TRƯỜNG ĐẠI HỌC BÁCH KHOA TP HCM
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH

HỆ ĐIỀU HÀNH

Báo cáo bài tập lớn 2

Simple Operating System

GVHD: Trần Ngọc Anh Tú

SV thực hiện: Nguyễn Hoài Danh 1610391
Nguyễn Giáp Phương Duy 1610473
Bùi Bảo Cường 1610342

Tp. Hồ Chí Minh, Tháng 5/2018

Tóm tắt báo cáo

Bài tập lớn lần này yêu cầu sinh viên mô phỏng lại một hệ điều hành đơn giản ,qua đó ,chúng ta sẽ hiểu rõ hơn các khái niệm nền tảng về các thành phầnchính của một hệ điều hành.Chi tiết hơn,chúng ta sẽ hiện thực lại ba thành phần cơ bản đó là :

- Bộ định thời.
- Cơ chế đồng bộ hóa
- Các toán tử liên quan đến việc phân bổ từ bộ nhớ ảo sang bộ nhớ vật lí

Mục lục

1 Giới thiệu **3**
- 1.1 Định thời CPU . . . 3
  - 1.1.1 Đặt vấn đề . . . 3
  - 1.1.2 Bộ định thời CPU . . . 3
  - 1.1.3 Dispatcher . . . 3
- 1.2 Quản lý bộ nhớ . . . 4
  - 1.2.1 Đặt vấn đề . . . 4
  - 1.2.2 Không gian địa chỉ luận lý và không gian địa chỉ vật lý . . . 4
  - 1.2.3 Phân đoạn . . . 4
  - 1.2.4 Phân trang . . . 5

2 Hiện thực **6**
- 2.1 Scheduler . . . 6
  - 2.1.1 Yêu cầu . . . 6
  - 2.1.2 Qúa trình hiện thực . . . 7
  - 2.1.3 Input sched_0: . . . 8
  - 2.1.4 Output sched_0 . . . 8
  - 2.1.5 Grantt diagram sched_0 . . . 10
  - 2.1.6 Input sched_1 . . . 10
  - 2.1.7 Output sched_1 . . . 10
  - 2.1.8 Grantt diagram sched_1 . . . 13
- 2.2 Memory . . . 13
  - 2.2.1 Yêu cầu . . . 13
  - 2.2.2 Qúa trình hiện thực . . . 14
  - 2.2.3 Memory test . . . 20
  - 2.2.4 Memory test for m0 . . . 20
  - 2.2.5 Memory test for m1 . . . 21
- 2.3 Answer the questions . . . 22
- 2.4 Kết luận . . . 22

Tài liệu tham khảo . . . 22

Chương 1

Giới thiệu

1.1 Định thời CPU

1.1.1 Đặt vấn đề

Trong hệ thống chỉ có đơn vi xử lí (single-processor),thì đồng nghĩa chỉ có một process được chạy trong một thời điểm nhất định,những process còn lại phải chờ cho đến khi CPU trống thì mới được xử lí tiếp.Khi một process được thực thi,nó sẽ chạy cho đến khi rơi vào trạng thái chờ (Điển hình là chờ thực thi I/O),đối với hệ thống máy tính đơn giản,thì lúc này CPU sẽ "răng rỗi" không làm gì cả,chờ cho đến khi process này thực thi xong I/O thì CPU mới hoạt động tiếp.Điều này sẽ gây ra việc lãng phí thời gian,sử dụng CPU không hiệu quả.Vì vậy hệ thống đa chương ra đời.

Mục đích của hệ thống đa chương đó là có nhiều process cùng chạy trong một thời điểm nhất định để có thể tối đa hóa độ lợi CPU.Lúc này nhiều process sẽ được giữ trong bộ nhờ ở cùng một thời điểm,khi một process rơi vào trạng thái chờ,hệ điều hành sẽ lấy một process khác trong hàng đợi và đưa cho CPU xử lí,việc này sẽ được lặp đi lặp lại.Lúc này CPU sẽ không bao giờ nhàn rỗi.

1.1.2 Bộ định thời CPU

Bất cứ khi nào CPU rảnh, hệ điều hành phải chọn một trong những quá trình trong hàng đợi sẵn sàng để thực thi. Chọn quá trình được thực hiện bởi bộ định thời biểu ngắn (short-term scheduler) hay bộ định thời CPU. Bộ định thời này chọn các quá trình trong bộ nhớ sẵn sàng thực thi và cấp phát CPU tới một trong các quá trình đó.

