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

BÁO CÁO BÀI TẬP LỚN

HỆ ĐIỀU HÀNH - CO2018

Group: Nhóm OSLab

Assignment #2 - Simple Operating System

GV hướng dẫn: Hoàng Lê Hải Thanh
SV thực hiện: Nguyễn Đức An – 2010102
Trần Phúc Anh – 2010133
Nguyễn Quang Huy – 1916081
Trần Hà Tuấn Kiệt – 2011493

Tp. Hồ Chí Minh, Tháng 6/2022

PHÂN CÔNG CÔNG VIỆC

STT	Họ và tên	MSSV	Problems	% BTL
1	Nguyễn Đức An	2010102	- Scheduler Question - Scheduler Result - Implement mem.c - Put It All Together	100%
2	Trần Phúc Anh	2010133	- Memory Management Question - Memory Management Result	100%
3	Nguyễn Quang Huy	1916081	- Memory Management Question - Implement mem.c - Memory Management Result	100%
4	Trần Hà Tuấn Kiệt	2011493	- Scheduler Question - Implement queue.c, sched.c	100%

Mục lục

1 Scheduler

1.1 Scheduler Question

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

Giới thiệu về Priority Feedback Queue

Giải thuật định thời priority feedback queue là sự kết hợp giữa Multilevel queue và cơ chế feedback. Multilevel queues được sử dụng để lưu các process có cùng độ ưu tiên trên cùng một hàng, ngoài ra giữa các hàng có thể đổi chỗ cho nhau (đây gọi là cơ chế feedback). Đây là giải thuật preemptive (cho phép ngắt process đang chạy để nhường cho một process khác), thể hiện qua cơ chế sử dụng 2 hàng đợi ưu tiên (priority queue) là ready_queue và run_queue và cách CPU vận hành theo kiểu round-robin. Các process được nạp vào ready_queue và chờ được đẩy vào CPU theo thứ tự ưu tiên. Mỗi process được phép chạy trong một khoảng thời gian nhất định. Sau khoảng thời gian đó, process được đẩy sang run_queue. Nếu ready_queue không còn process nào, tất cả process trong run_queue (nếu có) sẽ được đẩy về ready_queue.

Ưu điểm của Priority Feedback Queue

- Priority Feedback Queue sử dụng time slice để quy định thời gian tối đa một process chạy trong một đơn vị thời gian nhất định, có vai trò giống với quantum time trong thuật toán Round Robin, nhằm đảm bảo rằng các process có thể luân phiên nhau được thực thi, tránh được trường hợp starvation.
- Sau khoảng thời gian time_slice, các process trong run_queue sẽ được chuyển về ready_queue, trong trường hợp những process chưa hoàn thành (incompleted process) trong run_queue thì những process đó sẽ được tạm dừng và lưu lại trạng thái để tiếp tục chạy trong những lần sau.
- Những process có độ ưu tiên thấp hơn dù đến sau nhưng vẫn có thể được thực thi luân phiên với các process có độ ưu tiên cao hơn.
- Những process có độ ưu tiên cao hơn vẫn có thể được thực thi trước nếu nó ở trong ready_queue, vẫn đảm bảo được tính chất priority.

So sánh với các thuật toán định thời khác

- **First Come First Serve (FCFS)**
  - Trong thuật toán FCFS, process có thời gian thực thi nhỏ nhưng đến sau sẽ phải đợi các process đến trước, có thể xảy ra trường hợp starvation nếu process đến trước có thời gian thực thi lớn, do đó thời gian đợi của process có thể sẽ lớn, dẫn đến thời gian đợi trung bình lớn.
  - Trong thuật toán PFQ, mỗi process chỉ có thể thực thi trong một khoảng thời gian nhất định (limited quantum time) theo cơ chế Round-Robin, do đó giữa các process có thể chia sẻ thời gian thực thi luân phiên nhau, dẫn đến thời gian đợi trung bình nhỏ hơn.
- **Shortest Job First (SJF)**
  - Trong thuật toán SJF, process có thời gian thực thi (burst time) nhỏ nhất sẽ được thực thi sớm nhất, do đó có thể sẽ xảy ra starvation một process đến trước nếu có burst time lớn phải đợi nhiều process đến sau nhưng có burst time nhỏ hơn.
  - Trong thuật toán PFQ, các process sẽ được luân phiên thực thi với nhau trong các khoảng thời gian nhất định (quantum time).
- **Round Robin (RR)**
  - Trong thuật toán RR, throughput của hàng đợi phụ thuộc vào quantum time. Nếu quantum time đủ lớn, thuật toán RR có performance tương tự với thuật toán FCFS.
  - Trong thuật toán PFQ, mặc dù cơ chế sử dụng quantum time tương tự như Round-Robin, nhưng có thêm độ ưu tiên (priority) để lựa chọn thứ tự process sẽ được thực thi, do đó starvation xảy ra ít hơn.
- **Priority Scheduling (PS)**
  - Trong thuật toán PS, những process có độ ưu tiên nhỏ sẽ phải đợi cho đến khi những process có độ ưu tiên lớn hơn thực thi xong (Preemptive PS). Điều này có thể dẫn đến starvation khi các process có độ ưu tiên nhỏ phải đợi trong một thời gian dài cho đến lượt.
  - Trong thuật toán PFQ, sự chuyển đổi giữa các queues cho phép những process có độ ưu tiên nhỏ hơn có thể được thực thi sớm hơn, không cần phải đợi một khoảng thời gian quá dài, trong khi đó vẫn đảm bảo được thứ tự thực thi theo

độ ưu tiên (priority), những process có độ ưu tiên cao hơn sau khi được thực thi trước và sau khi trở về ready_queue sẽ được lưu lại trạng thái để tiếp tục thực thi trong những lần sau.

- **Multilevel Queue (MLQ)**
  - Trong MLQ, các process chia thành nhiều hàng đợi khác nhau dựa trên độ ưu tiên, mỗi hàng đợi bao gồm các process có cùng độ ưu tiên. Trong quá trình thực thi, tất cả các process trong cùng một hàng đợi sẽ được thực thi cùng một lúc, do đó các process trong hàng đợi có độ ưu tiên thấp phải đợi cho đến khi tất cả các process trong hàng đợi có độ ưu tiên cao hơn hoàn thành, dẫn đến hiện tượng starvation đối với các process có độ ưu tiên nhỏ.
  - Trong PFQ, các process cũng chia thành nhiều hàng đợi khác nhau dựa trên độ ưu tiên, tuy nhiên điểm khác biệt ở đây là cơ chế feedback, cho phép các hàng đợi có thể luân phiên nhau thực thi. Do đó, các process trong hàng đợi có độ ưu tiên thấp không cần phải đợi cho đến khi tất cả các process trong hàng đợi cao hơn hoàn thành, mà có thể luân phiên thực thi sau những khoảng thời gian nhất định (quantum time). Do đó thời gian đợi trung bình sẽ thấp hơn so với MLQ. PFQ là sự kết hợp những ưu điểm của các thuật toán multilevel queue và round-robin, PFQ có thể giúp giảm hiện tượng starvation đối với các process có độ ưu tiên nhỏ.

