REPORT

BÀI TẬP LỚN - SIMPLE OPERATING SYSTEM

MÔN HỌC: HỆ ĐIỀU HÀNH

10/04 - 29/05/2023, TP.HỒ CHÍ MINH, VIỆT NAM

NGUYỄN ĐỨC AN – 2112737
VÕ TẤN HƯNG – 2113623
LÊ ĐÌNH HUY – 2113481
ĐÀO DUY LONG - 2113928

AGRI TERRA

HCMC University of Technology

Block A3, 268 Ly Thuong Kiet
Ward 14, District 10
HCM City
T +84 28 3864 7256

Faculty of Computer Science and Engineering,
Department of Systems and Networkings

Operating System

Simple Operating System

Assignment Report

Instructor: Nguyen Phuong Duy
Authors: Nguyen Duc An – 2112737
Vo Tan Hung – 2113623
Le Dinh Huy – 2113481
Dao Duy Long - 2113928

Mục Lục

SUMMARY

Trong bài tập lớn phần môn Hệ điều hành này, nhóm chúng em đã hoàn thành mô phỏng một hệ điều hành đơn giản với các hoạt động: Scheduling (định thời) với giải thuật Multilevel Queue, memory management (quản lý bộ nhớ) cũng như là synchronization (đồng bộ hóa).

Sau khi hoàn thành các phần, nhóm chúng em đã phần nào hiểu được các nguyên lý, nguyên tắc cơ bản của một hệ điều hành đơn giản. Từ đó, có thể nắm được khái quát nội dung được học trên lớp lý thuyết cũng như học phần thí nghiệm thí nghiệm để áp dụng vào thực tiễn.

Author(s)

NGUYỄN ĐỨC AN – 2112737 – Sinh viên năm 2 ngành Khoa học Máy tính
VÕ TẤN HƯNG – 2113623 – Sinh viên năm 2 ngành Khoa học Máy tính
LÊ ĐÌNH HUY – 2113481 – Sinh viên năm 2 ngành Khoa học Máy tính
ĐÀO DUY LONG – 2113928 – Sinh viên năm 2 ngành Khoa học Máy tính

1 INTRODUCTION TO THE ASSIGNMENT

1.1 General

1.1.1 Mục tiêu

Bài tập lớn giúp cho sinh viên giả lập các thành phần lớn trong một hệ điều hành đơn giản. Ví dụ: định thời (scheduler), đồng bộ (synchronization), quan hệ giữa bộ nhớ vật lý (physical memory) và bộ nhớ ảo (virtual memory).

1.1.2 Nội dung

Sinh viên sẽ hiện thực 3 mô-đun chính: định thời, đồng bộ, cơ chế (mechanism) của cấp phát (allocation) bộ nhớ từ bộ nhớ ảo sang vật lý.

1.1.3 Kết quả đạt được

Sau bài tập lớn này, sinh viên hiểu được phần nào đó nguyên lý hoạt động của một hệ điều hành cơ bản. Sinh viên có thể hiểu vai trò và ý nghĩa của các mô-đun (key modules) ở hệ điều hành cũng như nó hoạt động như thế nào.

1.2 Overview

Bài tập lớn về việc mô phỏng một hệ điều hành đơn giản để giúp sinh viên hiểu được khái niệm nền tảng của định thời, đồng bộ và sự quản lý bộ nhớ. Hình 1 thể hiện kiến trúc tổng quát của hệ điều hành mà chúng ta chuẩn bị hiện thực. Nói chung, hệ điều hành phải quản lý 2 nguồn ảo: CPU(s) và RAM sử dụng 2 thành phần cốt lõi:

- Định thời (Scheduler) và điều phối (Dispatcher): xác định tiến trình nào được cho phép chạy trên CPU
- Bộ nhớ ảo (VME): cô lập khoảng không gian bộ nhớ của mỗi tiến trình với nhau. Mặc dù RAM được chia sẻ bởi nhiều tiến trình (multi processes) nhưng mỗi tiến trình đều không biết sự tồn tại của nhau. Điều đó cho phép mỗi tiến trình ở hữu không gian bộ nhớ ảo riêng cho mình và bộ nhớ ảo sẽ kết nối và dịch các địa chỉ ảo được cung cấp bởi các tiến trình để tương ứng với địa chỉ vật lý.

Mặc cho các mô-đun, hệ điều hành cho phép nhiều tiến trình (multi processes) được tạo ra bởi người dùng để có thể chia sẻ và sử dụng nguồn tính toán ảo (virtual computing resource). Do đó, ở phần bài tập lớn này, nhóm sinh viên phải tập trung vào hiện thực định thời và bộ nhớ ảo.

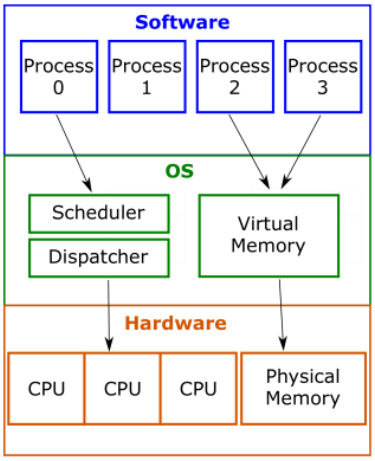


Figure 1: The general view of key modules in this assignment

2 FINDINGS AND CONCLUSIONS

2.1 Scheduler

Nhóm đã hoàn thành các hàm enqueue(), dequeue(), getproc() cũng như hoàn thiện phần thiết kế và hiện thực bộ định thời với giải thuật MLQ hoàn chỉnh. Sau đó cũng đã trả lời câu hỏi, diễn giải kết quả chạy thử nghiệm và vẽ sơ đồ Gantt mô tả các tiến trình được thực hiện bởi CPU như thế nào.