Hàng đợi sẵn sàng không nhất thiết là hàng đợi vào trước, ra trước (FIFO). Xem xét một số giải thuật định thời khác nhau, một hàng đợi sẵn sàng có thể được cài đặt như một hàng đợi FIFO, một hàng đợi ưu tiên, một cây, hay đơn giản là một danh sách liên kết không thứ tự. Tuy nhiên, về khái niệm tất cả các quá trình trong hàng đợi sẵn sàng được xếp hàng chờ cơ hội để chạy trên CPU. Các mẫu tin trong hàng đợi thường là khối điều khiển quá trình của quá trình đó. [1]

1.1.3 Dispatcher

Dispatcher sẽ chuyển quyền điều khiển CPU về cho process được chon bởi bộ định thời ngắn hạn Bao gồm:

- Chuyển ngữ cảnh (sử dụng thông tin ngữ cảnh trong PCB).
- Chuyển về user mode

- Nhảy đến vị trí thích hợp (chính là program counter trong PCB) trong chương trình ứng dụng để quá trình tiếp tục thực thi

Công việc này gây ra phí tổn Dispatch latency: thời gian dispatcher cần từ lúc dừng một process đến lúc một process khác tiếp tục chạy

1.2 Quản lý bộ nhớ

1.2.1 Đặt vấn đề

Nhờ vào bộ đình thời,chúng ta có thể gia tăng độ lợi của CPU và tốc độ phản hồi của máy tính tới người dùng.Tuy nhiên,để làm được điều này,thì chúng ta phải chứa nhiều process tại cùng một thời điểm ở trong bộ nhớ.Vấn đề được đặt ra là chúng ta phải phân chia cho các process này có có các vùng nhớ độc lập với nhau.Việc phân chia này nhằm bảo vệ process này truy cập dữ liệu của process khác.

1.2.2 Không gian địa chỉ luận lý và không gian địa chỉ vật lý

Một địa chỉ được tạo ra bởi CPU thường được gọi là địa chỉ luận lý (logical address), ngược lại một địa chỉ được xem bởi đơn vị bộ nhớ-nghĩa là, một địa chỉ được nạp vào thanh ghi địa chỉ bộ nhớ-thường được gọi là địa chỉ vật lý (physical address).

Các phương pháp liên kết địa chỉ thời điểm biên dịch và thời điểm nạp tạo ra địa chỉ luận lý và địa chỉ vật lý xác định. Tuy nhiên, cơ chế liên kết địa chỉ tại thời điểm thực thi dẫn đến sự khác nhau giữa địa chỉ luận lý và địa chỉ vật lý. Trong trường hợp này, chúng ta thường gọi địa chỉ luận lý như là địa chỉ ảo (virtual address). Tập hợp tất cả địa chỉ luận lý được tạo ra bởi chương trình là không gian địa chỉ luận lý ; tập hợp tất cả địa chỉ vật lý tương ứng địa chỉ luận lý này là không gian địa chỉ vật lý. Do đó, trong cơ chế liên kết địa chỉ tại thời điểm thực thi, không gian địa chỉ luận lý và không gian địa chỉ vật lý là khác nhau.[2]

1.2.3 Phân đoạn

Phân đoạn là một cơ chế quản lí bộ nhớ,phân chương trình thành các đoạn.,cho phép không gian địa chỉ bộ nhớ vật lý cấp cho process không liên tục nhau.Một không gian địa chỉ luận lí là một tập hợp của các đoạn.Để hiện thực một cách đơn giản,mỗi đoạn sẽ được đánh chỉ số riêng biệt.Mỗi khi chương trình được biên dịch,bộ biên dịch sẽ tự động phân chia các đoạn dựa trên chương trình được nạp vào.

VD:Bộ biên dịch C phân các đoạn ra như sau:

1. Code
2. Biến toàn cục
3. Heap (cấp phát động)
4. Stack dùng bởi mỗi thread
5. Thư viện C cơ bản

1.2.4 Phân trang

Kỹ thuật phân trang cho phép không gian địa chỉ vật lí của một process có thể không liên tục nhau.Bố nhớ được chia thành các khối cố định và có kích thước bằng nhau,được gọi là frame.Bộ nhớ luận lý được cũng được chia thành các khối cố định,bằng nhau được gọi là page.

Chương 2

Hiện thực

2.1 Scheduler

2.1.1 Yêu cầu

Chúng ta sẽ hiện thực scheduler (giải sử hệ thống chỉ có một vi xử lí).Hệ điều hành sử dụng priority feedback queue để quyết định xem process nào sẽ được thực thi khi CPU đang rãnh.Cho mỗi chương trình:

- Tạo một process và PCB cho process đó.
- Chương trình được load vào hệ thống
- PCB của process đó sẽ được bỏ vào ready queue và chờ CPU phản hồi
- CPU chạy process đó theo round-robin
- Sau khi process kết thúc quantum time,nếu process đó chưa hoàn thành thì nó sẽ bị dừng lại và đẩy vào run queue.
- CPU vẫn tiếp tục lấy process từ ready queue và chạy tiếp.
- Nếu ready queue không còn process nào,thì sẽ lấy tất cả process từ run queue bỏ vào ready queue

Lưu ý: ready queue là một priority queue,tức là mỗi lần lấy process từ queue này,ta sẽ lấy process có priority lớn nhất.

