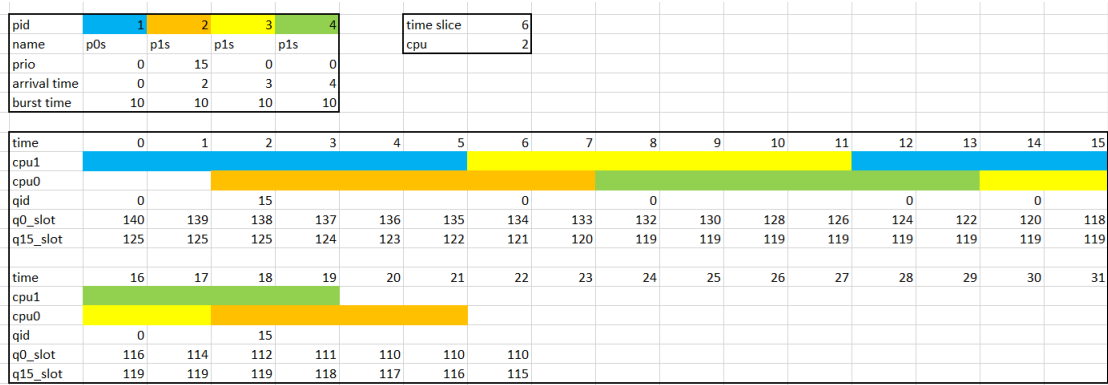


```

CPU 0: Processed 3 has finished
CPU 0: Dispatched process 2
Time slot 19
Time slot 20
CPU 1: Processed 4 has finished
CPU 1 stopped
Time slot 21
Time slot 22
CPU 0: Processed 2 has finished
CPU 0 stopped
phuong@admin:~/OSAssignment/ossim_test$

```

Biểu đồ Gantt:



Hình 23: Biểu đồ Gantt cho test case os_0_mlq_paging

Bộ nhớ RAM/SWAP0:

Xem thêm trong đường dẫn <https://drive.google.com/drive/folders/10cJX-Tdb6ehenUQV3--fZCoWLNvGseusp=sharing>

3.2 Trả lời câu hỏi

3.2.1 Hậu quả của việc không đồng bộ hóa

Câu hỏi: 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 đồng bộ hóa không được xử lý, nó có thể dẫn đến các vấn đề Race Condition. Race Condition là một lỗi khi hai hay nhiều luồng đồng thời truy cập, đọc, ghi cùng dữ liệu hay tài nguyên.

Ví dụ: Trong OS có 2 process là A và B cùng với tài nguyên X. Trong khi process A đang muốn truy cập vào X thì đồng thời process B muốn thay đổi giá trị X. Nếu đồng bộ hóa không được xử lý thì kết quả thu giá trị không chính xác hoặc lỗi đến từ tài nguyên X.

Cách khắc phục: Sử dụng Mutex.

- Khởi tạo một biến mutex, đảm bảo rằng sẽ sử dụng cùng một mutex cho tất cả các thread muốn đồng bộ hóa.
- Khi một thread truy cập vào dữ liệu nào đó, nó sẽ khóa mutex bằng cách sử dụng hàm ‘

```
pthread_mutex_lock(&mutex);
```

- Sau khi thread đã hoàn thành thao tác sẽ giải phóng mutex bằng cách sử dụng hàm

```
pthread_mutex_unlock(&mutex);
```

- Tất cả các thread khác sẽ phải chờ đợi mutex được giải phóng trước khi có thể truy cập vào dữ liệu được bảo vệ bởi mutex.

Ví dụ: Giả sử với hàm `put_mlq_proc()` và `enqueue()` như sau:

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

// queue.c
void enqueue(struct queue_t * q, struct pcb_t * proc) {
    /* TODO: put a new process to queue [q] */
    if (q->size == MAX_QUEUE_SIZE) // queue full
    {
        #ifdef SCHED_DEBUG
        printf("Queue is full\n");
        #endif
        abort();
    }
    if (proc == NULL)
    {
```

```
        #ifdef SCHED_DEBUG
        printf("Enqueue a NULL process!\n");
        #endif
    }
    q->proc[q->size] = proc;
    q->size++;
}
```

Giả sử ta không sử dụng cơ chế đồng bộ mutex lock (`queue_lock`). Vì chương trình có thể sử dụng nhiều CPU nên chúng có thể gọi hàm `put_mfq_proc()` cùng một lúc cũng như hàm `enqueue()`. Tuy nhiên, `queue` ở đây (tham số `q`) chính là một con trỏ tới shared resource chính là một trong các `mq_ready_queue`. Nếu như 2 CPU cùng gọi hàm `enqueue()` 1 lúc, rất có thể sẽ dẫn tới thứ tự thực thi như sau:

```
...
q->proc[q->size] = proc; // CPU 1
q->proc[q->size] = proc; // CPU 2
...
```

Khi đó, process được gán vào ở đây lại là process của CPU 2, còn process của CPU 1 được gán trước đó sẽ bị ghi đè lên dẫn đến tình trạng thiếu hụt process.

Ngoài ra còn có trường hợp ảnh hưởng tới biến `q->size` như sau:

```
...
reg1 = q->size // CPU 1
reg2 = q->size // CPU 2
reg1 = reg1 + 1
reg2 = reg2 + 1
q->size = reg1
q->size = reg2
...
```

Nếu xét đến khía cạnh ngôn ngữ cấp thấp (hợp ngữ) của chương trình thì vấn đề sẽ xảy ra là giá trị `q->size` chỉ tăng 1 theo thứ tự thực thi trên.

Do đó, việc sử dụng các công cụ đồng bộ trong môi trường thực thi song song như vậy là rất cần thiết.



Tài liệu tham khảo

[1] Kernel Documentation

<https://www.kernel.org/doc/Documentation>

[2] Swap Management

<https://www.kernel.org/doc/gorman/html/understand/understand014.html#toc74>