2.2 Memory Management

Nhóm đã hoàn thành phần thiết kế và hiện thực các hàm để hoàn chỉnh phần quản lý bộ nhớ theo cơ chế paging. Sau đó cũng đã trả lời các câu hỏi, diễn giải kết quả chạy thử nghiệm và hiển thị được trạng thái của RAM sau mỗi lần gọi chức năng cấp phát và hủy cấp phát bộ nhớ.

2.3 Put it all together

Kết hợp cả 2 phần định thời và quản lý bộ nhớ, đồng thời xử lý việc bất đồng bộ để chạy thử nghiệm. Sau đó trả lời câu hỏi của phần này.

3 ACTIONS FOR FOLLOW-UP

3.1 Module 1 - Scheduler

3.1.1 Design

Thiết kế MLQ bao gồm
- Tạo thuộc tính
- Hệ thống hàm

3.1.1.1 Tạo thuộc tính

Thêm biến slot (biểu thị timeslot của từng hàng đợi) để dễ dàng quản lý:

```
/* in queue.h */
struct queue_t {
    struct pcb_t * proc[MAX_QUEUE_SIZE];
    int size;

    #ifdef MLQ_SCHED
        int slot;
    #endif
};
```

Thêm biến slot_left để xử lý trong trường hợp hàng đợi sắp hết slot, process đang chạy sẽ bị dang dở:

```
/* in os.c */
//this var is efficient only if out-of-timeslot
static int slot_left = -1;
```

3.1.1.2 Tệp queue.c

Ở đây chúng ta lưu trữ các process trong queue dưới dạng mảng (số lượng giới hạn) theo mô tả đề chứ không lưu dưới dạng linked list. Vì vậy, chúng ta cũng hiện thực các thao tác dequeue và enqueuer trên mảng như sau:

- Hàm **enqueue()**:

+ Hàm enqueue() dùng để đưa process proc vào queue q.
+ Nếu queue q đầy thì return.
+ Ngược lại, ta chèn proc vào cuối queue q đồng thời tăng kích thước queue lên 1 đơn vị.

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

- Hàm **dequeue()**:

+ Hàm dequeue() dùng để đưa process có độ ưu tiên mặc định cao nhất ra khỏi queue q.
+ Nếu queue q rỗng thì return NULL.
+ Ngược lại, ta duyệt từ đầu đến cuối queue và lấy process có độ ưu tiên mặc định cao nhất đầu tiên ra khỏi queue. Đồng thời ta dịch chuyển các phần tử phía sau phần tử cần lấy ra lên trước 1 index và giảm biến size 1 đơn vị.

```
struct pcb_t * dequeue(struct queue_t * q) {
```

```
    /* TODO: return a pcb whose prioprity is the highest
        * in the queue [q] and remember to remove it from q */
    if (q->size == 0) return NULL;

    int max_prio = 0;
    int j;

    for (j = 1; j < q->size; j++)
    {
        if (q->proc[j]->priority < q->proc[max_prio]->priority)
            max_prio = j;
    }

    struct pcb_t *pcb_out = q->proc[max_prio];

    for (j = max_prio; j < q->size - 1; j++)
        q->proc[j] = q->proc[j+1];
    q->proc[q->size - 1] = NULL;
    q->size--;

    return pcb_out;
}
```

3.1.1.3 Tệp sched.c

- Hàm **get_mlq_proc():** Lấy PCB của 1 process đang chờ từ hệ thống ready queue.

```
/* in sched.c */
struct pcb_t * get_mlq_proc(int quantumn, int *slot_left) {
    struct pcb_t * proc = NULL;
    //the level to choose proc from
    uint32_t prio_out = 0;

    while (1)
    {
        //get process satisfies:  the queue not empty && slot != 0
        if (!empty(&mlq_ready_queue[prio_out])){
            if (mlq_ready_queue[prio_out].slot != 0)
            {
                pthread_mutex_lock(&queue_lock);
                proc = dequeue(&mlq_ready_queue[prio_out]);
                //exception 1: proc->code->size doesn't division by quantumn
                //handle in minus_slot_except1()
                mlq_ready_queue[prio_out].slot = mlq_ready_queue[prio_out].slot -
    quantumn;
                //exception 2: out-of-resource, unfinished process
                if (mlq_ready_queue[prio_out].slot < 0)
                {
                    //now slot_left is utilised
                    *slot_left = mlq_ready_queue[prio_out].slot + quantumn;
                    mlq_ready_queue[prio_out].slot = 0;
                }
                pthread_mutex_unlock(&queue_lock);
                return proc; //Sucessfully getproc
            }
        }
        //when we touch the end of MLQ
        if ((++prio_out) == MAX_PRIO)
        {
            prio_out = 0;
            //reset our slots
            int i;
            for (i = 0; i < MAX_PRIO; i++)
            {
                mlq_ready_queue[i].slot = MAX_PRIO - i;
```

```
            }
            break;
        }
    }

    return NULL; //Empty-handed collection
}
```

- Hàm **minus_slot_except1():** giúp ta giải quyết ngoại lệ 1: proc->code->size không chia hết cho quantum time.

```
void minus_slot_except1(struct pcb_t * proc, int quantumn){
    pthread_mutex_lock(&queue_lock);
    if (proc->code->size % quantumn != 0)
    {
        mlq_ready_queue[proc->prio].slot = mlq_ready_queue[proc->prio].slot + quantumn - proc->code->size % quantumn;
    }
    pthread_mutex_unlock(&queue_lock);
}
```

