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

BÁO CÁO BÀI TẬP LỚN

MÔ PHỎNG HỆ ĐIỀU HÀNH ĐƠN GIẢN

MÔN HỌC: HỆ ĐIỀU HÀNH

HƯỚNG DẪN: LÊ THANH VÂN
VÕ ĐĂNG KHOA
NGUYỄN THANH QUÂN

DANH SÁCH NHÓM:

STT	Họ và tên	MSSV	Lớp LT_TN
1	Nguyễn Phương Duy	2110091	L04_L07
2	Lê Anh Khoa	2110271	L04_L07
3	Nguyễn Văn Ngọc Quang	2114511	L04_L07

Tp. Hồ Chí Minh, Ngày 19/05/2023

Phân chia khối lượng công việc

STT	Họ và tên	MSSV	Công việc được giao	Hoàn thành
1	Nguyễn Phương Duy	2110091	Hiện thực quản lý bộ nhớ và định thời Trình bày và xây dựng báo cáo qua Latex	100%
2	Lê Anh Khoa	2110271	Hiện thực quản lý bộ nhớ Thiết kế slide và xây dựng nội dung báo cáo	100%
3	Nguyễn Văn Ngọc Quang	2114511	Hiện thực chính sách định thời Thiết kế slide và trình bày báo cáo	100%

Mục lục

1 Chính sách định thời

1.1 Tổng quan

Chính sách định thời mà hệ điều hành được trang bị là Multi Level Queue (MLQ) kết hợp Round Robin. Trong hệ thống, mỗi process có một giá trị prio nhất định thể hiện độ ưu tiên của một process và được dùng để xác định ready_queue nào sẽ được dùng để chứa process đó, prio càng thấp thể hiện cho độ ưu tiên càng cao.

Ban đầu các hàng đợi có độ ưu tiên khác nhau sẽ có số lần sử dụng CPU khác nhau (thể hiện qua giá trị slot, tính bằng công thức MAX_PRIO - prio), hàng đợi có độ ưu tiên càng cao thì càng có nhiều lượt sử dụng CPU.

Hàng đợi có độ ưu tiên cao nhất sẽ được chọn để thi thực trước cho đến khi đã sử dụng hết số slot cho phép. Lúc này, CPU sẽ chuyển sang thực thi các process trong hàng đợi có độ ưu tiên thấp hơn. Tất cả hàng đợi sẽ được áp dụng cơ chế Round Robin, tức là việc thực thi một hàng đợi chính xác là thay phiên nhau thực thi các process trong hàng đợi ấy theo time slice cho trước, một lần thay phiên được tính là một lần sử dụng CPU.

Đến một thời điểm mà tất cả các hàng đợi đều sử dụng hết số slot ban đầu thì hệ thống sẽ cấp phát lại số slot cho từng hàng đợi theo công thức ban đầu sau đó lại tiếp thực hiện công việc định thời.

1.2 Hiện thực

Đầu tiên, ta định nghĩa cấu trúc của một hàng đợi gồm ba thuộc tính lần lượt là mảng lưu trữ các process, số process hiện tại và số lần sử dụng CPU của hàng đợi, cụ thể như sau:

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

Trong file sched.c ngoài việc sử dụng file header cho trước là "queue.h" nhóm bổ sung thêm file header "sched.h", hai hàm dequeue và enqueu được hiện thực như sau:

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

struct pcb_t * dequeue(struct queue_t * q) {
  #ifdef MLQ_SCHED
  // Return process at index = 0;
  if (empty(q)) return NULL;
  struct pcb_t * proc = q->proc[0];
  for (int i = 0; i < q->size - 1; i++) {
    q->proc[i] = q->proc[i + 1];
  }
  q->size--;
```

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

- Hàm enqueue đơn giản là thực hiện đặt một process mới vào cuối hàng đợi.
- Còn hàm dequeue thì nhóm hiện thực theo hai trường hợp:
  - Thứ nhất, nếu có sử dụng module MLQ_SCHED, dequeue sẽ lấy ra process đầu tiên của hàng đợi đồng thời trừ đi 1 lần sử dụng CPU của hàng đợi tương ứng. Việc lấy ra một process ở đầu và đặt vào một process ở cuối là do hàng đợi áp dụng Round Robin.
  - Thứ hai, nếu không sử dụng module MLQ_SCHED, hệ thống sẽ được áp dụng chính sách định thời một hàng đợi ưu tiên, vậy nên dequeue sẽ trả về process có độ ưu tiên cao nhất (dựa theo thuộc tính priority lưu trong pcb_t).

Đối với file "sched.c", ta chỉnh sửa hàm init_scheduler() và hiện thực hàm get_mlq_proc(), cụ thể như sau:

```
#ifdef MLQ_SCHED
static struct queue_t mlq_ready_queue[MAX_PRIO];
#endif

void init_scheduler(void) {
#ifdef MLQ_SCHED
    int i ;

    for (i = 0; i < MAX_PRIO; i ++) {
    mlq_ready_queue[i].size = 0;
    mlq_ready_queue[i].slot = MAX_PRIO - i;
    }
#endif
```

```
  ready_queue.size = 0;
  run_queue.size = 0;
  pthread_mutex_init(&queue_lock, NULL);
}