Công việc trong phần này của chúng ta là hiện thực:

- Trong file queue.c : hiện thực hai hàm enqueue() và dequeue() cho priority queue
- Trong file sched.c : Hiện thực hàm get_proc() để lấy process cho CPU xử lí,và xử lí trường hợp ready queue trống

2.1.2 Qúa trình hiện thực

Hiện thực enqueue và dequeue

```
void enqueue(struct queue_t *q, struct pcb_t *proc){er
    int size = q->size;
    q->proc[size] = proc;   q->size++;
}
struct pcb_t *dequeue(struct queue_t *q){
    int q_size = q->size;
    if (q_size == 0)
        return NULL;
    uint32_t highest_prioprity = q->proc[0]->priority;
    uint32_t prioprity;
    int hp_index = 0;
    //Get the index of highest prioprity process
    for (int i = 0; i < q_size; i++){
        prioprity = q->proc[i]->priority;
        if (prioprity > highest_prioprity){
            highest_prioprity = prioprity;
            hp_index = i;
        }
    }
    struct pcb_t *r_proc = q->proc[hp_index];
    //Delete highest prioprity process in q
    for (int i = hp_index; i < q_size - 1; i++)
        q->proc[i] = q->proc[i + 1];
    q->proc[q_size - 1] = NULL;     q->size--;
    return r_proc;
}
```

Hiện thực get_proc

```
struct pcb_t *get_proc(void)
{
    struct pcb_t *proc = NULL;
    pthread_mutex_lock(&queue_lock);
    if (queue_empty())
    {
        pthread_mutex_unlock(&queue_lock);
        return NULL;
    }

    if (empty(&ready_queue))
    {
        //Push all process form run_queue to ready_queue
        while (!empty(&run_queue))
        {
            proc = dequeue(&run_queue);
            enqueue(&ready_queue, proc);
        }
```

```
    }
    //Get process from ready dequeue
    proc = dequeue(&ready_queue);
    pthread_mutex_unlock(&queue_lock);
    return proc;
}
```

Giải thuật get_proc:

- Nếu size của ready_queue là 0 , ta chuyển tất cả các phần tử proc về ready queue đồng thời xóa chúng trong run queue rồi sau đó mới dùng hàm dequeue đã hiện thực cho ready queue để gọi process có priority cao nhất.
- Nếu size của ready_queue khác 0, ta chỉ cần gọi hàm dequeue trên ready queue.
- Trả về process đó.

2.1.3 Input sched_0:

```
2 1 2
0 s0
4 s1
```

2.1.4 Output sched_0

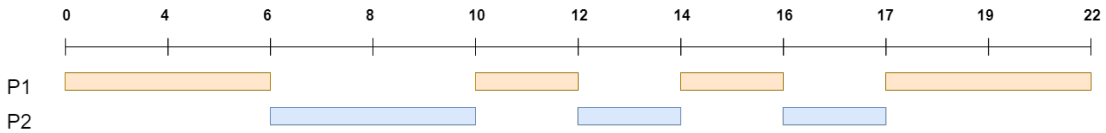
Sau khi thực hiện thì chúng ta nhận được kết quả như sau:

```
------ SCHEDULING TEST 0 ---------------------------------------
./os sched_0
Time slot   0
Loaded a process at input/proc/s0, PID: 1
CPU 0: Dispatched process  1
Time slot   1
Time slot   2
CPU 0: Put process  1 to run queue
CPU 0: Dispatched process  1
Time slot   3
Time slot   4
Loaded a process at input/proc/s1, PID: 2
CPU 0: Put process  1 to run queue
CPU 0: Dispatched process  2
Time slot   5
Time slot   6
CPU 0: Put process  2 to run queue
CPU 0: Dispatched process  2
Time slot   7
Time slot   8
CPU 0: Put process  2 to run queue
CPU 0: Dispatched process  1
```

```
Time slot   9
Time slot  10
CPU 0: Put process  1 to run queue
CPU 0: Dispatched process  2
Time slot  11
Time slot  12
CPU 0: Put process  2 to run queue
CPU 0: Dispatched process  1
Time slot  13
Time slot  14
CPU 0: Put process  1 to run queue
CPU 0: Dispatched process  2
Time slot  15
CPU 0: Processed  2 has finished
CPU 0: Dispatched process  1
Time slot  16
Time slot  17
CPU 0: Put process  1 to run queue
CPU 0: Dispatched process  1
Time slot  18
Time slot  19
CPU 0: Put process  1 to run queue
CPU 0: Dispatched process  1
Time slot  20
Time slot  21
CPU 0: Put process  1 to run queue
CPU 0: Dispatched process  1
Time slot  22
CPU 0: Processed  1 has finished
CPU 0 stopped
```

Time slot	Dispatcher	ready_queue	run_queue
0		P1	
0.x	P1		
1	P1		
2		empty	P1
2.x			P1
2,x	P1		
3	P1		
4		empty	P1
4.x		P1	
4.x	P1	P2	
5	P1	P2	
6	P2	empty	P1
6.x	P2	P1	
7	P2	P1	
8	P1	P2	
9	P1	P2	
10	P2	P1	
11	P2	P1	
12	P1	P2	
13	P1	P2	
14	P2	P1	
15	P1		
...	P1		
22			