3.1.1.4 Tệp os.c – hàm cpu_routine()

- Đầu tiên chúng tôi thêm hàm xử lý ngoại lệ 1 minus_slot_except1 ở tùy chọn proc->pc = proc->code->size.
- Sau đó, thêm tùy chọn (slot_left == 0) trong phần "Check the status of current process" và giảm biến slot_left khi process được chạy (xử lý ngoại lệ 2).

```
static void * cpu_routine(void * args) {
    struct timer_id_t * timer_id = ((struct cpu_args*)args)->timer_id;
    int id = ((struct cpu_args*)args)->id;
    /* Check for new process in ready queue */
    int time_left = 0;
    struct pcb_t * proc = NULL;
    while (1) {
        /* Check the status of current process */
        if (proc == NULL) {
            /* No process is running, the we load new process from
            * ready queue */
            proc = get_proc(time_slot, &slot_left);
            if (proc == NULL) {
                        next_slot(timer_id);
                        continue; /* First load failed. skip dummy load */
                    }
        }else if (proc->pc == proc->code->size) {
            /* The porcess has finish it job */
            printf("\tCPU %d: Processed %2d has finished\n",
                id ,proc->pid);
            //handling except1 of get_proc()
            minus_slot_except1(proc, time_slot);
            free(proc);
            proc = get_proc(time_slot, &slot_left);
            time_left = 0;
        }else if (time_left == 0) {
            /* The process has done its job in current time slot */
            printf("\tCPU %d: Put process %2d to run queue\n",
                id, proc->pid);
            put_proc(proc);
            proc = get_proc(time_slot, &slot_left);
        }
        //added option
        else if (slot_left == 0)
        {
            /* This queue ran-out-of time slot */
```

```
            printf("\tCPU %d: Put process %2d to run queue because of depleting r
esources\n",
                id, proc->pid);
            put_proc(proc);
            slot_left = -1;
            proc = get_proc(time_slot, &slot_left);
        }

        /* Recheck process status after loading new process */
        if (proc == NULL && done) {
            /* No process to run, exit */
            printf("\tCPU %d stopped\n", id);
            break;
        }else if (proc == NULL) {
            /* There may be new processes to run in
             * next time slots, just skip current slot */
            next_slot(timer_id);
            continue;
        }else if (time_left == 0) {
            printf("\tCPU %d: Dispatched process %2d\n",
                id, proc->pid);
            time_left = time_slot;
        }
        /* Run current process */
        run(proc);
        time_left--;
        //handling exception 2
        if (slot_left > 0)
            slot_left--;
        next_slot(timer_id);
    }
    detach_event(timer_id);
    pthread_exit(NULL);
}
```

3.1.2 Test

Sử dụng make file: make sched

3.1.2.1 Test 1

- Input:
Configuration: sched_0

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

Process:

Process	s0	s1	s2	s3
Description	12 15 calc calc calc calc calc calc calc calc calc calc calc	20 7 calc calc calc calc calc calc calc	20 13 calc calc calc calc calc calc calc calc calc calc calc	7 17 calc calc calc calc calc calc calc calc calc calc calc

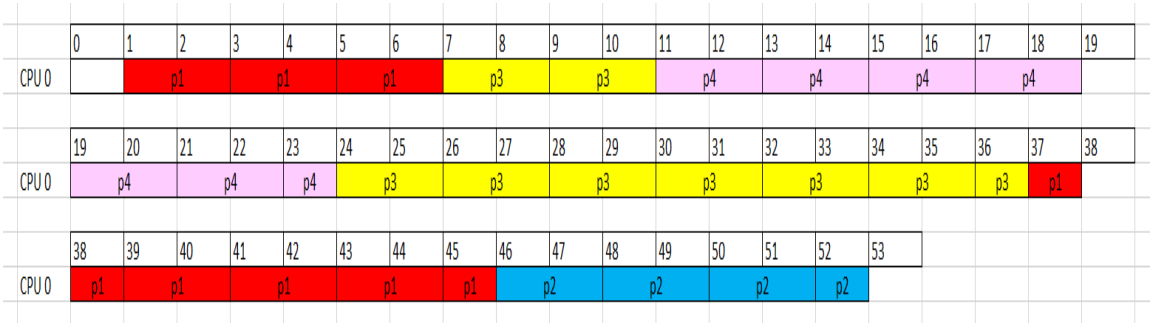
	calc calc calc calc		calc calc	calc calc calc calc calc calc