1.2 Scheduler Implementation

Đầu tiên ta hiện thực các thao tác làm việc trên priority queue. Ở đây nhóm hiện thực priority queue bằng mảng một chiều.

Cấu trúc dữ liệu cho 1 queue được đặc tả tại file queue.h như sau:

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

void enqueue(struct queue_t * q, struct pcb_t * proc);

struct pcb_t * dequeue(struct queue_t * q);

int empty(struct queue_t * q);
```

Hàm empty(struct queue_t*) xác định một queue đang rỗng hay không. Nếu một queue rỗng, hàm trả về giá trị là 1, ngược lại trả về giá trị là 0

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

Đối với file queue.c

Đầu tiên, ta sẽ hiện thực hàm **enqueue**. Hàm enqueue(struct queue_t*, struct pcb_t*) đẩy một PCB vào một queue bằng cách thêm vào cuối mảng và tăng kích thước của queue lên 1. Độ phức tạp tính toán cho thao tác thêm này là O(1)

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

Tiếp theo, ta sẽ hiện thực hàm **dequeue**. Hàm dequeue(struct queue_t*) lấy một PCB có độ ưu tiên cao nhất ra khỏi queue bằng cách duyệt mảng một cách tuần tự và lưu giá trị của độ ưu tiên và vị trí của PCB có độ ưu tiên cao hơn vào hai biến. Hàm trả về địa chỉ của PCB có độ ưu tiên cao nhất.

```
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 (q->size != 0) {
     struct pcb_t *temp_proc = NULL;
     int max_pri = 0;
     int max_idx = 0;
     for (int i = 0; i < q->size; i++) {
        if (q->proc[i]->priority > max_pri) {
           temp_proc = q->proc[i];
           max_pri = temp_proc->priority;
           max_idx = i;
        }
     }
```

```
        for (int j = max_idx; j < q->size - 1; j++) {
            q->proc[j] = q->proc[j + 1];
        }
        q->proc[q->size - 1] = NULL;
        q->size--;
        return temp_proc;
    }
    return NULL;
}
```

Đối với file shed.c

Tiếp theo, ta hiện thực giải thuật hàm **get_proc**. Khi ready_queue rỗng, thì hàm này được dùng để load các process từ run_queue to ready_queue.

```
struct pcb_t *get_proc(void)
{
    struct pcb_t *proc = NULL;
    /*TODO: get a process from [ready_queue]. If ready queue
     * is empty, push all processes in [run_queue] back to
     * [ready_queue] and return the highest priority one.
     * Remember to use lock to protect the queue.
     * */
    pthread_mutex_lock(&queue_lock);
    if (empty(&ready_queue)) {
        for (int i = 0; i < run_queue.size; i++) {
            ready_queue.proc[i] = run_queue.proc[i];
            run_queue.proc[i] = NULL;
        }
        ready_queue.size = run_queue.size;
        run_queue.size = 0;
    }
    proc = dequeue(&ready_queue);
    pthread_mutex_unlock(&queue_lock);
    return proc;
}
```

1.3 Scheduler Result

Requirement: Draw Gantt diagram describing how processes are executed by the CPU.

Giả định: Bởi vì trong thực tế, tại một thời điểm có thể có nhiều threads chạy đồng

thời với nhau (concurrent threads), do đó kết quả từ Gantt chart về mặt lý thuyết có thể có khác biệt nhỏ so với kết quả thực tế.

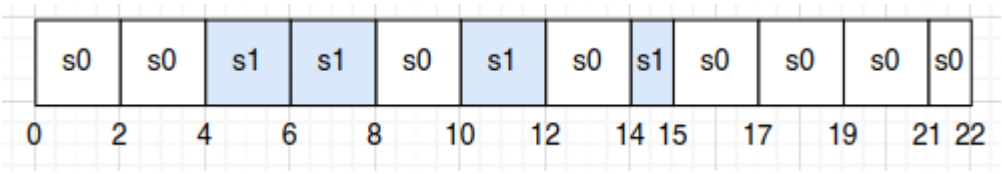
1.3.1 Testcase sched_0

(i) **Input Analysis**

- **Config File**
  - Time slice = 2
  - N = Number of CPU = 1
  - M = Number of Processes to be run = 2
- **Process Info**

Proc	Arrived Time	Priority	Number of Instruction
s0	0	12	15
s1	4	20	7

(ii) **Gantt Diagram**



Hình 1: *Gantt Diagram sched_0*

***Lưu ý:** Một instruction cần 1 time slot để thực thi.

(iii) Output

```
ducan@ducan-HP-Pavilion-Laptop-15-cs3xxx:~/Downloads/Assigment/BTL/source_code$ make test_sched
------ SCHEDULING TEST 0 ----------------------------------------
./os sched_0
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
Time slot   7
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot   8
Time slot   9
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  1
Time slot  10
Time slot  11
        CPU 0: Put process  1 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  1
Time slot  14
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
Time slot  15
Time slot  16
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  1
Time slot  17
Time slot  18
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  19
Time slot  20
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  21
Time slot  22
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  23
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Hình 2: *Output sched_0*

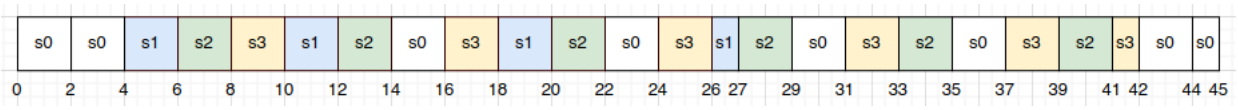
1.3.2 Testcase sched_1

(i) **Input Analysis**

- **Config File**
  - Time slice = 2
  - N = Number of CPU = 1
  - M = Number of Processes to be run = 4
- **Process Info**

Proc	Arrived Time	Priority	Number of Instruction
s0	0	12	15
s1	4	20	7
s2	6	20	12
s3	7	7	11

(ii) **Gantt Diagram**



Hình 3: *Gantt Diagram sched_1*

***Lưu ý:** Một instruction cần 1 time slot để thực thi.

(iii) Output

```
------ 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
```

Hình 4: *Output sched_1 part 1*

```
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
```

Hình 5: *Output sched_1 part 2*

2 Memory Management

2.1 Segmentation

Segmentation là một phương pháp memory management dùng để chia memory thành nhiều phần khác nhau được gọi là `segments` và gán các process vào những segments này. Vì những process có kích thước khác nhau, nên các segments này cũng có kích thước khác nhau.