2.1.5 Grantt diagram sched_0



Hình 2.1: Gantt diagram for `sched_0`

2.1.6 Input sched_1

```
2 1 4
0 s0
4 s1
6 s2
7 s3
```

2.1.7 Output sched_1

Sau khi thực hiện thì chúng ta nhận được kết quả như sau:

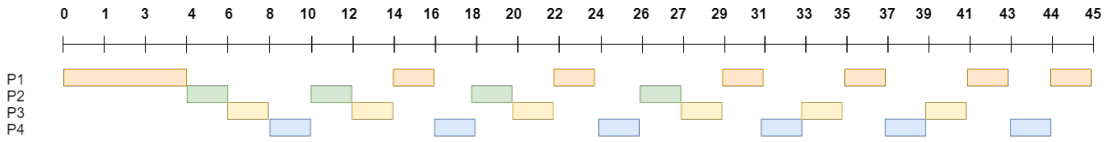
```
------ SCHEDULING TEST 1 --------------------------------------------
./os sched_1
Time slot   0
Loaded a process at input/proc/s0, PID: 1
Time slot   1
CPU 0: Dispatched process  1
Time slot   2
Time slot   3
CPU 0: Put process  1 to run queue
CPU 0: Dispatched process  1
Time slot   4
Loaded a process at input/proc/s1, PID: 2
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
Time slot   7
CPU 0: Put process  2 to run queue
CPU 0: Dispatched process  3
Loaded a process at input/proc/s3, PID: 4
Time slot   8
Time slot   9
CPU 0: Put process  3 to run queue
CPU 0: Dispatched process  4
Time slot  10
Time slot  11
CPU 0: Put process  4 to run queue
CPU 0: Dispatched process  2
Time slot  12
Time slot  13
CPU 0: Put process  2 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  1
Time slot  16
Time slot  17
CPU 0: Put process  1 to run queue
CPU 0: Dispatched process  4
Time slot  18
Time slot  19
CPU 0: Put process  4 to run queue
CPU 0: Dispatched process  2
Time slot  20
Time slot  21
CPU 0: Put process  2 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  4
Time slot  26
Time slot  27
CPU 0: Put process  4 to run queue
CPU 0: Dispatched process  2
Time slot  28
CPU 0: Processed  2 has finished
CPU 0: Dispatched process  3
Time slot  29
Time slot  30
CPU 0: Put process  3 to run queue
CPU 0: Dispatched process  1
Time slot  31
Time slot  32
CPU 0: Put process  1 to run queue
CPU 0: Dispatched process  4
Time slot  33
Time slot  34
CPU 0: Put process  4 to run queue
CPU 0: Dispatched process  3
Time slot  35
Time slot  36
CPU 0: Put process  3 to run queue
CPU 0: Dispatched process  1
Time slot  37
Time slot  38
CPU 0: Put process  1 to run queue
CPU 0: Dispatched process  4
Time slot  39
Time slot  40
CPU 0: Put process  4 to run queue
CPU 0: Dispatched process  3
Time slot  41
Time slot  42
CPU 0: Processed  3 has finished
CPU 0: Dispatched process  1
Time slot  43
Time slot  44
CPU 0: Put process  1 to run queue
CPU 0: Dispatched process  4
Time slot  45
CPU 0: Processed  4 has finished
```

```
CPU 0: Dispatched process  1
Time slot  46
CPU 0: Processed  1 has finished
CPU 0 stopped
```

Time slot	Dispatcher	ready_queue	run_queue
0		P1	
1	P1		
2	P1		
3	P1		
4	P1	P2	
5	P2		P1
6	P2	P3	P1
7	P3	P4	P1,P2
8	P3	P4	P1,P2
9	P4	P1,P2,P3	
10	P4	P1,P2,P3	
11	P2	P1,P3	P4
12	P2	P1,P3	P4
13	P3	P1	P4,P2
14	P3	P1	P4,P2
15	P1	P4,P2,P3	
16	P1	P4,P2,P3	
...	...	...	...
45	P1		

2.1.8 Grantt diagram sched_1



Hình 2.2: Gantt diagram for sched_1

2.2 Memory

2.2.1 Yêu cầu

Bộ nhớ ảo của chúng ta sẽ sử dụng cơ chế phân trang kết hợp phân đoạn cho việc quản lý bộ nhớ.Dung lượng bộ nhớ RAM là 1MB nên chúng ta sẽ sử dụng 20 bit để thể hiện địa chỉ của bộ nhớ.Với cơ chế phân trang kết hợp phân đoạn,Chúng ta sẽ sử dụng 5 bit cho segment index,5 bit cho page index và 10 bit còn lại cho offset
Trong file mem.c:

- Hiện thực get_page_table() để tìm page table trong segment với index segment là tham số
- Hiện thực translate() Để chuyển từ địa chỉ ảo sang địa chỉ vật lý