- Output: ./sched sched_0

```
Time slot   0
ld_routine
        Loaded a process at input/proc/s0, PID: 1 PRIO: 1 at time 0
Time slot   1
        CPU 0: Dispatched process  1 at time 1
Time slot   2
Time slot   3
        CPU 0: Put process  1 to run queue at time 3
        CPU 0: Dispatched process  1 at time 3
Time slot   4
        Loaded a process at input/proc/s1, PID: 2 PRIO: 2 at time 4
Time slot   5
        CPU 0: Put process  1 to run queue at time 5
        CPU 0: Dispatched process  1 at time 5
Time slot   6
Time slot   7
        Loaded a process at input/proc/s3, PID: 3 PRIO: 1 at time 7
        CPU 0: Put process  1 to run queue at time 7
        CPU 0: Dispatched process  3 at time 7
Time slot   8
Time slot   9
        CPU 0: Put process  3 to run queue at time 9
        CPU 0: Dispatched process  3 at time 9
Time slot  10
        Loaded a process at input/proc/s2, PID: 4 PRIO: 0 at time 10
Time slot  11
        CPU 0: Put process  3 to run queue at time 11
        CPU 0: Dispatched process  4 at time 11
Time slot  12
Time slot  13
        CPU 0: Put process  4 to run queue at time 13
        CPU 0: Dispatched process  4 at time 13
Time slot  14
Time slot  15
        CPU 0: Put process  4 to run queue at time 15
        CPU 0: Dispatched process  4 at time 15
Time slot  16
Time slot  17
        CPU 0: Put process  4 to run queue at time 17
        CPU 0: Dispatched process  4 at time 17
Time slot  18
Time slot  19
        CPU 0: Put process  4 to run queue at time 19
        CPU 0: Dispatched process  4 at time 19
Time slot  20
Time slot  21
        CPU 0: Put process  4 to run queue at time 21
        CPU 0: Dispatched process  4 at time 21
Time slot  22
Time slot  23
        CPU 0: Put process  4 to run queue at time 23
        CPU 0: Dispatched process  4 at time 23
Time slot  24
        CPU 0: Processed  4 has finished at time 24
        CPU 0: Dispatched process  3 at time 24
Time slot  25
Time slot  26
        CPU 0: Put process  3 to run queue at time 26
```

```
        CPU 0: Dispatched process  3 at time 26
Time slot  27
Time slot  28
        CPU 0: Put process  3 to run queue at time 28
        CPU 0: Dispatched process  3 at time 28
Time slot  29
Time slot  30
        CPU 0: Put process  3 to run queue at time 30
        CPU 0: Dispatched process  3 at time 30
Time slot  31
Time slot  32
        CPU 0: Put process  3 to run queue at time 32
        CPU 0: Dispatched process  3 at time 32
Time slot  33
Time slot  34
        CPU 0: Put process  3 to run queue at time 34
        CPU 0: Dispatched process  3 at time 34
Time slot  35
Time slot  36
        CPU 0: Put process  3 to run queue at time 36
        CPU 0: Dispatched process  3 at time 36
Time slot  37
        CPU 0: Processed  3 has finished at time 37
        CPU 0: Dispatched process  1 at time 37
Time slot  38
Time slot  39
        CPU 0: Put process  1 to run queue at time 39
        CPU 0: Dispatched process  1 at time 39
Time slot  40
Time slot  41
        CPU 0: Put process  1 to run queue at time 41
        CPU 0: Dispatched process  1 at time 41
Time slot  42
Time slot  43
        CPU 0: Put process  1 to run queue at time 43
        CPU 0: Dispatched process  1 at time 43
Time slot  44
Time slot  45
        CPU 0: Put process  1 to run queue at time 45
        CPU 0: Dispatched process  1 at time 45
Time slot  46
                CPU 0: Processed  1 has finished at time 46
                CPU 0: Dispatched process  2 at time 46
       Time slot  47
       Time slot  48
                CPU 0: Put process  2 to run queue at time 48
                CPU 0: Dispatched process  2 at time 48
       Time slot  49
       Time slot  50
                CPU 0: Put process  2 to run queue at time 50
                CPU 0: Dispatched process  2 at time 50
       Time slot  51
       Time slot  52
                CPU 0: Put process  2 to run queue at time 52
                CPU 0: Dispatched process  2 at time 52
       Time slot  53
                CPU 0: Processed  2 has finished at time 53
                CPU 0 stopped at time 53
```

- Sơ đồ Gantt:



- Giải thích:

+ Time 4 load process 2 nhưng vì process 2 có prio nhỏ hơn của process 1 nên process 1 được tiếp tục đưa vào CPU

+ Time 7 load process 3 có prio bằng với của process 1 nhưng có độ ưu tiên mặc định priority lớn hơn của process 1 nên process 3 được đưa vào CPU lúc đó

+ Time 10 load process 4 nhưng process 3 chưa thực hiện hết quantum nên vẫn chờ

+ Time 11, vì process 4 có prio lớn nhất nên được đưa vào CPU thay thế

+ Tương tự cho các time slot sau đó

3.1.2.2 Test 2

- Input:

Configuration: sched_1

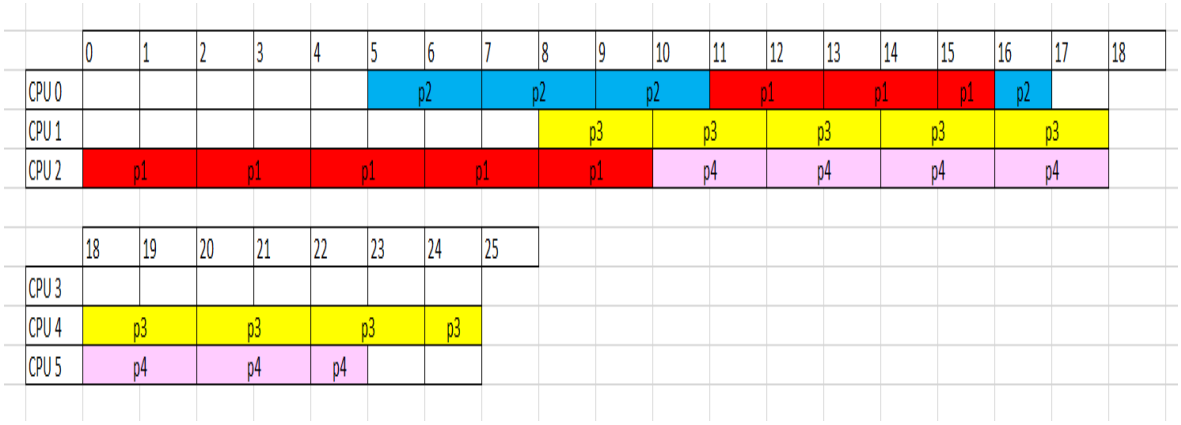
```
2 3 4
0 s0 1
4 s1 2
7 s3 1
10 s2 0
```