/*
 *  Stateful design for routine calling
 *  based on the priority and our MLQ policy
 *  We implement stateful here using transition technique
 *  State representation   prio = 0 .. MAX_PRIO, curr_slot = 0..(MAX_PRIO - prio)
 */
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 iterator;
  int num_empty_queue = 0;
  for (iterator = 0; iterator < MAX_PRIO; iterator++) {
    if (!empty(&mlq_ready_queue[iterator])) {
      if (mlq_ready_queue[iterator].slot > 0) {
        proc = dequeue(&mlq_ready_queue[iterator]);
        break;
      }
    }
    else num_empty_queue++;
  }
  if (num_empty_queue == MAX_PRIO) goto rt_proc;
  if (iterator == MAX_PRIO) {
    for (int i = 0; i < MAX_PRIO; i++) {
      mlq_ready_queue[i].slot = MAX_PRIO - i;
    }
    iterator = 0;
    while (iterator < MAX_PRIO) {
      if (!empty(&mlq_ready_queue[iterator])) {
        proc = dequeue(&mlq_ready_queue[iterator]);
        break;
      }
      iterator++;
    }
  }
  rt_proc:
  pthread_mutex_unlock(&queue_lock);
  return proc;
}
```

- Trong trường hợp của MLQ, hàm init_scheduler() sẽ thực hiện thêm việc khởi tạo kích thước và số slot cho tất cả hàng đợi theo công thức cho trước.
- Hàm get_mlq_proc() thực hiện việc lấy ra một process từ hàng đợi mlq_ready_queue. Đầu tiên, một con trỏ lần lượt duyệt qua các hàng đợi, bắt đầu từ hàng ưu tiên cao nhất (prio = 0) nếu tìm thấy hàng đợi khả thi (không rỗng và còn số lần sử dụng CPU) thì ta dequeue một process tại đó và kết thúc hàm.

- Nếu đã duyệt qua hết tất cả các hàng mà vẫn chưa thu được process, ta xét hai trường hợp:
  - Nếu tất cả các hàng đều rỗng, ta kết thúc hàm bằng cách trả về NULL.
  - Ngược lại, tức là tất cả hàng đợi đều có số slot = 0 thì ta khởi tạo lại số slot và thực hiện lại quá trình duyệt hàng và dequeue process.

1.3 Trả lời các câu hỏi

Câu hỏi: Ưu điểm của việc sử dụng hàng đợi ưu tiên so với các thuật toán lập lịch khác mà bạn đã học là gì?

Trả lời:

Hàng đợi ưu tiên có một số ưu điểm chính sau đây:

- Ưu tiên các công việc quan trọng: Hàng đợi ưu tiên cho phép xác định mức độ ưu tiên của các công việc. Các công việc quan trọng hơn có thể được ưu tiên xử lý trước, giúp đảm bảo tính cấp thiết của các tác vụ quan trọng.
- Đáp ứng nhanh các yêu cầu ưu tiên: Các công việc có mức độ ưu tiên cao được đặt vào đầu hàng đợi và được xử lý ngay sau khi các công việc ưu tiên thấp hơn đã hoàn thành. Điều này giúp giảm thời gian chờ đợi và đảm bảo tính nhanh chóng trong việc đáp ứng các yêu cầu ưu tiên.
- Định thời ưu tiên linh hoạt: Hàng đợi ưu tiên cho phép cung cấp các mức độ ưu tiên khác nhau cho các công việc. Điều này cho phép linh hoạt trong việc thiết lập độ ưu tiên dựa trên yêu cầu cụ thể và điều chỉnh mức độ ưu tiên theo thời gian.

Tuy nhiên, giải thuật này cũng tồn tại vấn đề là ưu tiên quá mức đối với process ưu tiên cao dẫn đến một hoặc một vài process ưu tiên thấp hơn có thể không được thực thi trong thời gian dài. Đây là hiện tượng Starvation, để giải quyết vấn đề này ta có thể dùng đến kỹ thuật Aging đó là tăng dần độ ưu tiên của các process chưa được thực thi trong ready queue.

1.4 Phân tích kết quả kiểm thử

1.4.1 Test case 1

Test case thứ nhất được thể hiện trong tệp "os_mlq_01" có nội dung như sau:

```
2 1 6
0 s0 134
4 s1 132
6 s2 134
7 s3 136
11 m0 128
12 p1 136
```

Hình 1: Test case 1

Trong test case này, hệ điều hành sử dụng 1 CPU áp dụng Round Robin với time slice = 2 để định thời sáu process. Thông tin tổng quan của sáu process như sau:

PID	proc	số chu kỳ thực thi	prio	thời điểm đến
1	s0	15	134	0
2	s1	7	132	4
3	s2	12	134	6
4	s3	11	136	7
5	m0	7	128	11
6	p1	10	136	12

*** Kết quả:**

```
phuongduy@phuongduy-VirtualBox:~/sched/ossim_source_code_part1_hk231$ ./os os_mlq_01
Time slot   0
        Loaded a process at input/proc/s0, PID: 1 PRIO: 134
Time slot   1
        CPU 0: Dispatched process  1
Time slot   2
Time slot   3
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   4
        Loaded a process at input/proc/s1, PID: 2 PRIO: 132
Time slot   5
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
Time slot   6
        Loaded a process at input/proc/s2, PID: 3 PRIO: 134
Time slot   7
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
        Loaded a process at input/proc/s3, PID: 4 PRIO: 136
Time slot   8
Time slot   9
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot  10
Time slot  11
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
        Loaded a process at input/proc/m0, PID: 5 PRIO: 128
Time slot  12
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  5
        Loaded a process at input/proc/p1, PID: 6 PRIO: 136
Time slot  13
Time slot  14
        CPU 0: Put process  5 to run queue
        CPU 0: Dispatched process  5
Time slot  15
Time slot  16
        CPU 0: Put process  5 to run queue
        CPU 0: Dispatched process  5
Time slot  17
Time slot  18
        CPU 0: Put process  5 to run queue
        CPU 0: Dispatched process  5
Time slot  19
        CPU 0: Processed  5 has finished
        CPU 0: Dispatched process  1
Time slot  20
```

Hình 2: Kết quả test case 1 (phần 1)

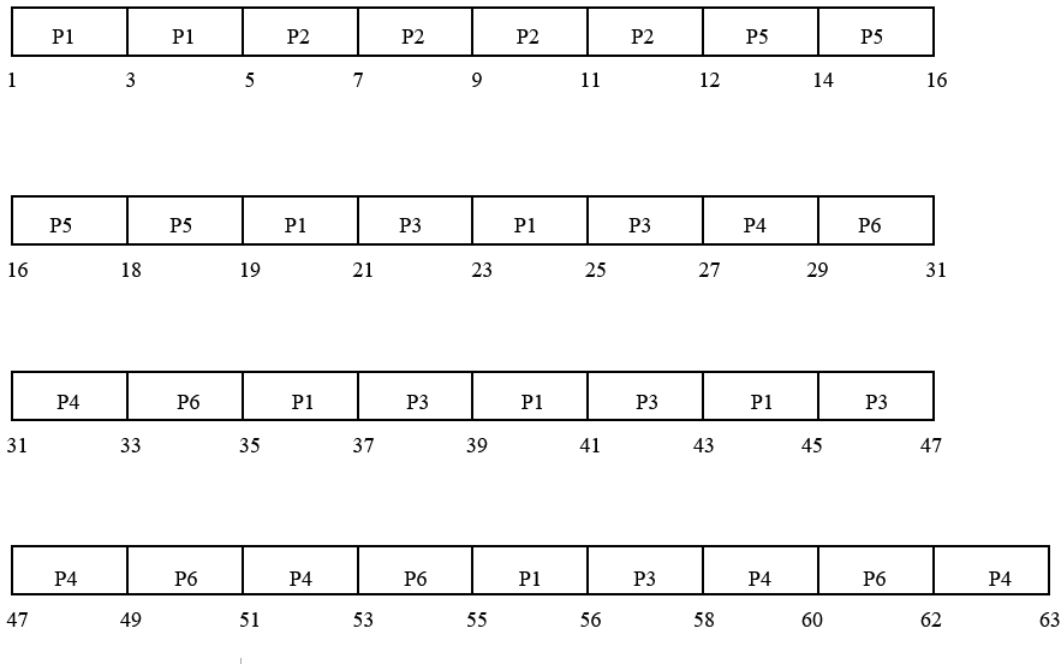
```
Time slot  21
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  3
Time slot  22
Time slot  23
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  1
Time slot  24
Time slot  25
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  3
Time slot  26
Time slot  27
        CPU 0: Put process  3 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  6
Time slot  30
Time slot  31
        CPU 0: Put process  6 to run queue
        CPU 0: Dispatched process  4
Time slot  32
Time slot  33
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  6
Time slot  34
Time slot  35
        CPU 0: Put process  6 to run queue
        CPU 0: Dispatched process  1
Time slot  36
Time slot  37
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  3
Time slot  38
Time slot  39
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  1
Time slot  40
Time slot  41
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  3
Time slot  42
Time slot  43
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  1
```

Hình 3: Kết quả test case 1 (phần 2)