2.2.2 Qúa trình hiện thực

Hiện thực get_page_table

```
static struct page_table_t *get_page_table(addr_t index, struct
    seg_table_t *seg_table)
{ // first level table

    if (seg_table->size == 0)
        return NULL;
    int i = 0;
    addr_t seg_index;
    struct page_table_t *r_pages = NULL;

    for (i = 0; i < seg_table->size; i++)
    {
        // Enter your code here
        seg_index = seg_table->table[i].v_index;
        if (seg_index == index)
        {
            r_pages = seg_table->table[i].pages;
            break;
        }
    }
    return r_pages;
}
```

Giải thuật get_page_table:

- Với đầu vào là [index] của segment và pointer trỏ đến một segment table ([seg_table]), ta duyệt từng segment trong bảng.
- Nếu ở tại segment đó, [v_index] của segment trùng với [index] được cho trong thông số, ta trả về page table ứng với segment đó.
- Tuy nhiên, trong vòng lặp for nếu không tìm thấy segment phù hợp, hàm sẽ trả về NULL.

Hiện thực translate

```
static int translate(addr_t virtual_addr,addr_t *physical_addr,struct
    pcb_t *proc)
{ // Process uses given virtual address
    /* Offset of the virtual address */
    addr_t offset = get_offset(virtual_addr);
    /* The first layer index */
    addr_t first_lv = get_first_lv(virtual_addr);
    /* The second layer index */
    addr_t second_lv = get_second_lv(virtual_addr);
    /* Search in the first level */
    struct page_table_t *page_table = NULL;
    page_table = get_page_table(first_lv, proc->seg_table);
    if (page_table == NULL) return 0;
```

```
    int i;
    addr_t p_index;
    for (i = 0; i < page_table->size; i++){
        if (page_table->table[i].v_index == second_lv){
            /* TODO: Concatenate the offset of the virtual addess
             * to [p_index] field of page_table->table[i] to
             * produce the correct physical address and save it to
             * [*physical_addr]  */
            p_index = page_table->table[i].p_index;
            *physical_addr = (p_index << OFFSET_LEN) + offset;
            return 1;
        }
    }
    return 0;
}
```

Giải thuật translate:

- Với đầu vào là địa chỉ ảo [virtual_addr], ta phân rã địa chỉ ra thành 3 thành phần: [offset] bằng hàm get_offset, [first_lv] bằng hàm get_first_lv, [second_lv] bằng hàm get_second_lv.
- Với [first_lv], ta sử dụng hàm get_page_table ở trên, với thông số là [first_lv] và segment table từ process [proc], để lấy được page table của segment hiện tại ứng với địa chỉ ảo. Nếu không tìm thấy page table trong segment table, hàm sẽ tự động trả về 0.
- Với [second_lv], ta duyệt mọi page trong page table mới tìm được. Nếu tại trang đó, [v_index] trùng với [second_lv], ta sẽ lấy [p_index] tại trang đó và nối vào bên trái của [offset] tìm được ở trên. Sau đó, hàm sẽ trả về 1. Nếu không tìm thấy page cần tìm trong page table, hàm trả ngay lập tức trả về 0.