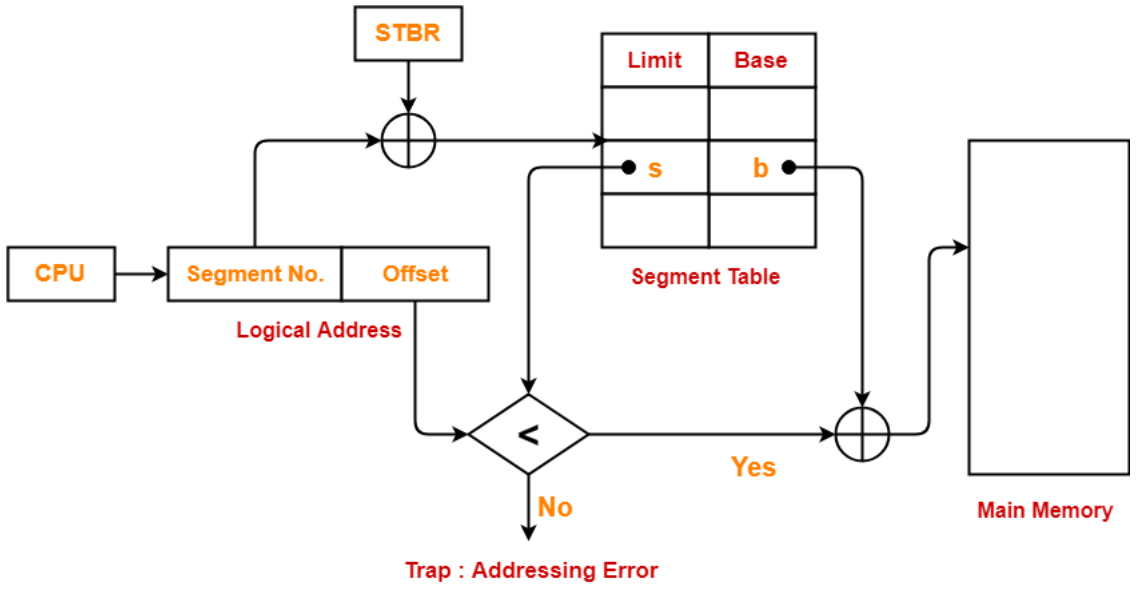
Với mỗi process, phần địa chỉ luận lý `logical address` chứa hai thông tin:

- Segment index
- Offset

Bảng **segment table** có tác dụng map mỗi segment của nó với phần base address tương ứng trong physical memory. Phần base address này sẽ kết hợp với phần offset từ logical address để tạo nên địa chỉ thật `physical address`. Physical address bao gồm hai thông tin:

- The base address
- The length (limit)

Trong đó, giá trị offset sẽ được so sánh với limit; nếu **offset** $>$ **limit**, phần base address sẽ hợp lệ và có thể được chuyển thành physical address; còn nếu không, hệ thống sẽ báo lỗi.



Hình 6: *Segmentation Method*

2.2 Paging

Paging là một phương pháp memory management được dùng để chia physical memory thành những khối **fixed - size** được gọi là `frames` và chia logical memory thành những khối có kích thước bằng với frames gọi là `pages`

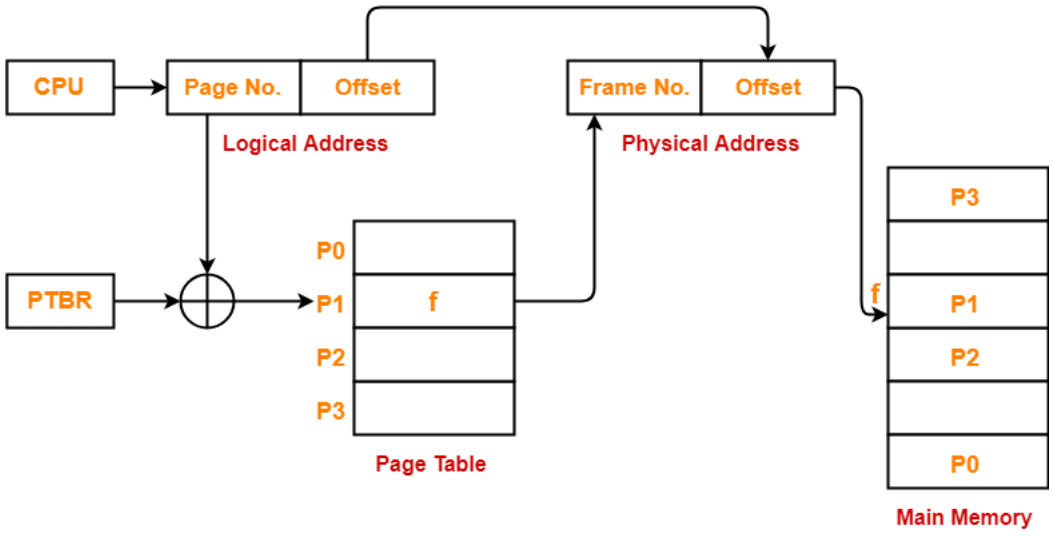
Physical address bao gồm các thông tin:

- Frame index
- Offset

Logical address bao gồm:

- Page Number
- Offset

Bảng **page table** có tác dụng map mỗi page với base address (frame index) tương ứng của nó. Phần base address này sẽ kết hợp với phần offset ở logical address để hình thành physical address.



Hình 7: *Paging Method*

2.3 Segmentation with paging

Segmentation Paging là một phương pháp kết hợp cả Paging và Segmentation. Phần **logical address** sẽ được biểu diễn bởi segment index, page table index và offset. Phần **physical address** sẽ bao gồm physical index và offset.

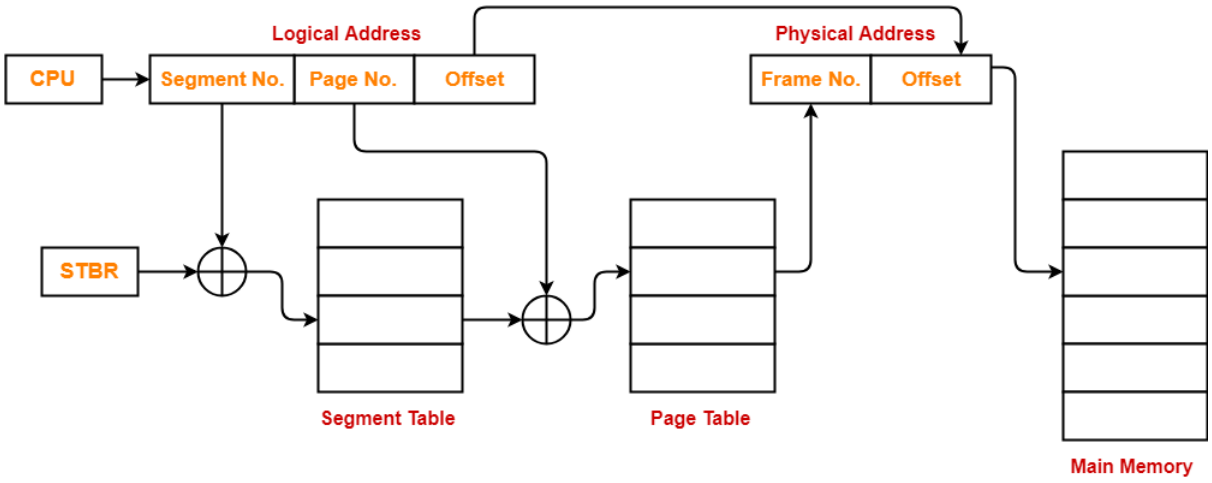
Mỗi process sẽ được chia thành nhiều segments được gọi là các `entries` trong bảng segment table:

- Mỗi entry cung cấp một segment index và một pointer trỏ đến một page table
- Kích thước của page table trong segment là có giới hạn

Trong bảng **segment table**, page table được xác định bằng segment pointer và dùng page index để xác định page tương ứng. Theo đó, page table đã xác định bao gồm số entries đại diện cho số page được phân chia từ một segment:

- Mỗi entry cung cấp một page index và một physical index trong bộ nhớ chính
- Vì một process được chia thành các segments, một process sẽ có thể có nhiều page tables.

Với **page table**, physical address sẽ được kết hợp với page offset để hình thành physical address (tương tự với cơ chế paging)



Hình 8: *Segmentation with paging Method*

2.4 Memory Mangement Question

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

Ưu điểm:

- Tránh được phân mảnh ngoại **external segmentation**, vì các pages đều có kích thước bằng nhau.
- Tối ưu thời gian chuyển đổi. Việc lấy page table index và physical address sẽ nhanh hơn so với việc tích hợp tất cả các entries của một page table lớn.
- Khắc phục size của page quá lớn nhờ paging trong từng segment

- Việc cấp phát vùng nhớ sẽ dễ dàng hơn do kế thừa các tính chất của cơ chế paging

Khuyết điểm:

- Vẫn xuất hiện tình trạng phân mảnh nội **internal segmentation** vì cơ chế paging
- Size của process bị giới hạn bởi size của physical memory
- Khó có thể bảo trì nhiều process cùng lúc trong memory, do vậy khó nâng cao multiprogramming của OS
- Nếu segment table không được sử dụng thì không cần lưu page table. Page table cần nhiều memory space, nên sẽ không tốt cho system có RAM nhỏ
- Độ phức tạp cho việc thực thi cao hơn nhiều so với segmentation và paging mechanism, cần đến hardware support.

2.5 Memory Mangement Implementation

Đối với file mem.c

Đầu tiên, ta hiện thực hàm **get_page_table**. Hàm get_page_table sẽ nhận 2 tham số là segment table và index, sau đó trả về page table tương ứng với các tham số nhận vào.

```
/* Search for page table table from the a segment table */
static struct page_table_t * get_page_table(
    addr_t index,  // Segment level index
    struct seg_table_t * seg_table) { // first level table

  /*
   * TODO: Given the Segment index [index], you must go through each
   * row of the segment table [seg_table] and check if the v_index
   * field of the row is equal to the index
   *
   * */

  int i;
  for (i = 0; i < seg_table->size; i++) {
     if (seg_table->table[i].v_index == index) {
        return seg_table->table[i].pages; // Return the page table from the
            segment index
     }
  }
```

```
    return NULL;
}
```

Có thể thấy khá cụ thể ở đây, logic của hàm **get_page_table** là nó sẽ lặp qua từng index trong segment table, sau đó chọn ra và trả về page table có index tương ứng với tham số index nhận vào. Nếu không tìm được page table có index phù hợp, hàm sẽ trả về NULL.

Tiếp đến, ta hiện thực hàm **translate**. Hàm translate sẽ nhận các tham số là process và địa chỉ ảo của process đó, sau đó lưu địa chỉ vật lý đã được chuyển đổi vào 1 tham số khác.

```
static int translate(
    addr_t virtual_addr,  // Given virtual address
    addr_t * physical_addr, // Physical address to be returned
    struct pcb_t * proc) { // Process uses given virtual address
    /* Offset of the virtual address */
    addr_t offset = get_offset(virtual_addr); // Offset <-> 10 bits
    /* The first layer index */
    addr_t first_lv = get_first_lv(virtual_addr); // Segment index <-> 5 bits
    /* The second layer index */
    addr_t second_lv = get_second_lv(virtual_addr); // Page index <-> 5 bits

    /* Search in the first level */
    struct page_table_t * page_table = NULL;
    page_table = get_page_table(first_lv, proc->seg_table);
    /*^ By using the first_lvl which is the segment index, we can find the
        page table
       corresponding to it and then use the page table for the later process*/
    if (page_table == NULL) {
       return 0;
    }

    int i;
    for (i = 0; i < page_table->size; i++) {
       if (page_table->table[i].v_index == second_lv) {

          /* The second_lvl is the page index, that will used along with page
              base number
             to find the correct frame number in page table of the
                corresponding segment */
```

```
        /* In short: It it used to find the frame number in page table */

        /* 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] */

        addr_t physical_index = page_table->table[i].p_index;
        * physical_addr = (physical_index << OFFSET_LEN) | offset; //
            Concatenate and save to p_addr
        return 1;
      }
    }
    return 0;
  }
```

Đầu tiên, ta xác định các giá trị offset, segment index và page index từ địa chỉ ảo. Sau đó, từ segment index, ta gọi hàm get_page_table để lấy ra page table phù hợp. Tiếp theo, ta dựa vào page index để lặp và chọn ra frame phù hợp trong page table để thực hiện ánh xạ, sau khi xác định được frame, ta dựa vào 10bit offset cuối để biết cụ thể bytes mình cần ánh xạ nằm ở vị trí nào của frame đó. Cuối cùng, ta lưu địa chỉ vật lý đã được chuyển đổi vào tham số physical address.