```
Time slot  44
Time slot  45
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  3
Time slot  46
Time slot  47
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
Time slot  48
Time slot  49
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  6
Time slot  50
Time slot  51
        CPU 0: Put process  6 to run queue
        CPU 0: Dispatched process  4
Time slot  52
Time slot  53
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  6
Time slot  54
Time slot  55
        CPU 0: Put process  6 to run queue
        CPU 0: Dispatched process  1
Time slot  56
        CPU 0: Processed  1 has finished
        CPU 0: Dispatched process  3
Time slot  57
Time slot  58
        CPU 0: Processed  3 has finished
        CPU 0: Dispatched process  4
Time slot  59
Time slot  60
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  6
Time slot  61
Time slot  62
        CPU 0: Processed  6 has finished
        CPU 0: Dispatched process  4
Time slot  63
        CPU 0: Processed  4 has finished
        CPU 0 stopped
phuongduy@phuongduy-VirtualBox:~/sched/ossim_source_code_part1_hk231$
```

Hình 4: Kết quả test case 1 (phần 3)

*** Gantt diagram:**



Hình 5: Gantt diagram cho test case 1

Ban đầu P1 được trao quyền thực thi cho đến thời điểm 5, lúc này ngoài P1 mlq_ready_ queue còn có P2, process này có độ ưu tiên cao hơn P1 nên quyền sử dụng CPU được trao lại cho P2 cho đến thời điểm 12 (do P2 đã hoàn thành).

Khoảng thời gian từ 5 tới 12 mặc dù có nhiều process đang đợi trong mlq_ready_ queue nhưng P2 vẫn được thực thi đó là do P2 thuộc hàng đợi có ưu tiên cao nhất.

Bắt đầu thời điểm 12, process có độ ưu tiên cao nhất là P5 được ưu tiên đến khi hoàn thành ở thời điểm 19. Lúc này trong mlq_ready_queue chỉ còn hai hàng đợi không rỗng đó là hai hàng đợi có độ ưu tiên lần lượt 134 và 136. Tiếp tục công việc, hàng đợi được chọn thực thi là hàng đợi ưu tiên thứ 134, do hàng này đang chứa P1 và P3 nên hai process này được thực thi xen kẽ nhau bắt đầu từ P1.

Nhận thấy rằng, tại thời điểm 27 quyền sử dụng CPU bị thu hồi và trao lại cho P4 (có độ ưu tiên nhỏ hơn P1 và P3) mặc dù P1, P3 chưa được hoàn thành. Lý do là hàng đợi thứ 134 đã hết số lần sử dụng CPU cho phép của một vòng (6 lần cụ thể là: 1-3, 3-5, 19-21, 21-23, 23-25, 25-27) vì vậy quyền sử dụng CPU lúc này thuộc về hàng đợi thứ 136 (chứa P4 và P6).

Từ thời điểm 27 tới 35, trải qua 4 lần sử dụng CPU của hàng đợi 136. Tại đây tất cả hàng đợi có chứa process không còn lượt sử dụng nữa nên hệ thống sẽ cấp phát lại số lần sử dụng như ban đầu để tiếp tục công việc và rồi quyền ưu tiên lại quay về hàng đợi thứ 134 ở thời điểm 36. Công việc cứ thế được thực hiện cho đến khi tất cả process hoàn thành.

1.4.2 Test case 2

Với tệp đầu vào "os_mlq_2" như bên dưới, hệ điều hành được trang bị 2 CPU (Round Robin với time slice = 2) để định thời 8 process.