Hiện thực alloc_mem

```
addr_t alloc_mem(uint32_t size, struct pcb_t *proc)
{
    pthread_mutex_lock(&mem_lock);
    addr_t ret_mem = 0;

    uint32_t num_pages = ((size % PAGE_SIZE) == 0) ? size / PAGE_SIZE
        : size / PAGE_SIZE + 1; // Number of pages we will use
    int mem_avail = 0;
                                // We could allocate new memory
       region or not?

    uint32_t free_p_page = 0;

    //Get physical page available
    for (int i = 0; i < NUM_PAGES; i++)
        if (_mem_stat[i].proc == 0)
            ++free_p_page;
```

```
//Get number mem need to use,heap + stack ?
uint32_t using_v_mem = proc->bp + num_pages * PAGE_SIZE;

//Check that physical and virtual mem is available to use
if (num_pages <= free_p_page)
    if (using_v_mem <= (1 << ADDRESS_SIZE))
        mem_avail = 1;

if (mem_avail)
{
    int i = 0;
    int pre_page = -1;
    int p_index = -1;
    int cnt_page = 0;

    //Find first page
    while (_mem_stat[i].proc != 0)
        i++;

    pre_page = p_index = i;
    _mem_stat[i].proc = proc->pid;
    _mem_stat[i].index = cnt_page++;
    ++i;
    //Update mem status of process
    while (cnt_page != num_pages)
    {
        //If mem isn't used
        if (_mem_stat[i].proc == 0)
        {
            //Update mem status
            _mem_stat[i].proc = proc->pid;
            _mem_stat[i].index = cnt_page++;
            _mem_stat[pre_page].next = i;
            pre_page = i;
        }
        ++i;
    }
    //Next index of last page is -1
    _mem_stat[pre_page].next = -1;

    int seg_size;
    int page_size;
    addr_t bp = ret_mem; //break point

    /*Add entries to segment table and page table*/
    //Init first seg table
    if (!proc->seg_table->size)
    {
        seg_size = ++proc->seg_table->size;
```

```
            proc->seg_table->table[seg_size - 1].v_index = seg_size -
                1;
        }
        //If proc already has seg teable
        seg_size = proc->seg_table->size;
        //Init first page table for first segment
        if (!proc->seg_table->table[seg_size - 1].pages)
        {
            proc->seg_table->table[seg_size - 1].pages = malloc(
                sizeof(struct page_table_t));
            proc->seg_table->table[seg_size - 1].pages->size = 0;
        }

        while (num_pages)
        {
            /*----------------Page process-----------------------*/
            page_size = proc->seg_table->table[seg_size - 1].pages->
                size;
            //If page size < limit page of one segment,use that
                segment to store page
            if (page_size + 1 <= (1 << PAGE_LEN))
            {
                //Set new page
                page_size = ++proc->seg_table->table[seg_size - 1].
                    pages->size;
                //
                proc->seg_table->table[seg_size - 1].pages->table[
                    page_size - 1].v_index = get_second_lv(bp);

                //Map physic index of page equal first index in
                    _mem_stat of process
                proc->seg_table->table[seg_size - 1].pages->table[
                    page_size - 1].p_index = p_index;
                //Get next index in _mem_stat
                p_index = _mem_stat[p_index].next;
                num_pages--;
                //Increase breakpoint one page
                bp += PAGE_SIZE;
            }
            else //Creat new segment to store page
            {
                /*----------------Segment process----------------------*/
                seg_size = ++proc->seg_table->size;

                //Set index for segment,auto increament from 0
                proc->seg_table->table[seg_size - 1].v_index =
                    seg_size - 1;

                //Init new page table for segment
```

```
                if (!proc->seg_table->table[seg_size - 1].pages)
                {
                    proc->seg_table->table[seg_size - 1].pages =
                        malloc(sizeof(struct page_table_t));
                    proc->seg_table->table[seg_size - 1].pages->size
                        = 0;
                }
            }
        }
    }
    pthread_mutex_unlock(&mem_lock);
    return ret_mem;
}
```

Giải thuật alloc_mem:

- Trước khi bắt đầu công việc allocate cho process, ta phải sử dụng hàm pthread_mutex_lock(&mem_lock) để đảm bảo các tài nguyên sử dụng không được xung khắc với nhau.
- Hàm sẽ bắt đầu thực hiện với các công việc sau đây: Đầu tiên hàm sẽ tính số page [num_pages] mà ta muốn process được allocate vào [_mem_stat]. Tiếp theo, hàm sẽ kiểm tra ở vùng nhớ vật lý [_mem_stat] bằng cách xem số page trống trong [_mem_stat] có đủ cho số page cần allocate hay không. Cuối cùng, hàm sẽ kiểm tra ở vùng nhớ ảo với điều kiện kiểm tra như lúc ở vùng nhớ vật lý, nhưng bằng cách lấy [bp] của process và cộng với num_page * PAGE_SIZE để xem kết quả có vượt quá RAM_SIZE hay không.
- Sau khi kiểm tra các điều kiện, nếu thỏa [mem_avail] sẽ bằng 1 và ta bắt đầu công đoạn allocate các page cho process. Ta lưu byte đầu tiên của process thông qua [bp] bằng [ret_mem] để sau khi hoàn tất hàm sẽ trả về [ret_mem]. [bp] của process sau đó sẽ được thêm vào num_pages * PAGE_SIZE. Tiếp theo, hàm sẽ duyệt mọi page trong [_mem_stat]. Nếu tại page đó [proc] bằng 0, có nghĩa page đó chưa có một process nào sử dụng, và ta được quyền allocate một page lên trên đó. Khi đó, ta sẽ update [index] và [next] của page đó, với [index] là số thứ tự của page trong dánh sách page sẽ được allocate của process, và [next] là chỉ số tiếp theo của page kế tiếp sẽ được allocate trong [_mem_stat]. Tiếp theo, ta sẽ thêm entries cho process với page vừa được allocated vào segment table và page table của process [proc]. Công việc bắt đầu bằng việc ta tìm địa chỉ ảo bằng cách lấy [ret_mem] cộng với (index – 1) * PAGE_SIZE (với [index] là index hiện tại của trang đang được allocated). Từ địa chỉ ảo đó ta tìm [first_lv] bằng hàm get_first_lv và [second_lv] bằng hàm get_second_lv. Với [first_lv], ta duyệt mọi segment trong segment table của process. Nếu [v_index] của segment đó trùng với [first_lv], ta chọn page table ứng với segment tìm được. Nếu không tìm được segment trong segment table, ta sẽ tạo một segment mới với [v_index] của segment mới này là [first_lv], và page table sẽ là một table mới của segment đó. Với [second_lv], ta lần lượt duyệt các page trong page table mới tìm được. Nếu tại một page, [v_index] của page đó trùng với [second_lv], ta update [p_index] của page đó bằng [index] của page vừa được allocated trong [_mem_stat]. Nếu không tìm thấy page nào, ta tạo một page mới trong page table, update [v_index] của page đó bằng [second_lv] và [p_index] bằng [index] của page vừa được allocated. Cuối cùng, ta sử dụng hàm pthread_mutex_unlock(&mem_lock) và hàm trả về giá trị [ret_mem].