Tiếp theo trong file **mem.c**, ta hiện thực 2 hàm **alloc_mem** để mô phỏng việc cấp phát và hàm **free_mem** để mô phỏng việc giải phòng vùng nhớ.

```
addr_t alloc_mem(uint32_t size, struct pcb_t * proc) {
  pthread_mutex_lock(&mem_lock);
  addr_t ret_mem = 0;
  /* TODO: Allocate [size] byte in the memory for the
   * process [proc] and save the address of the first
   * byte in the allocated memory region to [ret_mem].
   * */

  uint32_t num_pages = (size % PAGE_SIZE) ? size / PAGE_SIZE + 1 :
    size / PAGE_SIZE; // Number of pages we will use pagesize
  int mem_avail = 0; // We could allocate new memory region or not?
  /* First we must check if the amount of free memory in
   * virtual address space and physical address space is
   * large enough to represent the amount of required
```

```
     * memory. If so, set 1 to [mem_avail].
     * Hint: check [proc] bit in each page of _mem_stat
     * to know whether this page has been used by a process.
     * For virtual memory space, check bp (break pointer).
     * */
    //Check physical
    int phy_free_pages = 0;
    for (int i = 0; i < NUM_PAGES; i++)
    {
       if (_mem_stat[i].proc == 0) phy_free_pages++;
    }
    mem_avail = phy_free_pages >= num_pages;

    if (mem_avail) {
       /* We could allocate new memory region to the process */
       ret_mem = proc->bp;
       int num_seg_entries = num_pages % (1 << PAGE_LEN) ? num_pages / (1 <<
           PAGE_LEN) + 1 : num_pages / (1 << PAGE_LEN);
       proc->bp += num_seg_entries * (1 << PAGE_LEN) * PAGE_SIZE; //offset n
           segment
       /* Update status of physical pages which will be allocated
        * to [proc] in _mem_stat. Tasks to do:
        *    - Update [proc], [index], and [next] field
        *    - Add entries to segment table page tables of [proc]
        *      to ensure accesses to allocated memory slot is
        *      valid. */
       int curr_page = 0;
       int prev_mem_index = -1;
       int phy_index = 0;
       int flag = 0;
       for (int i = 0; i < NUM_PAGES; i++)
       {
          if (_mem_stat[i].proc == 0)
          {
             //Update [proc], [index], and [next] field
             if (flag == 0)
             {
                phy_index = i;
                flag = 1;
             }
```

```
            _mem_stat[i].proc = proc->pid;
            _mem_stat[i].index = curr_page;
            if (prev_mem_index != -1)
            {
               _mem_stat[prev_mem_index].next = i;
            }
            prev_mem_index = i;
            curr_page++;
            if (curr_page == num_pages)
            {
               _mem_stat[i].next = -1;
               break;
            }
         }
      }

      for (int seg_idx = 0; seg_idx < num_seg_entries; seg_idx++)
      {
         proc->seg_table->table[proc->seg_table->size].v_index =
             get_first_lv(ret_mem + seg_idx * (1 << PAGE_LEN) * PAGE_SIZE);
         proc->seg_table->table[proc->seg_table->size].pages = (struct
             page_table_t *)malloc(sizeof(struct page_table_t));
         proc->seg_table->table[proc->seg_table->size].pages->size = 0;
         proc->seg_table->size++;
      }

      curr_page = 0;
      for (int i = phy_index; i != -1; i = _mem_stat[i].next)
      {
         //Add entries to segment table page tables of [proc]
         addr_t cur_vir_addr = ret_mem + curr_page * PAGE_SIZE;
         struct page_table_t *located_page_table =
             get_page_table(get_first_lv(cur_vir_addr), proc->seg_table);
         located_page_table->table[located_page_table->size].v_index =
             get_second_lv(cur_vir_addr);
         located_page_table->table[located_page_table->size].p_index = i;
         located_page_table->size++;
         curr_page++;
      }
```

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

Hàm alloc_mem sẽ nhận 2 tham số là process và dung lượng mà process đó yêu cầu, sau đó trả về địa chỉ bytes đầu tiên của vùng nhớ đã được cấp phát (qua biến ret_name). Để hiện thực điều này, trước hết ta cần tính xem cần bao nhiêu page table cần cho việc cấp phát, nếu số lượng trang cần lớn hơn số trang hiện tại đang trống, việc cấp phát sẽ thất bại và hàm sẽ trả về biến ret_name = 0. Nếu số trang còn lại là đủ cho việc cấp phát, việc cấp phát sẽ được thực hiện. Bước đầu tiên cần thực hiện trong việc cấp phát đó là ta sẽ lặp qua từng ô mem_stat đại diện cho từng trang trong vùng nhớ vật lý, tìm ra những trang còn trống (có thể không liên tiếp nhau) sau đó cập nhật PID, index của những ô sẽ được dùng cấp phát cho process. Sau khi đánh dấu các trang ở vùng nhớ vật lý, ta cấp phát và tạo ra các page table ở vùng nhớ ảo, sau đó thực hiện ánh xạ địa chỉ từ vùng nhớ ảo đến từng ô vùng nhớ thật thông qua vòng lặp.

Cuối cùng, ta cần hiện thực hàm giải phóng vùng nhớ **free_mem**.

```
int free_mem(addr_t address, struct pcb_t * proc) {
  /*TODO: Release memory region allocated by [proc]. The first byte of
    this region is indicated by [address]. Task to do:
   *    - Set flag [proc] of physical page use by the memory block
   *      back to zero to indicate that it is free.
   *    - Remove unused entries in segment table and page tables of
   *      the process [proc].
   *    - Remember to use lock to protect the memory from other
   *      processes. */

  pthread_mutex_lock(&mem_lock);
  addr_t phy_addr;
  //printf("ret_mem free: %d\n", get_second_lv(address));
  if (translate(address, &phy_addr, proc) == 0)
  {
     //printf("ret_mem free: %d\n", get_second_lv(address));
     pthread_mutex_unlock(&mem_lock);
     return 0;
  }

  int num_pages = 0;
```

```
for (int i = phy_addr >> OFFSET_LEN; i != -1; i = _mem_stat[i].next)
{
   _mem_stat[i].proc = 0;
   num_pages++;
}

int num_seg_entries = num_pages % (1 << PAGE_LEN) ? num_pages / (1 <<
    PAGE_LEN) + 1 : num_pages / (1 << PAGE_LEN);
if (address + num_seg_entries * (1 << PAGE_LEN) * PAGE_SIZE == proc->bp)
   proc->bp = proc->bp - num_seg_entries * (1 << PAGE_LEN) * PAGE_SIZE;

int curr_seg_entry = 0;
while (curr_seg_entry < num_seg_entries)
{
   int vir_first_lv = get_first_lv(address);
   struct page_table_t *page_table = get_page_table(vir_first_lv,
       proc->seg_table);
   free(page_table);
   for (int i = 0; i < proc->seg_table->size; i++)
   {
      if (proc->seg_table->table[i].v_index == vir_first_lv)
      {
         for (int j = i; j < proc->seg_table->size - 1; j++)
         {
            proc->seg_table->table[j].pages = proc->seg_table->table[j +
                1].pages;
            proc->seg_table->table[j].v_index = proc->seg_table->table[j +
                1].v_index;
         }

         proc->seg_table->table[proc->seg_table->size - 1].pages = NULL;
         proc->seg_table->size--;
         break;
      }
   }

   curr_seg_entry++;
   address = address + curr_seg_entry * (1 << PAGE_LEN) * PAGE_SIZE;
}
```

```
    pthread_mutex_unlock(&mem_lock);

    return 0;
}
```

Hàm free_mem sẽ nhận 2 tham số là process và địa chỉ bytes đầu tiên của vùng nhớ được cấp phát cho process đó. Sau đó, dựa vào 2 tham số trên ta sẽ giải phóng các vùng nhớ vật lý, vùng nhớ ảo và cả các entries của segment table dùng để cấp phát cho process này. Để làm được vậy, đầu tiên ta cần chuyển đổi địa chỉ ảo của process sang địa chỉ vật lý qua hàm translate, sau đó đi tới địa chỉ vật lý này để free các ô vùng nhớ vật lý(mem_stat). Sau khi free vùng nhớ vật lý, ta đi đến từng entries của segment table để free các frame của từng page table đã sử dụng. Cuối cùng, ta free các page table không dùng tới nữa và các segment table entries chứa những page table đó.