```
2 2 8
1 s4  4
2 s3  3
4 m1  2
6 s2  3
7 m0  3
9 p1  2
11 s0 1
16 s1 0
```

Hình 6: Test case 2

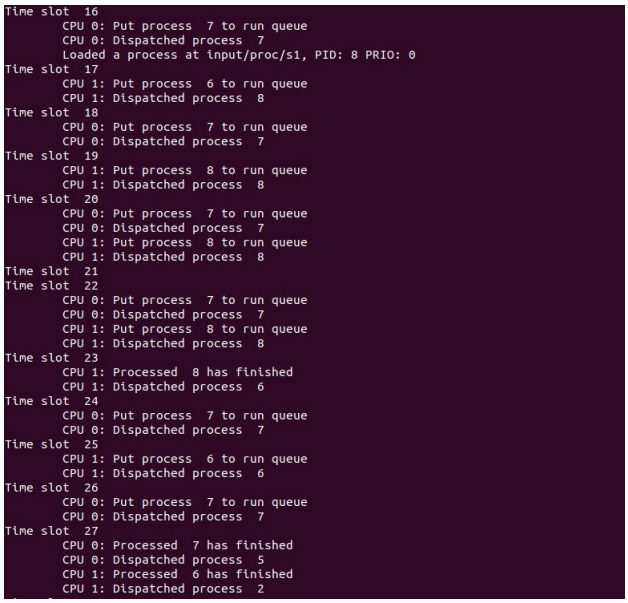
Thông tin của các process liên quan như sau:

PID	proc	số chu kỳ thực thi	prio	thời điểm đến
1	s4	7	4	1
2	s3	11	3	2
3	m1	8	2	4
4	s2	12	3	6
5	m0	7	3	7
6	p1	10	2	9
7	s0	15	1	11
8	s1	7	0	16

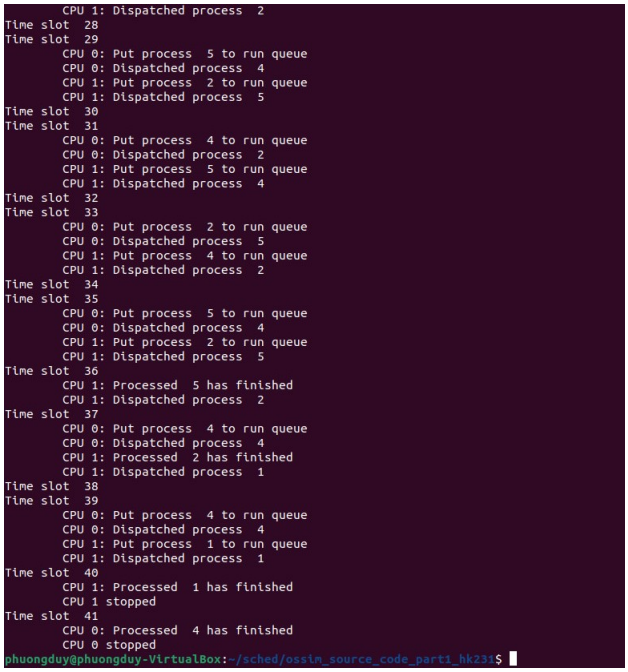
*** Kết quả:**

```
phuongduy@phuongduy-VirtualBox:~/sched/os$sln_source_code_part1_hk231$ ./os os_mlq_2
Time slot   0
Time slot   1
        Loaded a process at input/proc/s4, PID: 1 PRIO: 4
Time slot   2
        CPU 0: Dispatched process  1
        Loaded a process at input/proc/s3, PID: 2 PRIO: 3
Time slot   3
        CPU 1: Dispatched process  2
Time slot   4
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
        Loaded a process at input/proc/m1, PID: 3 PRIO: 2
Time slot   5
        CPU 1: Put process  2 to run queue
        CPU 1: Dispatched process  3
Time slot   6
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
        Loaded a process at input/proc/s2, PID: 4 PRIO: 3
Time slot   7
        CPU 1: Put process  3 to run queue
        CPU 1: Dispatched process  3
        Loaded a process at input/proc/m0, PID: 5 PRIO: 3
Time slot   8
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  4
Time slot   9
        CPU 1: Put process  3 to run queue
        CPU 1: Dispatched process  3
        Loaded a process at input/proc/p1, PID: 6 PRIO: 2
Time slot  10
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  6
Time slot  11
        CPU 1: Put process  3 to run queue
        CPU 1: Dispatched process  3
        Loaded a process at input/proc/s0, PID: 7 PRIO: 1
Time slot  12
        CPU 0: Put process  6 to run queue
        CPU 0: Dispatched process  7
Time slot  13
        CPU 1: Processed  3 has finished
        CPU 1: Dispatched process  6
Time slot  14
        CPU 0: Put process  7 to run queue
        CPU 0: Dispatched process  7
Time slot  15
        CPU 1: Put process  6 to run queue
        CPU 1: Dispatched process  6
```

Hình 7: Kết quả test case 2 (phần 1)

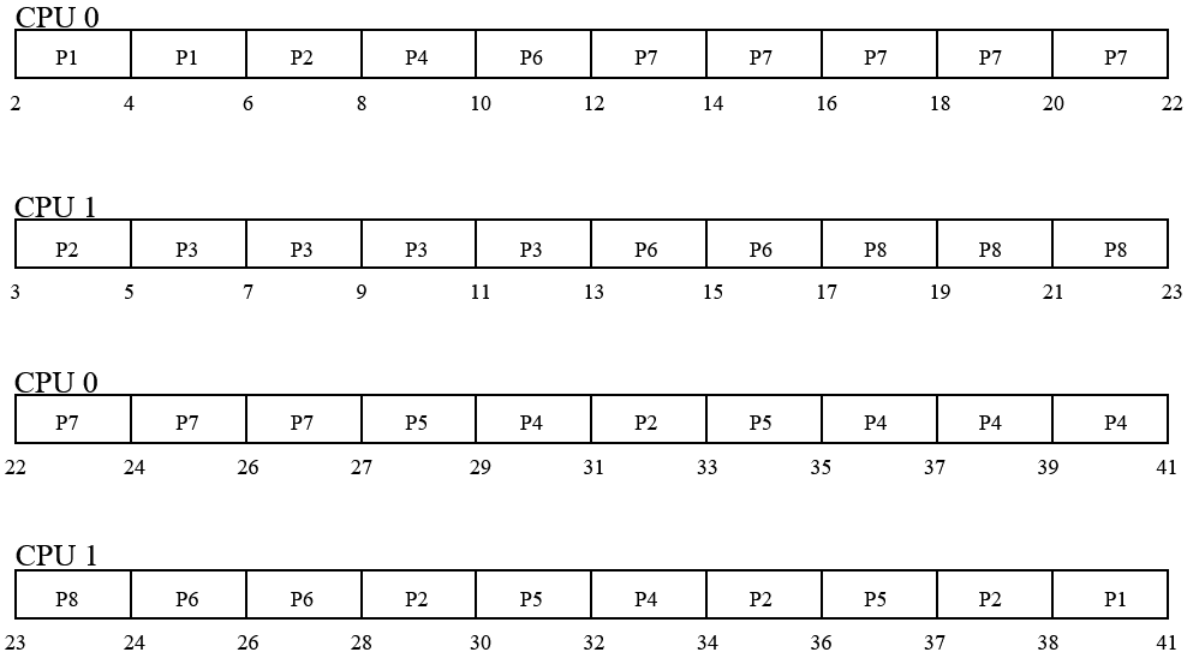


Hình 8: Kết quả test case 2 (phần 2)



Hình 9: Kết quả test case 2 (phần 3)

*** Gantt diagram:**



Hình 10: Gantt diagram cho test case 2

2 Quản lý bộ nhớ

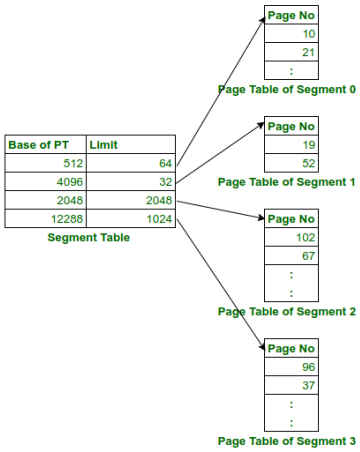
2.1 Cơ chế quản lý bộ nhớ của hệ điều hành đơn giản

Hệ điều hành mà nhóm mô phỏng sẽ quản lý bộ nhớ với kỹ thuật phân đoạn kết hợp phân trang (Segmentation with paging hoặc Segmented paging) mục đích là tận dụng được lợi thế của cả 2 phương pháp này.

Đây là cơ chế mà ở đó không gian bộ nhớ của mỗi process sẽ được chia thành một số Memory segment (hay Memory area) có thể khác nhau về kích thước để phân loại các dữ liệu (code segment, data segment, heap segment, stack segment,...) và được mỗi process quản lý thông qua bảng phân đoạn (Segment table). Các phân đoạn (Memory segment) này lại được hệ thống chia thành các trang được quản lý qua bảng phân trang (Page table). Mỗi phân đoạn sẽ sở hữu một bảng phân trang riêng. Sau đây là các thông số cấu hình cho hệ điều hành mô phỏng (dòng được tô đậm):

CPU bus	PAGE size	PAGE bit	No pg entry	PAGE Entry sz	PAGE TBL	OFFSET bit	PGT mem	MEMPHY	fram bit
20	256B	12	~4000	4byte	16KB	8	2MB	1MB	12
22	**256B**	**14**	**~16000**	**4byte**	**64KB**	**8**	**8MB**	**1MB**	**12**
22	512B	13	~8000	4byte	32KB	9	4MB	1MB	11
22	512B	13	~8000	4byte	32KB	9	4MB	128kB	8
16	512B	8	256	4byte	1kB	9	128K	128kB	4

Hình 11: Các thông số cho một hệ điều hành



Hình 12: Segmentation kết hợp Paging

2.2 Hiện thực

2.2.1 Cấp phát bộ nhớ

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