Hiện thực free_mem

```
int free_mem(addr_t address, struct pcb_t *proc)
{
    pthread_mutex_lock(&mem_lock);
    addr_t virtual_addr = address;

    addr_t physical_addr;
    int num_pages = 0;
    struct page_table_t *page_table = NULL;

    if (translate(address, &physical_addr, proc))
    {
        //physic address = p_index << OFFSET_LEN + offset => p_index
           = physic addr >> OFFSET_LEN
        uint32_t p_index = physical_addr >> OFFSET_LEN;

        //Free _mem_stat by assign proc stat to zero
        while (p_index != -1) //last page has next = -1
        {
            _mem_stat[p_index].proc = 0;
            p_index = _mem_stat[p_index].next;
            num_pages++;
        }

        //Free unused entries
        while (num_pages != 0)
        {
            //Get v_index of segment
            addr_t first_lv = get_first_lv(virtual_addr);
            //Get page table have v_index segment
            page_table = get_page_table(first_lv, proc->seg_table);

            //Get v_index of page
            addr_t second_lv = get_second_lv(virtual_addr);

            if (page_table == NULL)
                break;

            for (int i = 0; i < (1 << PAGE_LEN); i++)
            {
                if (second_lv == page_table->table[i].v_index)
                {
                    //Clear entry
                    page_table->table[i].v_index = -1;
                    page_table->table[i].p_index = -1;
                    //
                    virtual_addr = virtual_addr + PAGE_SIZE;
                    num_pages--;
                    break;
```

```
                }
            }
        }
    }
    pthread_mutex_unlock(&mem_lock);
    return 0;
}
```

Giải thuật free_mem:

- Trước khi bắt đầu công việc deallocate cho process, ta phải sử dụng hàm pthread_mutex_lock(&mem_lock) để đảm bảo các tài nguyên sử dụng không được xung khắc với nhau.
- Với [address] là địa chỉ ảo của page đầu tiên, ta cần tìm [p_index] của page đầu tiên đó trong [_mem_stat] bằng cách: chuyển đổi địa chỉ ảo [address] sang địa chỉ vật lý [p_addr] bằng hàm translate, và lấy được [p_index] bằng cách bỏ offset của địa chỉ vật lý và lấy phần còn lại.
- Với [p_index] cùa page đầu tiên cần deallocate, ta update [proc] tại vị trí [p_index] trong [_mem_stat] thành 0 (để biết rằng tại vị trí đó không có process nào sử dụng) và update [p_index] bằng [next] của frame tại vị trí đó. Tiếp theo, ta sử dụng [address] để tìm [first_lv] bằng hàm get_first_lv, [second_lv] bằng hàm get_second_lv, và update [address] bằng cách cộng nó thêm với PAGE_SIZE (do trong bộ nhớ ảo các page của process liên tiếp nhau, nên để tìm địa chỉ ảo của page kế tiếp ta chỉ cần thực hiện hành động như trên). Với [first_lv], ta duyệt các segment trong segment table của process. Nếu tại segment đó [v_index] trùng với [first_lv], ta lấy page table của segment đó (trong việc tìm kiếm này không cần xử lý trường hợp không tìm thấy segment, do một khi page đã được allocated thì chắc chắn nó sẽ xuất hiện tại một page table của một segment nào đó). Với [second_lv], ta duyệt từng page trong page table mới tìm được. Nếu [v_index] tại một page trùng với [second_lv], ta xóa page đó đi bằng cách lấy page cuối cùng trong page table và chèn vào page cần xóa. Nếu sau khi xóa trong kích thước của page table là 0, có nghĩa là segment đó không có page nào. Khi đó, ta xóa segment đó đi bằng cách lấy segment cuối chèn vào segment cần xóa. Cuối cùng, ta sử dụng hàm pthread_mutex_unlock(&mem_lock) và hàm trả về giá trị 0.
-

2.2.3 Memory test

2.2.4 Memory test for m0

```
1 7
alloc 13535 0
alloc 1568 1
free 0
alloc 1386 2
alloc 4564 4
write 102 1 20
write 21 2 1000
```

Sau khi thực hiện thì chúng ta nhận được trạng thái của RAM kết quả như bảng sau

Page	Pid	Index	Next
0	1	0	1
1	1	1	-1
2	1	0	3
3	1	1	4
4	1	2	5
5	1	3	6
6	1	4	-1
14	1	0	15
15	1	1	-1

Giải thích kết quả m0:

- alloc 13535 0: Từ page thứ 0 đến page thứ 13 được allocated và địa chỉ của byte đầu tiên được lưu vào thanh ghi 0.
- alloc 1568 1: page thứ 14 và 15 được allocated và địa chỉ của byte đầu tiên lưu vào thành ghi 1.
- free 0: Lấy địa chỉ từ thanh ghi 0 và deallocate từ page thứ 0 đến page thứ 13.
- alloc 1386 2: page thứ 0 và page thứ 1 được allocated và địa chỉ của byte đầu tiên được lưu vào thanh ghi 2.
- alloc 4564 4: page thứ 2 đến page thứ 6 được allocated và địa chỉ của byte đầu tiên được lưu vào thành ghi 4.
- write 100 1 20: Lấy địa chỉ ở thanh ghi thứ 1 cộng cho 20 (theo thập phân) ta được địa chỉ 0x3814, và ở địa chỉ đó ta ghi 0x64 (là 100 ở hệ thập phân) vào địa chỉ đó.
- write 20 2 1000: Lấy địa chỉ ở thanh ghi thứ 2 cộng cho 1000 (theo thập phân) ta được địa chỉ 0x003e8, và ở địa chỉ đó ta ghi 0x14 (là 20 ở hệ thập phân) vào địa chỉ đó.

2.2.5 Memory test for m1

```
1 8
alloc 13535 0
alloc 1568 1
free 0
alloc 1386 2
alloc 4564 4
free 2
free 4
free 1
```

Sau khi thực hiện thì test m1 không cho ra output nào cả.
Giải thích kết quả m1:

- alloc 13535 0: Từ page thứ 0 đến page thứ 13 được allocated và địa chỉ của byte đầu tiên được lưu vào thanh ghi 0.

- alloc 1568 1: page thứ 14 và 15 được allocated và địa chỉ của byte đầu tiên lưu vào thành ghi 1.
- free 0: Lấy địa chỉ từ thanh ghi 0 và deallocate từ page thứ 0 đến page thứ 13.
- alloc 1386 2: page thứ 0 và page thứ 1 được allocated và địa chỉ của byte đầu tiên được lưu vào thanh ghi 2.
- alloc 4564 4: page thứ 2 đến page thứ 6 được allocated và địa chỉ của byte đầu tiên được lưu vào thành ghi 4.
- free 2: Lấy địa chỉ từ thanh ghi 2 và deallocate page thứ 0 và page thứ 1.
- free 4: Lấy địa chỉ từ thanh ghi 4 và deallocate từ page thứ 2 đến page thứ 6.
- free 1: Lấy địa chỉ từ thanh ghi 1 và deallocate page thứ 14 và page thứ 15.
- Do tất cả các page trong _mem_stat đều đã được deallocated, test m1 không cho ra output nào cả.

2.3 Answer the questions

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

Answer: Không giống như các giải thuật khác sử dụng priority feedback queue (PFQ) cho phép process di chuyển qua lại giữa các hàng đợi. Nếu process có CPU burst time lớn thì nó sẽ chuyển sang hàng đợi có độ ưu tiên thấp hơn và ngược lại process đợi quá lâu trong hàng đợi có độ ưu tiên thấp ở sẽ chuyển sang hàng đợi có độ ưu tiên cao hơn nhằm giảm tình trạng chờ đợi vô hạn định. Do đó PFQ vừa tránh được tình trạng process có thời gian thực thi dài độc chiếm CPU (như trường hợp của FCFS), vừa tránh được tình trạng những process có thời gian thực thi dài phải chờ rất lâu (như trường hợp của SJF, SRTF). PFQ cũng đồng thời hạn chế vấn đề time slice và chuyển process của Round Robin.

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

Answer: Ưu điểm của segmentation with paging:

- Tận dụng được ưu điểm của phân trang: Các đoạn của 1 process được chia thành nhiều trang, và ở bộ nhớ vật lý các trang không cần liên tiếp nhau, do đó khắc phục được phân mảnh ngoại.
- Tận dụng được ưu điểm của phân đoạn: Khắc phục tình trạng nếu một vùng quá lớn thì có thể không nạp nó được vào bộ nhớ, cũng như kích thước các đoạn có thể thay đổi cho chương trình.

Answer: Nhược điểm:

- Nhược điểm của phân trang: các trang có kích thước cố định nên trang cuối cùng thường phải lớn hơn bộ nhớ cần nạp, dẫn đến phân mảnh nội.
- Tốn nhiều chi phí tính toán cho các bảng phân trang và bảng phân đoạn.

2.4 Kết luận

Qua assignment 2 này chúng ta đã biết cách mà hệ điều hành thực hiện định thời CPU, cơ chế hoạt động đồng bộ, làm thế nào để chuyển địa chỉ logic sang địa chỉ vật lý và các giải thuật phân đoạn, phân trang.

Tài liệu tham khảo

[1] VOER - Định thời biểu CPU (http://voer.edu.vn/c/dinh-thoi-bieu-cpu/a22667db/02426301).

[2] VOER - Bộ nhớ ảo (http://voer.edu.vn/c/bo-nho-ao/a039fa79/db59252d).

[3] Slide bài giảng cô Lê Thanh Vân: Scheduling, Memory.