2.6 Memory Mangement Result

Requirement: Show the status of RAM after each memory allocation and deallocation function call.

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

```

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

Hình 9: *Input m0 và m1*

```
ducan@ducan-HP-Pavilion-Laptop-15-cs3xxx:~/Downloads/Assigment/BTL/source_code$ make test_mem
------ MEMORY MANAGEMENT TEST 0 ----------------------------------
./mem input/proc/m0
000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
        003e8: 15
001: 00400-007ff - PID: 01 (idx 001, nxt: -01)
002: 00800-00bff - PID: 01 (idx 000, nxt: 003)
003: 00c00-00fff - PID: 01 (idx 001, nxt: 004)
004: 01000-013ff - PID: 01 (idx 002, nxt: 005)
005: 01400-017ff - PID: 01 (idx 003, nxt: 006)
006: 01800-01bff - PID: 01 (idx 004, nxt: -01)
014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
        03814: 66
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
NOTE: Read file output/m0 to verify your result
------ MEMORY MANAGEMENT TEST 1 ----------------------------------
./mem input/proc/m1
NOTE: Read file output/m1 to verify your result (your implementation should print nothing)
```

Hình 10: *Output*

Giải thích:

Hệ thống mà chúng ta đang mô phỏng có địa chỉ kiến trúc là 20-bit, với 10 bits cuối là offset. Do đó, kích thước trang sẽ là $2^{10} = 1024$ bytes, và số lượng trang trong RAM là $2^{20}/2^{10} = 1024$ trang.

Output sẽ có cấu trúc như sau:

Page Index: start_addr - end_addr - PID: pid (idx: index, nxt: next).

Trong đó:

- Page Index: index của Page trong RAM (vùng nhớ vật lý).
- start_addr: địa chỉ bắt đầu trong vùng nhớ vật lý (theo hệ thập lục phân).
- end_addr: địa chỉ kết thúc trong vùng nhớ vật lý (theo hệ thập lục phân).
- pid: pid của process được cấp phát.
- index: index của Page trong vùng nhớ ảo.
- next: index của vùng nhớ tiếp theo dùng để cấp phát cho process.

Sau khi đã hiểu rõ cấu trúc output, ta tiến hành giải thích output:

Testcase m0:

- **alloc 13535 0**: Lệnh này sẽ cấp phát 13535 bytes, tương ứng 14 trang (từ trang 0 đến trang 13) vì ceil(13535/1024) = 14 và lưu địa chỉ trang đầu tiên vào thanh ghi 0.

- **alloc 1568 1**: Tương tự như lệnh trên, lệnh này yêu cầu cấp phát 1568 bytes tương ứng 2 trang (trang 14, trang 15) và lưu địa chỉ trang đầu tiên vào thanh ghi 1.
- **free 0**: Lệnh này sẽ giải phóng vùng nhớ được giữ bởi thanh ghi 0, tức là 14 trang từ trang 0 đến trang 13 trong trường hợp này.
- **alloc 1386 2**: Lệnh này sẽ cấp phát 1386 bytes tương ứng 2 trang (trang 0 và 1) và lưu địa chỉ trang đầu tiên vào thanh ghi 2.
- **alloc 4564 4**: Lệnh này sẽ cấp phát 4564 bytes tương ứng 5 trang (từ trang 2 đến trang 6) và lưu địa chỉ trang đầu tiên vào thanh ghi 4.
- **write 102 1 20**: Quá trình m0 ghi 102 (66 ở định dạng hex) vào bộ nhớ có địa chỉ vật lý được tính như sau: $physical\_address = base\_address + offset$
  - base_address: địa chỉ vật lý của byte đầu tiên được giữ bởi thanh ghi thứ nhất, có địa chỉ 0x03800
  - offset: 20 (14 ở dạng hex)

  Ở trường hợp này, địa chỉ vật lý là 0x0381
- **write 21 2 1000**: Quá trình m0 ghi 21 (15 ở định dạng hex) vào bộ nhớ có địa chỉ vật lý 00000 (0x00000) + 1000 (0x3e8) = 0x3e8.

Testcase m1:
Testcase m1 tương tự testcase m0, nhưng vì 3 lệnh cuối đã free hết vùng nhớ trước đó đã cấp phát (giữ bởi các thanh ghi 2,4,1) nên output của testcase này trống (vùng nhớ cấp phát được lưu ở thanh ghi 0 đã được giải phóng trước đó giống như ở testcase m0).

3 Put It All Together

3.1 Synchronization

Chúng ta sử dụng 2 mutex locks để bảo vệ cho queue và memory trong quá trình tương tác với các process. Cơ chế synchronization sử dụng mutex lock đảm bảo duy nhất tại một thời điểm chỉ có duy nhất 1 process truy cập đến vùng nhớ dùng chung (shared memory region)

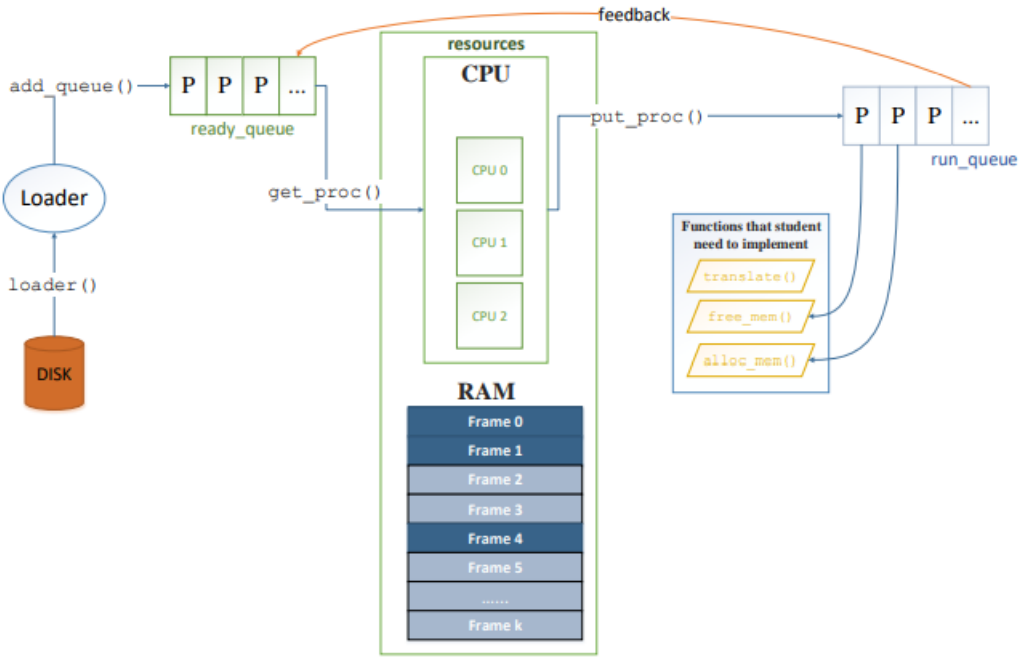
- **queue_lock**: dùng để bảo vệ các hàng đợi (queue) trong quá trình chuyển các process từ run_queue to ready_queue.
- **mem_lock**: dùng để bảo vệ các vùng nhớ (memory) trong quá trình allocate, free, read, write vào memory.

3.2 Interpret Result

Requirement: Student find their own way to interpret the results of simulation.

Giả định: Bởi vì trong thực tế, các process chạy đồng thời với nhau, do đó các kết quả có thể sẽ có khác biệt nhỏ về thứ tự phụ thuộc vào thứ tự thực thi của các process.

3.2.1 Overall



Hình 11: *Model of Assignment*

Giải thích mô hình Simple OS

- Đầu tiên, OS load config file để xác định time slice, số lượng CPU, số lượng process cần thực thi.
- Bước thứ hai, OS load các process đó vào priority feedback queue (bao gồm ready_queue và run_queue) theo thuật toán định thời phù hợp.
- Tiếp theo, những process sẽ được lần lượt load vào CPU dựa vào độ ưu tiên. Mỗi process bao gồm nhiều instructions cần thực thi, CPU thực thi mỗi instruction trong 1 time slot.
- Ngoài ra, một số instructions của process yêu cầu phải được cấp phát bộ nhớ, do đó ta cũng cần phải xây dựng thuật toán cấp phát cho bộ nhớ cho Memory, phương pháp cấp phát bộ nhớ được sử dụng là Segmentation with Paging.

***Lưu ý:** Output của testcase bao gồm hai thông tin, đầu tiên là scheduling log cho từng time slot, thứ hai là memory status sau khi process finish.

3.2.2 Testcase os_0

(i) **Input Analysis**

- **Config File**
  - Time slice = 6
  - N = Number of CPU = 2
  - M = Number of Processes to be run = 4
- **Process Info**

Proc	Arrived Time	Priority	Number of Instruction
p0	0	1	10
p1	4	1	10

(ii) **Output**