```

```
35   *alloc_addr = old_sbrk;
36
37   return 0;
38 }
```

Thông qua __alloc() hệ thống sẽ cấp phát một vùng nhớ theo yêu cầu của process caller. Ban đầu hệ thống sẽ tìm kiếm vùng nhớ (Memory region) phù hợp trong số các vùng trống mà phân đoạn hiện tại (xác định qua biến vmaid) đang có nếu tìm thấy hệ thống sẽ cấp phát vùng nhớ này cho caller. Ngược lại, tức là không có vùng nào đủ kích thước để đáp ứng yêu cầu hiện tại hệ thống sẽ phải tăng giới hạn của phân đoạn qua hàm inc_vma_limit().

Các hành vi của hàm inc_vma_limit() như sau:

- Tạo vùng nhớ mới (bắt đầu từ địa chỉ của con trỏ sbrk) có kích thước bằng với kích thước mà caller yêu cầu.
- Kiểm tra chồng chéo giữa vùng nhớ mới với các vùng đã tồn tại, nếu xảy ra chồng chéo thì cấp phát thất bại.
- Nếu không xảy ra chồng chéo, hệ thống sẽ thực hiện ánh xạ tới bộ nhớ RAM, nếu việc ánh xạ thuận lợi các thông tin mới về vùng nhớ mới sẽ được cập nhật, tức là cấp phát thành công.

2.2.2 Thu hồi bộ nhớ

```
1  int __free(struct pcb_t *caller, int vmaid, int rgid)
2  {
3    struct vm_rg_struct *rgnode = malloc(sizeof(struct vm_rg_struct));
4
5    if(rgid < 0 || rgid > PAGING_MAX_SYMTBL_SZ)
6      return -1;
7
8    /* TODO: Manage the collect freed region to freerg_list */
9    rgnode = get_symrg_byid(caller->mm, rgid);
10   /*enlist the obsoleted memory region */
11   enlist_vm_freerg_list(caller->mm, rgnode);
12
13   return 0;
14 }
```

Hàm __free() thực hiện công việc thu hồi vùng nhớ rgid, với hàm này hệ thống chỉ đưa vùng nhớ này vào danh sách các vùng nhớ trống để chờ đợi yêu cầu cấp phát tiếp theo.