- Output: ./sched sched_1

```
Time slot   0
ld_routine
        Loaded a process at input/proc/s0, PID: 1 PRIO: 1 at time 0
        CPU 2: Dispatched process  1 at time 0
Time slot   1
Time slot   2
        CPU 2: Put process  1 to run queue at time 2
        CPU 2: Dispatched process  1 at time 2
Time slot   3
Time slot   4
        Loaded a process at input/proc/s1, PID: 2 PRIO: 2 at time 4
        CPU 2: Put process  1 to run queue at time 4
        CPU 2: Dispatched process  1 at time 4
Time slot   5
        CPU 0: Dispatched process  2 at time 5
Time slot   6
        CPU 2: Put process  1 to run queue at time 6
        CPU 2: Dispatched process  1 at time 6
Time slot   7
        CPU 0: Put process  2 to run queue at time 7
        CPU 0: Dispatched process  2 at time 7
        Loaded a process at input/proc/s3, PID: 3 PRIO: 1 at time 7
Time slot   8
        CPU 2: Put process  1 to run queue at time 8
        CPU 2: Dispatched process  1 at time 8
        CPU 1: Dispatched process  3 at time 8
Time slot   9
        CPU 0: Put process  2 to run queue at time 9
        CPU 0: Dispatched process  2 at time 9
        Loaded a process at input/proc/s2, PID: 4 PRIO: 0 at time 10
        CPU 2: Put process  1 to run queue at time 10
```

```
        CPU 2: Dispatched process  4 at time 10
Time slot  10
        CPU 1: Put process  3 to run queue at time 10
        CPU 1: Dispatched process  3 at time 10
Time slot  11
        CPU 0: Put process  2 to run queue at time 11
        CPU 0: Dispatched process  1 at time 11
        CPU 2: Put process  4 to run queue at time 12
        CPU 2: Dispatched process  4 at time 12
Time slot  12
        CPU 1: Put process  3 to run queue at time 12
        CPU 1: Dispatched process  3 at time 12
Time slot  13
        CPU 0: Put process  1 to run queue at time 13
        CPU 0: Dispatched process  1 at time 13
        CPU 2: Put process  4 to run queue at time 14
        CPU 2: Dispatched process  4 at time 14
Time slot  14
        CPU 1: Put process  3 to run queue at time 14
        CPU 1: Dispatched process  3 at time 14
Time slot  15
        CPU 0: Put process  1 to run queue at time 15
        CPU 0: Dispatched process  1 at time 15
        CPU 2: Put process  4 to run queue at time 16
        CPU 2: Dispatched process  4 at time 16
        CPU 1: Put process  3 to run queue at time 16
        CPU 1: Dispatched process  3 at time 16
        CPU 0: Processed  1 has finished at time 16
        CPU 0: Dispatched process  2 at time 16
Time slot  16
Time slot  17
        CPU 0: Processed  2 has finished at time 17
        CPU 0 stopped at time 17
        CPU 2: Put process  4 to run queue at time 18
        CPU 2: Dispatched process  4 at time 18
        CPU 1: Put process  3 to run queue at time 18
        CPU 1: Dispatched process  3 at time 18
Time slot  18
Time slot  19
Time slot  20
        CPU 2: Put process  4 to run queue at time 20
        CPU 2: Dispatched process  4 at time 20
        CPU 1: Put process  3 to run queue at time 20
        CPU 1: Dispatched process  3 at time 20
Time slot  21
Time slot  22
        CPU 1: Put process  3 to run queue at time 22
        CPU 1: Dispatched process  3 at time 22
        CPU 2: Put process  4 to run queue at time 22
        CPU 2: Dispatched process  4 at time 22
Time slot  23
        CPU 2: Processed  4 has finished at time 23
        CPU 2 stopped at time 23
Time slot  24
        CPU 1: Put process  3 to run queue at time 24
        CPU 1: Dispatched process  3 at time 24
Time slot  25
        CPU 1: Processed  3 has finished at time 25
        CPU 1 stopped at time 25
```

- Sơ đồ Gantt:



3.1.3 Answer the question

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

Answer:

Giải thuật Priority Queue nghĩa là mỗi lần lấy 1 tiến trình từ queue thì tiến trình phải là tiến trình có độ ưu tiên (default priority) cao nhất.