```
----- OS TEST 0 -------------------------------------------------
./os os_0
Time slot   0
        Loaded a process at input/proc/p0, PID: 1
Time slot   1
        CPU 1: Dispatched process  1
        Loaded a process at input/proc/p1, PID: 2
Time slot   2
Time slot   3
        CPU 0: Dispatched process  2
        Loaded a process at input/proc/p1, PID: 3
Time slot   4
        Loaded a process at input/proc/p1, PID: 4
Time slot   5
Time slot   6
Time slot   7
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  3
Time slot   8
Time slot   9
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  4
Time slot  10
Time slot  11
Time slot  12
Time slot  13
        CPU 1: Put process  3 to run queue
        CPU 1: Dispatched process  1
Time slot  14
Time slot  15
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  2
Time slot  16
Time slot  17
        CPU 1: Processed  1 has finished
        CPU 1: Dispatched process  3
Time slot  18
Time slot  19
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  4
Time slot  20
Time slot  21
        CPU 1: Processed  3 has finished
        CPU 1 stopped
Time slot  22
Time slot  23
        CPU 0: Processed  4 has finished
        CPU 0 stopped
```

Hình 12: *Output os0 part Scheduler*

```
MEMORY CONTENT:
000: 00000-003ff - PID: 02 (idx 000, nxt: 001)
001: 00400-007ff - PID: 02 (idx 001, nxt: 007)
002: 00800-00bff - PID: 03 (idx 000, nxt: 003)
003: 00c00-00fff - PID: 03 (idx 001, nxt: 004)
004: 01000-013ff - PID: 03 (idx 002, nxt: 005)
005: 01400-017ff - PID: 03 (idx 003, nxt: -01)
006: 01800-01bff - PID: 04 (idx 000, nxt: 011)
007: 01c00-01fff - PID: 02 (idx 002, nxt: 008)
        01de7: 0a
008: 02000-023ff - PID: 02 (idx 003, nxt: 009)
009: 02400-027ff - PID: 02 (idx 004, nxt: -01)
010: 02800-02bff - PID: 01 (idx 000, nxt: -01)
        02814: 64
011: 02c00-02fff - PID: 04 (idx 001, nxt: 012)
012: 03000-033ff - PID: 04 (idx 002, nxt: 013)
013: 03400-037ff - PID: 04 (idx 003, nxt: -01)
015: 03c00-03fff - PID: 03 (idx 000, nxt: 016)
016: 04000-043ff - PID: 03 (idx 001, nxt: 017)
017: 04400-047ff - PID: 03 (idx 002, nxt: 018)
        045e7: 0a
018: 04800-04bff - PID: 03 (idx 003, nxt: 019)
019: 04c00-04fff - PID: 03 (idx 004, nxt: -01)
020: 05000-053ff - PID: 04 (idx 000, nxt: 021)
021: 05400-057ff - PID: 04 (idx 001, nxt: 022)
022: 05800-05bff - PID: 04 (idx 002, nxt: 023)
        059e7: 0a
023: 05c00-05fff - PID: 04 (idx 003, nxt: 024)
024: 06000-063ff - PID: 04 (idx 004, nxt: -01)
033: 08400-087ff - PID: 02 (idx 000, nxt: 034)
034: 08800-08bff - PID: 02 (idx 001, nxt: 035)
035: 08c00-08fff - PID: 02 (idx 002, nxt: 036)
036: 09000-093ff - PID: 02 (idx 003, nxt: -01)
```

Hình 13: *Output os0 part Memory Management*

(iii) **Đánh giá**

- Ở phần Scheduler, thực tế chỉ có 2 process, nhưng trong config file mô tả có đến 4 process cần được thực thi. Do đó, OS load process p0 1 time với PID = 1, load process p1 3 times với PID = 2,3,4.
- Ở phần Memory, hàm write ghi vào memory 3 giá trị 0a, và 1 giá trị 64, chứng tỏ p1 được thực thi 3 lần, p0 được thực thi 1 lần.

3.2.3 Testcase os_1

(i) **Input Analysis**

- **Config File**
  - Time slice = 2
  - N = Number of CPU = 4
  - M = Number of Processes to be run = 8
- **Process Info**

Proc	Arrived Time	Priority	Number of Instruction
p0	1	1	10
s3	2	7	11
m1	4	1	8
s2	6	20	12
m0	7	1	7
p1	9	1	10
s0	11	12	15
s1	16	20	7

(ii) **Output**