2.2.3 Đọc/ghi bộ nhớ

Đầu tiên, để thực hiện đọc hay ghi vào một vùng nhớ hệ thống phải xác định được trang đích để thực hiện thao tác. Để thu được trang này, hệ điều hành mô phỏng của nhóm sử dụng hàm sau đây:

```
1  int pg_getpage(struct mm_struct *mm, int pgn, int *fpn, struct pcb_t *caller)
2  {
3    uint32_t pte = mm->pgd[pgn];
4
```

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

    int tgtfpn = PAGING_SWP(pte);//the target frame storing our variable

    /* TODO: Play with your paging theory here */
    /* Find victim page */
    if (find_victim_page(caller->mm, &vicpgn) < 0) return -1;
    vicpte = mm->pgd[vicpgn];
    vicfpn = PAGING_SWP(vicpte);
    /* Get free frame in MEMSWP */
    if (MEMPHY_get_freefp(caller->active_mswp, &swpfpn) < 0) return -1;

    /* 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);

    /* Update page table */
    //pte_set_swap() &mm->pgd;
    pte_set_swap(&vicpte, 0, swpfpn);
    /* Update its online status of the target page */
    //pte_set_fpn() & mm->pgd[pgn];
    pte_set_fpn(&pte, vicfpn);

    mm->pgd[pgn] = pte;
    mm->pgd[vicpgn] = vicpte;

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

  *fpn = PAGING_FPN(pte);

  return 0;
}
```

Nếu trang hiện tại đã hiện diện trong RAM (thể hiện qua bit present) ta có thể trực tiếp thu được trang này nhưng trong trường hợp ngược lại (page fault), ta cần phải thực hiện hoán đổi (swap) để đưa trang vào RAM sau đó mới truy xuất vào trang. Các bước cụ thể như sau:

- Tìm trang thay thế qua hàm find_victim_page() nếu không tìm được trang khả thi hệ thống sẽ dừng lập tức.
- Tìm khung (frame) trống trong bộ nhớ SWAP qua hàm MEMPHY_get_freefp().
- Sau khi đã tìm được cả trang thay thế và khung trống, hệ thống sẽ thực hiện swap out và swap in. Đầu tiên hệ thống sẽ sao chép nội dung của trang thay thế từ RAM sang SWAP. Sau đó sẽ sao chép nội dung khung chứa dữ liệu cần truy xuất từ SWAP sang lại RAM.
- Cập nhật lại nội dung của Page table entry (PTE) tương ứng với hai đối tượng đã được swap.

```c
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;
  }
  if (pg->pg_next == NULL) {
    *retpgn = pg->pgn;
    mm->fifo_pgn = NULL;
    free(pg);
  }
  else {
    while (pg->pg_next->pg_next != NULL) {
      pg = pg->pg_next;
    }
    *retpgn = pg->pg_next->pgn;
    free(pg->pg_next);
    pg->pg_next = NULL;
  }
  return 0;
}
```

Trên đây là hàm được sử dụng để tìm trang thay thế theo nguyên tắc First in, first out (FIFO).

Sau khi thu được trang đích chung ta đã có thể đọc (hoặc ghi) dữ liệu vào trang mong muốn. Bên dưới là pg_getval() và pg_setval() lần lượt hỗ trợ công việc đọc và ghi dữ liệu vào bộ nhớ:

```c
/*pg_getval - read value at given offset
 *@mm: memory region
 *@addr: virtual address to acess
 *@value: value
 *
 */
int pg_getval(struct mm_struct *mm, int addr, BYTE *data, struct pcb_t *caller)
{
  int pgn = PAGING_PGN(addr);
  int off = PAGING_OFFST(addr);
  int fpn;

  /* Get the page to MEMRAM, swap from MEMSWAP if needed */
  if(pg_getpage(mm, pgn, &fpn, caller) != 0)
    return -1; /* invalid page access */

  int phyaddr = (fpn << PAGING_ADDR_FPN_LOBIT) + off;

  MEMPHY_read(caller->mram,phyaddr, data);

  return 0;
}

/*pg_setval - write value to given offset
 *@mm: memory region
```

```
 *@addr: virtual address to acess
 *@value: value
 *
 */
int pg_setval(struct mm_struct *mm, int addr, BYTE value, struct pcb_t *caller)
{
  int pgn = PAGING_PGN(addr);
  int off = PAGING_OFFST(addr);
  int fpn;

  /* Get the page to MEMRAM, swap from MEMSWAP if needed */
  if(pg_getpage(mm, pgn, &fpn, caller) != 0)
    return -1; /* invalid page access */
  int phyaddr = (fpn << PAGING_ADDR_FPN_LOBIT) + off;

  MEMPHY_write(caller->mram,phyaddr, value);