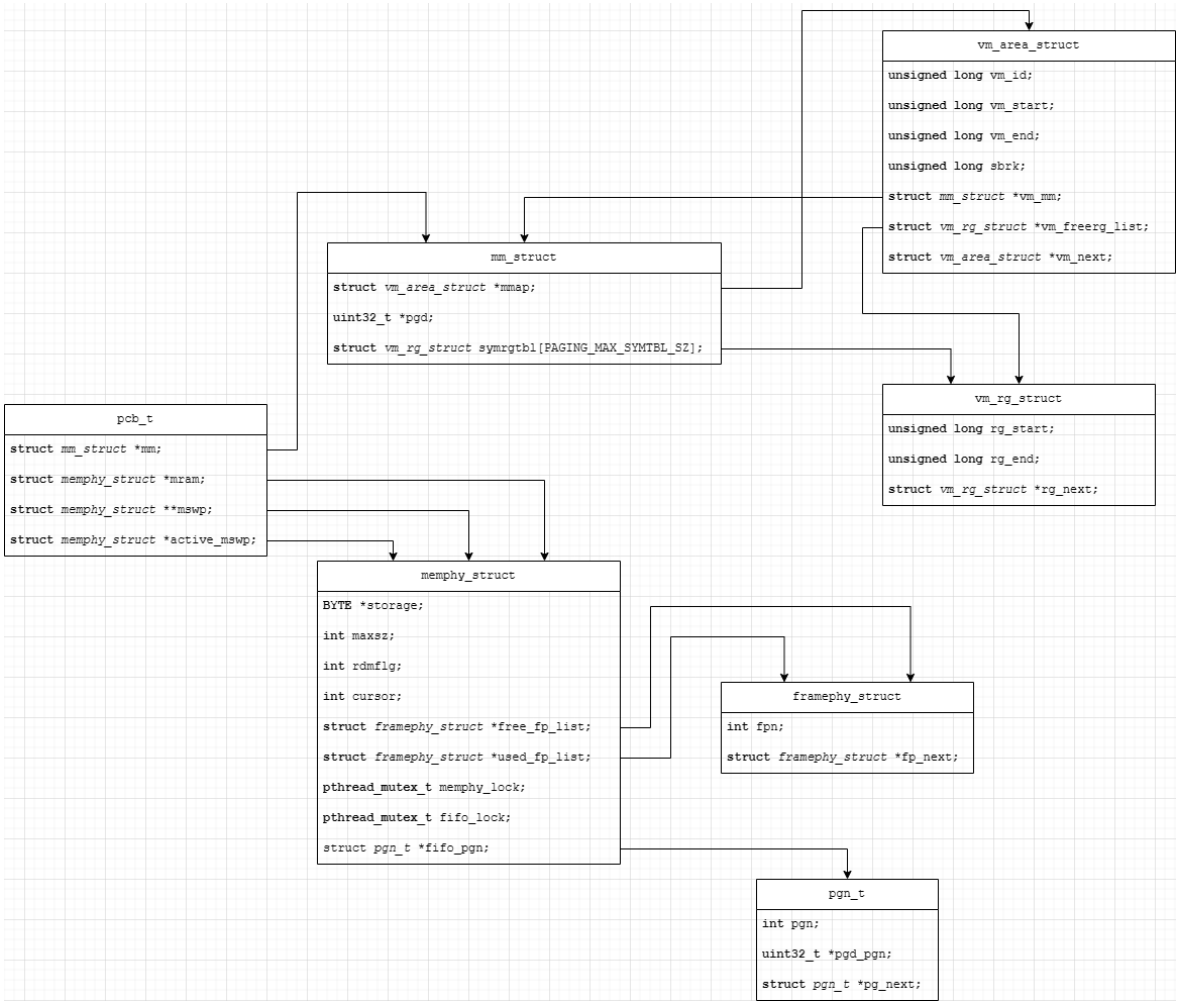
Khi so sánh với các giải thuật định thời khác thì giải thuật này có những lợi ích sau:

- Đơn giản, dễ hiểu, dễ sử dụng
- Các process có độ ưu tiên cao hơn thì thực thi trước giúp tiết kiệm thời gian cho các process quan trọng
- Tầm quan trọng của mỗi tiến trình được xác định chính xác.
- Cho phép gán các ưu tiên khác nhau cho các tiến trình dựa trên tầm quan trọng, mức độ khẩn cấp hoặc các tiêu chí khác của chúng.
- Đây là một giải thuật tốt cho việc ứng dụng với các yêu cầu về thời gian và nguồn lực luôn thay đổi.

3.2 Module 2 - Paging-based memory management

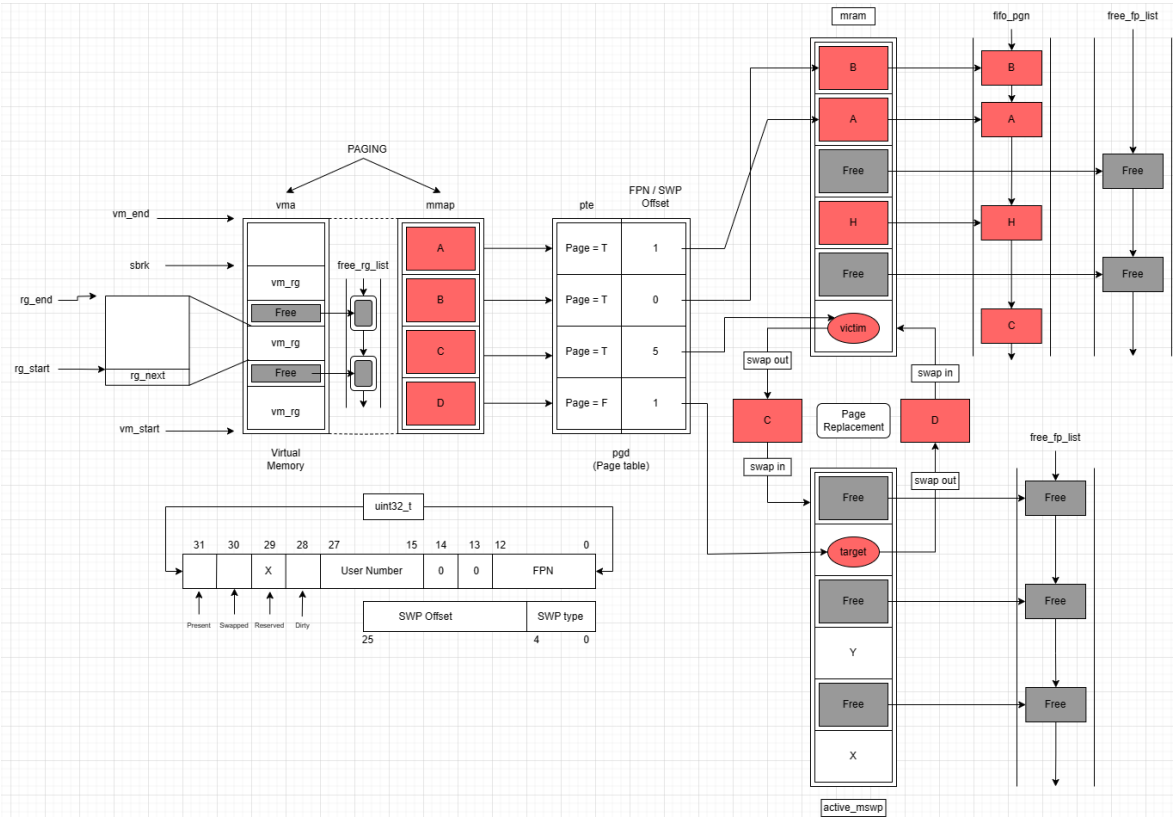
3.2.1 Design

Sự liên kết của các struct trong bài tập lớn này sẽ được minh họa bằng mô hình như dưới đây:



Một sự thay đổi so với source code template đó là trong **struct memphy_struct** đã được khai báo thêm ba thuộc tính đó là memphy_lock, fifo_lock để tránh tình trạng racing (tranh chấp tài nguyên xài chung của các process) và cuối cùng đó là thuộc tính fifo_pgn. Thuộc tính fifo_pgn có kiểu **struct pgn_t** được bổ sung thêm thuộc tính pgd_pgn. Chi tiết của việc thêm bớt này sẽ được trình bày chi tiết hơn trong ngoại lệ **OOM, OOR.**

Hình dưới đây mô tả toàn bộ cơ chế paging mà ta sử dụng trong bài tập lớn này



3.2.1.1 Tệp mm-memphy.c

- Hàm **MEMPHY_dump():**

```
1.  int MEMPHY_dump(struct memphy_struct * mp)
2.  {
3.      /*TODO dump memphy contnt mp->storage
4.       *     for tracing the memory content
5.       */
6.     pthread_mutex_lock(&mp->memphy_lock);
7.
8.     const int FIXED_SIZE = 1000000;
9.     char content[FIXED_SIZE]; //enough space for output format
10.    int i = 0, index = 0;
11.    while(i < mp->maxsz){
12.       if(mp->storage[i]){
13.          uint32_t pte = 0;
14.          pte_set_fpn(&pte, i/256);
15.          index += sprintf(&content[index], "\n                [%08x]-
    [%d,%d]", pte, (i-i/256*256), mp->storage[i]);
16.       }
17.       ++i;
18.    }
19.
20.    printf("Memory content [addr]-
    [offset,value]: %s| max size = %d\n\n",content, mp->maxsz);
21.
22.    pthread_mutex_unlock(&mp->memphy_lock);
23.
24.    return 0;
25. }
```

Hàm này mục đích là in ra nội dung trong bộ nhớ RAM hiện tại. Để cho dễ dàng kiểm tra tính đúng đắn khi đọc nội dung để debug, ta sẽ dùng cơ chế mutex lock với biến lock là memphy_lock. Các thao tác liên quan đến việc thêm frame trống hay lấy ra

frame trống ở free_fp_list hay đọc – ghi dữ liệu vào RAM sẽ đều xài chung biến lock này. Khi in ra nội dung sẽ như hình dưới đây:

```
Memory content [addr]-[offset,value]:
               [80000001]-[20,100]
               [80000005]-[64,102]
               [80000006]-[40,1]| max size = 1048576
```

3.2.1.2 Tệp mm-vm.c

- Hàm **__alloc():**

```
1.  int __alloc(struct pcb_t *caller, int vmaid, int rgid, int size, int *alloc_addr)
2.  {
3.
4.    /*Allocate at the toproof */
5.    struct vm_rg_struct rgnode;
6.
7.    if (get_free_vmrg_area(caller, vmaid, size, &rgnode) == 0)
8.    {
9.      caller->mm->symrgtbl[rgid].rg_start = rgnode.rg_start;
10.     caller->mm->symrgtbl[rgid].rg_end = rgnode.rg_end;
11.
12.     *alloc_addr = rgnode.rg_start;
13.
14.     return 0;
15.   }
16.
17.   /* TODO get_free_vmrg_area FAILED handle the region management (Fig.6)*/
18.
19.   /*Attempt to increate limit to get space */
20.   struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);
21.   //int inc_sz = PAGING_PAGE_ALIGNSZ(size);
22.
23.   int old_sbrk ;
24.   old_sbrk = cur_vma->sbrk;
25.   int remain = cur_vma->vm_end - old_sbrk;
26.   int inc_sz;
27.
28.   /* TODO INCREASE THE LIMIT
29.    * inc_vma_limit(caller, vmaid, inc_sz)
30.    */
31.   if(size > remain) //not enough space for allocation
32.   {
33.     inc_sz = PAGING_PAGE_ALIGNSZ(size - remain);
34.     if(inc_vma_limit(caller, vmaid, inc_sz) < 0) //overlap or out of mem
35.       return -1;
36.   }
37.
38.   /*Successful increase limit */
39.   caller->mm->symrgtbl[rgid].rg_start = old_sbrk;
40.   caller->mm->symrgtbl[rgid].rg_end = old_sbrk + size;
41.
42.   *alloc_addr = old_sbrk;
43.   cur_vma->sbrk += size;
44.
45.   return 0;
46. }
```

Hàm này sẽ yêu cầu virtual memory cung cấp cho nó một region với kích thước là size. Việc tìm free region sẽ sử dụng giải thuật First-Fit tức là sẽ lấy ngay free region đầu tiên có kích thước không nhỏ hơn size khi traverse qua danh sách các free region thông qua hàm get_free_vmrg_area.

+ Nếu kết quả trả về là bằng 0 (tìm kiếm thành công) thì ta sẽ set symrgtbl[rgid] như đoạn code trên.

+ Nếu kết quả trả về là -1 (không có free region nào fit với size hoặc không có free region nào trong danh sách). Khi đó việc quyết định có bắt buộc cung cấp thêm bộ nhớ ảo với lượng pages vừa đủ với kích thước size được yêu cầu hay không sẽ được thông qua việc so sánh size với remain như code ở trên bởi vì còn 1 khoảng trống mà chúng ta chưa tính tới đó là giữa sbrk và vm_end của vùng nhớ ảo vma. Biến inc_size sẽ tính ra được lượng kích thước (bội của kích thước 1 Page hay 1 Frame) ta cần cung cấp nếu size > remain.