```
----- OS TEST 1 -------------------------------------------------
./os os_1
Time slot   0
        Loaded a process at input/proc/p0, PID: 1
Time slot   1
        CPU 1: Dispatched process  1
Time slot   2
        Loaded a process at input/proc/s3, PID: 2
        CPU 3: Dispatched process  2
Time slot   3
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  1
        Loaded a process at input/proc/m1, PID: 3
        CPU 2: Dispatched process  3
Time slot   4
        CPU 3: Put process  2 to run queue
        CPU 3: Dispatched process  2
Time slot   5
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  1
        Loaded a process at input/proc/s2, PID: 4
        CPU 2: Put process  3 to run queue
        CPU 2: Dispatched process  3
Time slot   6
        CPU 0: Dispatched process  4
        CPU 3: Put process  2 to run queue
        Loaded a process at input/proc/m0, PID: 5
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  5
Time slot   7
        CPU 3: Dispatched process  2
        CPU 2: Put process  3 to run queue
        CPU 0: Put process  4 to run queue
Time slot   8
        CPU 2: Dispatched process  1
        CPU 0: Dispatched process  3
        Loaded a process at input/proc/p1, PID: 6
        CPU 3: Put process  2 to run queue
        CPU 3: Dispatched process  6
Time slot   9
        CPU 1: Put process  5 to run queue
        CPU 1: Dispatched process  4
        CPU 2: Put process  1 to run queue
        CPU 2: Dispatched process  2
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  5
```

Hình 14: *Output os1 Scheduler part 1*

```
Time slot  10
        Loaded a process at input/proc/s0, PID: 7
        CPU 3: Put process  6 to run queue
        CPU 1: Put process  4 to run queue
        CPU 1: Dispatched process  4
Time slot  11
        CPU 3: Dispatched process  7
Time slot  12
        CPU 0: Put process  5 to run queue
        CPU 0: Dispatched process  1
        CPU 2: Put process  2 to run queue
        CPU 2: Dispatched process  3
        CPU 3: Put process  7 to run queue
        CPU 1: Put process  4 to run queue
        CPU 1: Dispatched process  4
        CPU 3: Dispatched process  6
Time slot  13
Time slot  14
        CPU 2: Processed  3 has finished
        CPU 2: Dispatched process  7
        CPU 0: Processed  1 has finished
        CPU 0: Dispatched process  2
        CPU 3: Put process  6 to run queue
        CPU 3: Dispatched process  5
        CPU 1: Put process  4 to run queue
        CPU 1: Dispatched process  4
Time slot  15
        Loaded a process at input/proc/s1, PID: 8
        CPU 2: Put process  7 to run queue
        CPU 2: Dispatched process  8
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  6
Time slot  16
Time slot  17
        CPU 1: Put process  4 to run queue
        CPU 1: Dispatched process  4
        CPU 3: Put process  5 to run queue
        CPU 3: Dispatched process  7
        CPU 2: Put process  8 to run queue
        CPU 0: Put process  6 to run queue
        CPU 0: Dispatched process  8
Time slot  18
        CPU 2: Dispatched process  2
        CPU 3: Put process  7 to run queue
```

Hình 15: *Output os1 Scheduler part 2*

```
Time slot  19
        CPU 3: Dispatched process  5
        CPU 1: Processed  4 has finished
        CPU 1: Dispatched process  6
        CPU 2: Processed  2 has finished
        CPU 2: Dispatched process  7
        CPU 3: Processed  5 has finished
        CPU 3 stopped
Time slot  20
        CPU 0: Put process  8 to run queue
        CPU 0: Dispatched process  8
Time slot  21
        CPU 1: Put process  6 to run queue
        CPU 1: Dispatched process  6
        CPU 2: Put process  7 to run queue
        CPU 2: Dispatched process  7
Time slot  22
        CPU 0: Put process  8 to run queue
        CPU 0: Dispatched process  8
Time slot  23
        CPU 2: Put process  7 to run queue
        CPU 2: Dispatched process  7
        CPU 0: Processed  8 has finished
        CPU 0 stopped
        CPU 1: Processed  6 has finished
        CPU 1 stopped
Time slot  24
Time slot  25
        CPU 2: Put process  7 to run queue
        CPU 2: Dispatched process  7
Time slot  26
Time slot  27
        CPU 2: Put process  7 to run queue
        CPU 2: Dispatched process  7
Time slot  28
        CPU 2: Processed  7 has finished
        CPU 2 stopped
```

Hình 16: *Output os1 Scheduler part 3*

```
MEMORY CONTENT:
000: 00000-003ff - PID: 06 (idx 000, nxt: 001)
001: 00400-007ff - PID: 06 (idx 001, nxt: 031)
002: 00800-00bff - PID: 05 (idx 000, nxt: 003)
003: 00c00-00fff - PID: 05 (idx 001, nxt: 004)
004: 01000-013ff - PID: 05 (idx 002, nxt: 005)
005: 01400-017ff - PID: 05 (idx 003, nxt: 006)
006: 01800-01bff - PID: 05 (idx 004, nxt: -01)
007: 01c00-01fff - PID: 05 (idx 000, nxt: 008)
        01fe8: 15
008: 02000-023ff - PID: 05 (idx 001, nxt: -01)
013: 03400-037ff - PID: 06 (idx 000, nxt: 014)
014: 03800-03bff - PID: 06 (idx 001, nxt: 015)
015: 03c00-03fff - PID: 06 (idx 002, nxt: 016)
016: 04000-043ff - PID: 06 (idx 003, nxt: -01)
021: 05400-057ff - PID: 01 (idx 000, nxt: -01)
        05414: 64
024: 06000-063ff - PID: 05 (idx 000, nxt: 025)
        06014: 66
025: 06400-067ff - PID: 05 (idx 001, nxt: -01)
031: 07c00-07fff - PID: 06 (idx 002, nxt: 032)
        07de7: 0a
032: 08000-083ff - PID: 06 (idx 003, nxt: 033)
033: 08400-087ff - PID: 06 (idx 004, nxt: -01)
```

Hình 17: *Output os1 part Memory Management*

(iii) **Đánh giá**

– Trong testcase này, chúng ta có 8 process đến, và cũng có 8 process cần được thực thi, do đó các process đều được thực thi bởi 4 CPU.

4 Tài liệu tham khảo

1. Avi & Peter & Greg. (2018). *Operating system concepts 10th edition*

2. Remzi & Andrea. (2014). *Operating systems: Three easy pieces*