   return 0;
}
```

2.3 Trả lời câu hỏi

Câu 1: Hệ điều hành được mô phỏng có thiết kế gồm nhiều Memory Segment hoặc Memory Area (MA). Vậy tác dụng của kiểu thiết kế này là gì ?

Trả lời:

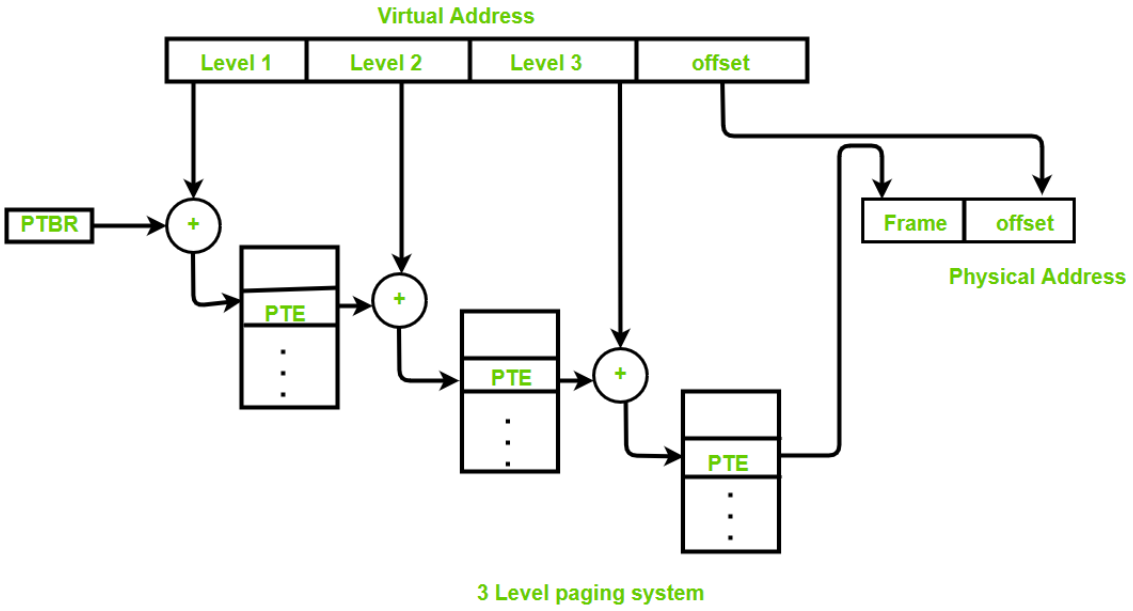
- Giải quyết được hiện tượng phân mảnh nội nhưng còn tồn tại phân mảnh ngoại.
- Thiết kế cung cấp sự độc lập nhất định giữa các MA của các process khác nhau và giữa các MA của các thành phần trong cùng một process. Điều này bảo vệ các MA của một process khỏi sự truy xuất của các process không mong muốn cũng như làm tăng tính chính xác và độ ổn định của các process khi chúng được thực thi đồng thời.
- Bộ nhớ được chia thành các MA có các kích thước khác nhau có tác dụng tối ưu hóa việc sử dụng bộ nhớ.

Câu 2: Sẽ ra sao nếu ta chia địa chỉ nhiều hơn 2 bậc trong hệ thống quản lý phân trang ?

Trả lời:

Với trường hợp kích thước của bảng phân trang lớn hơn kích thước của RAM, việc chia bảng phân trang thành nhiều cấp sẽ giảm đi kích thước của từng bảng điều này giúp cho kích thước của bảng trang phù hợp hơn để được sử dụng trong RAM.

Tuy nhiên, phân cấp phân trang đồng thời cũng làm tăng số lần truy xuất bộ nhớ mỗi khi có yêu cầu truy xuất địa chỉ vật lý của một page. Vì vậy, bậc phân cấp càng cao thì càng tốn chi phí truy xuất (phân cấp bậc 3 sẽ cần 3 lần truy xuất, bậc 4 sẽ cần 4 lần truy xuất,...).



Hình 13: Hệ thống phân trang bậc 3

Câu 3: Đâu là ưu thế và khó khăn của kỹ thuật Segmentation kết hợp Paging ?

Trả lời:

* Ưu thế:
  - Kích thước của bảng trang sẽ giảm đi do mỗi bảng trang sẽ lưu trữ cho một "loại" dữ liệu riêng, điều này làm giảm đi yêu cầu sử dụng bộ nhớ.
  - Hiện tượng phân mảnh ngoại xảy ra ít hơn nhờ kết hợp paging. Mặc dù các page có kích thước như nhau nhưng không phải process nào cũng bao gồm số segment như nhau do đó kích thước của segment table cũng khác nhau và chính sự khác nhau này gây ra phân mảnh ngoại.
  - Các thao tác swap trở nên đơn giản hơn khi thực hiện trên các trang (là một phần của segment) có kích thước không quá lớn.
* Khó khăn:
  - Do các page có kích thước như nhau trong khi yêu cầu sử dụng về các segment của mỗi process lại khác nhau nên hiện tượng phân mảnh nội vẫn xuất hiện trong các page.
  - Thời gian truy cấp bộ nhớ sẽ tăng lên so với Paging.

Câu 4: Điều gì sẽ xảy ra nếu hệ điều hành không thực hiện đồng bộ ?

Trả lời:

Khi nhiều process cùng truy xuất hay thay đổi dữ liệu trong vùng nhớ dùng chung mà không có sự đồng bộ sẽ gây ra xung đột dẫn đến mất mát dữ liệu. Vì vậy một hệ điều hành thiếu đi sự đồng bộ sẽ tồn tại vấn đề là dữ liệu không nhất quán dẫn đến giảm hiệu suất và sai lệch trong kết quả đầu ra.

Ví dụ trong thực thi định thời của hệ điều hành trong bài tập lớn này, vùng tranh chấp (critical section) giữa các process là mlq_ready_queue việc truy cập vào vùng này cần phải

được đồng bộ (cụ thể là nhóm sử dụng biến queue_lock) đảm bảo rằng trong một thời điểm chỉ một process được truy cập hay chỉnh sửa mlq_ready_queue. Xét trường hợp tại cùng thời điểm ban đầu cả hai process (cùng độ ưu tiên) được đưa vào hệ thống mà không có sự đồng bộ. Lúc này hệ thống sẽ đồng thời đưa hai process này vào cùng một hàng đợi và điều này tồn tại khả năng hàng đợi đó sẽ chỉ chứa một trong hai process mà không phải là cả hai như vậy sẽ làm mất mát dữ liệu và ảnh hưởng đến sự đúng đắn của hệ thống.

2.4 Phân tích kết quả kiểm thử

Sử dụng tệp đầu vào là "os_1_singleCPU_mlq_paging" có nội dung như sau:

```
2 1  8
1048576 16777216 0 0 0
1 s4  4
2 s3  3
4 m1s  2
6 s2  3
7 m0s  3
9 p1s  2
11 s0 1
16 s1 0
```

Hình 14: Test case

Đối với test case này, chỉ có process m1s (PID = 3) và m0s (PID = 5) có thao tác với bộ nhớ vậy nên ta sẽ tập trung vào các process này. Thông tin về m0s và m1s như sau:

```
1 7
alloc 300 0
alloc 100 1
free 0
alloc 100 2
write 102 1 20
write 1 2 1000
```

Hình 15: Thông tin về process m0s

```
1 8
alloc 300 0
alloc 100 1
free 0
alloc 100 2
free 2
free 1
```

Hình 16: Thông tin về process m1s

*** Kết quả:**

```
phuongduy@phuongduy-VirtualBox:~/ossim_source_code_part2_hk231_paging$ ./os os_1_singleCPU_mlq_paging
Time slot   0
ld_routine
Time slot   1
        Loaded a process at input/proc/s4, PID: 1 PRIO: 4
Time slot   2
        CPU 0: Dispatched process  1
        Loaded a process at input/proc/s3, PID: 2 PRIO: 3
Time slot   3
Time slot   4
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
        Loaded a process at input/proc/m1s, PID: 3 PRIO: 2
Time slot   5
Time slot   6
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  3
        Loaded a process at input/proc/s2, PID: 4 PRIO: 3
Time slot   7
        Loaded a process at input/proc/m0s, PID: 5 PRIO: 3
Time slot   8
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  3
Time slot   9
        Loaded a process at input/proc/p1s, PID: 6 PRIO: 2
Time slot  10
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  6
Time slot  11
        Loaded a process at input/proc/s0, PID: 7 PRIO: 1
Time slot  12
        CPU 0: Put process  6 to run queue
        CPU 0: Dispatched process  7
Time slot  13
Time slot  14
        CPU 0: Put process  7 to run queue
        CPU 0: Dispatched process  7
Time slot  15
Time slot  16
        CPU 0: Put process  7 to run queue
        CPU 0: Dispatched process  7
        Loaded a process at input/proc/s1, PID: 8 PRIO: 0
Time slot  17
Time slot  18
        CPU 0: Put process  7 to run queue
        CPU 0: Dispatched process  8
Time slot  19
Time slot  20
        CPU 0: Put process  8 to run queue
        CPU 0: Dispatched process  8
Time slot  21
Time slot  22
        CPU 0: Put process  8 to run queue
        CPU 0: Dispatched process  8
```

Hình 17: Kết quả test case (phần 1)

```
Time slot  23
Time slot  24
        CPU 0: Put process  8 to run queue
        CPU 0: Dispatched process  8
Time slot  25
        CPU 0: Processed  8 has finished
        CPU 0: Dispatched process  7
Time slot  26
Time slot  27
        CPU 0: Put process  7 to run queue
        CPU 0: Dispatched process  7
Time slot  28
Time slot  29
        CPU 0: Put process  7 to run queue
        CPU 0: Dispatched process  7
Time slot  30
Time slot  31
        CPU 0: Put process  7 to run queue
        CPU 0: Dispatched process  7
Time slot  32
Time slot  33
        CPU 0: Put process  7 to run queue
        CPU 0: Dispatched process  7
Time slot  34
        CPU 0: Processed  7 has finished
        CPU 0: Dispatched process  3
Time slot  35
Time slot  36
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  6
Time slot  37
Time slot  38
        CPU 0: Put process  6 to run queue
        CPU 0: Dispatched process  3
Time slot  39
Time slot  40
        CPU 0: Processed  3 has finished
        CPU 0: Dispatched process  6
Time slot  41
Time slot  42
        CPU 0: Put process  6 to run queue
        CPU 0: Dispatched process  6
Time slot  43
Time slot  44
        CPU 0: Put process  6 to run queue
        CPU 0: Dispatched process  6
Time slot  45
Time slot  46
        CPU 0: Processed  6 has finished
        CPU 0: Dispatched process  2
Time slot  47
Time slot  48
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  4
```

Hình 18: Kết quả test case (phần 2)

```
Time slot  49
Time slot  50
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  5
Time slot  51
Time slot  52
        CPU 0: Put process  5 to run queue
        CPU 0: Dispatched process  2
Time slot  53
Time slot  54
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  4
Time slot  55
Time slot  56
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  5
Time slot  57
Time slot  58
        CPU 0: Put process  5 to run queue
        CPU 0: Dispatched process  2
Time slot  59
Time slot  60
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  4
Time slot  61
Time slot  62
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  5
write region=1 offset=20 value=102
print_pgtbl: 0 - 768
00000000: 00000004
00000004: 00000003
00000008: 00000005
```

Hình 19: Kết quả test case (phần 3)

```
Time slot  63
write region=2 offset=1000 value=1
print_pgtbl: 0 - 768
00000000: 80000000
00000004: 00000003
00000008: 00000005
Time slot  64
        CPU 0: Put process  5 to run queue
        CPU 0: Dispatched process  2
Time slot  65
Time slot  66
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  4
Time slot  67
Time slot  68
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  5
write region=0 offset=0 value=0
print_pgtbl: 0 - 768
00000000: 80000000
00000004: 00000003
00000008: 00000005
Time slot  69
        CPU 0: Processed  5 has finished
        CPU 0: Dispatched process  2
Time slot  70
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  4
Time slot  71
Time slot  72
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  4
Time slot  73
Time slot  74
        CPU 0: Processed  4 has finished
        CPU 0: Dispatched process  1
Time slot  75
Time slot  76
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  77
Time slot  78
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  79
        CPU 0: Processed  1 has finished
        CPU 0 stopped
phuongduy@phuongduy-VirtualBox:~/ossim_source_code_part2_hk231_paging$
```

Hình 20: Kết quả test case (phần 4)

Tại thời điểm 62 (hình 18), màn hình hiển thị thông tin bảng phân trang process m0s cho biết rằng hiện tại m0s đã được cấp phát 3 trang nhớ.

- Do ban đầu, process yêu cầu 300 byte cho vùng nhớ 0 nhưng do kích thước trang nhớ được sử dụng trong hệ điều hành đơn giản này là 256 byte nên hệ thống sẽ dành ra 2

trang nhớ để đáp ứng yêu cầu này, sau đó m0s lại yêu cầu tiếp 100 byte cho vùng nhớ 1 và được hệ thống đáp ứng bằng 1 trang nhớ.

- Một yêu cầu khác là 100 byte cho vùng nhớ 2 nhưng trước đó process đã giải phóng vùng nhớ 0 nên vùng nhớ này có thể được sử dụng lại bởi process do đó hệ thống không cần cấp phát thêm một trang nhớ nào nữa.

Thời điểm 63 (hình 19), ta có thể thấy nội dung PTE số 0 đã bị thay đổi (từ 0x00000004 sang 0x80000000) đó là do lệnh khi thực hiện động tác ghi vào bộ nhớ hệ thống đã thực hiện hoán đổi nên gây ra sự thay đổi về giá trị PTE. Giá trị 0x80000000 nghĩa là trang tương ứng đang hiện diện trong RAM (bit present = 1) và frame number = 0.

Tài liệu tham khảo

1. Abraham Silberschatz, Peter Baer Galvin & Greg Gagne. (2018). *Operating System Concepts Tenth Edition*.

2. Sakharamgawade1. (2023). *Paged Segmentation and Segmented Paging*. Truy cập tại: Paged Segmentation and Segmented Paging