- Hàm **inc_vma_limit():**

```
1.  int inc_vma_limit(struct pcb_t *caller, int vmaid, int inc_sz)
2.  {
3.    struct vm_rg_struct * newrg = malloc(sizeof(struct vm_rg_struct));
4.    int inc_amt = PAGING_PAGE_ALIGNSZ(inc_sz);
5.    int incnumpage =  inc_amt / PAGING_PAGESZ;
6.    struct vm_rg_struct *area = get_vm_area_node_at_brk(caller, vmaid, inc_sz, inc_
    amt);
7.    struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);
8.
9.    int old_end = cur_vma->vm_end;
10.
11.   /*Validate overlap of obtained region */
12.   if (validate_overlap_vm_area(caller, vmaid, area->rg_start, area-
    >rg_end) < 0){
13.     free(area); //avoid mem leak
14.     return -1; /*Overlap and failed allocation */
15.   }
16.
17.   /* The obtained vm area (only)
18.    * now will be alloc real ram region */
19.   cur_vma->vm_end += inc_sz;
20.   if (vm_map_ram(caller, area->rg_start, area->rg_end,
21.                           old_end, incnumpage , newrg) < 0)
22.   {
23.     free(area); //avoid mem leak
24.     return -1; /* Map the memory to MEMRAM */
25.   }
26.
27.   free(area); //avoid mem leak
28.   return 0;
29. }
```

Hàm này sẽ thực hiện thao tác tăng kích thước của vùng nhớ ảo vma với kích thước inc_sz mà hàm __alloc được yêu cầu. Hàm validate_overlap_vm_area kiểm tra xem vùng nhớ ảo đang xét hiện tại có bị overlap hay không (việc này không xảy ra trong bài tập lớn này). Sau đó ta sẽ map các Pages giữa vùng nhớ ảo đó với bộ nhớ vật lí RAM thông qua hàm vm_map_ram. Mọi hàm đề trả về 0 nếu các công đoạn trong hàm đều thành công.

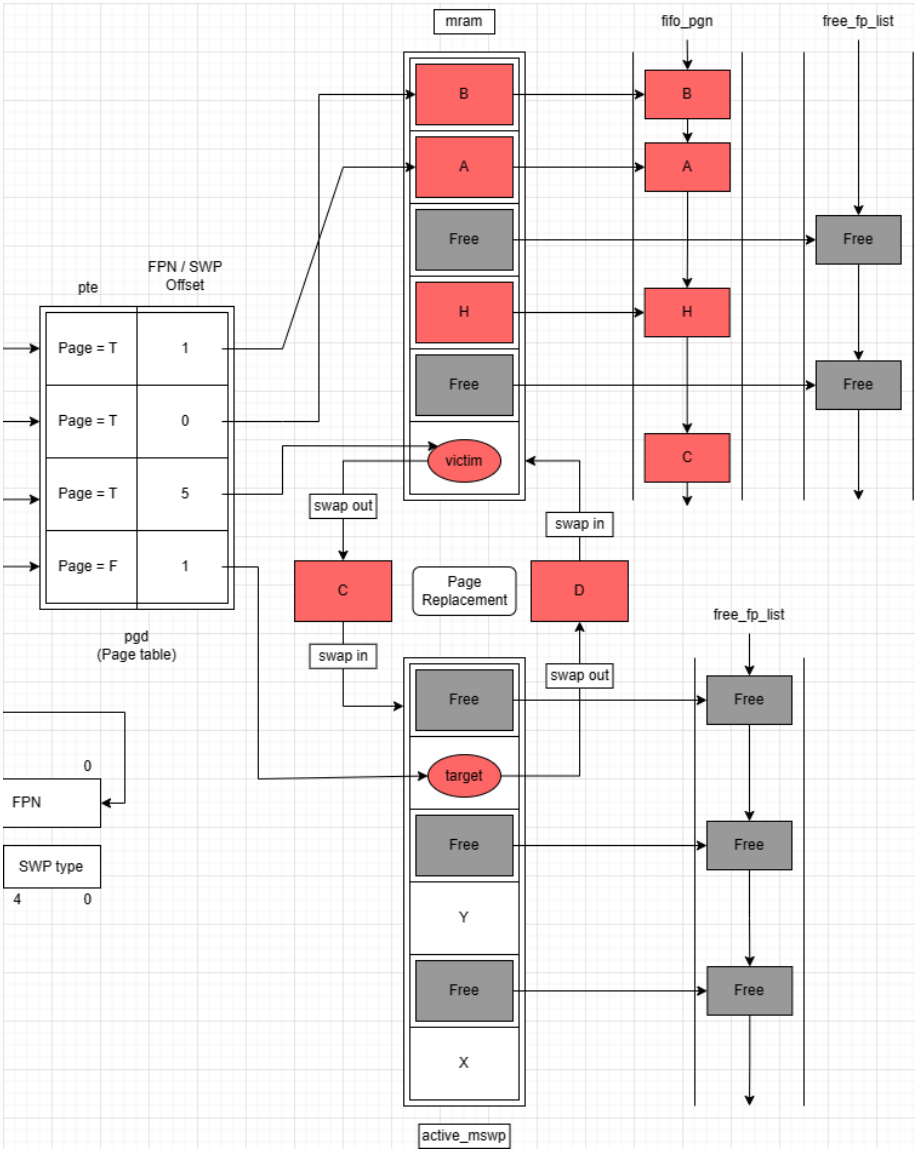
- Hàm **pg_getpage():**

```
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.
5.    if (!PAGING_PAGE_PRESENT(pte))
6.    { /* Page is not online, make it actively living */
7.
8.      struct pgn_t *fifo_node = malloc(sizeof(struct pgn_t));
9.      int swpfpn;
10.     int tgtfpn = PAGING_SWP(pte);//the target frame storing our variable
11.     printf("-----[PID: %d] PAGE FAULT----- target page in swap: %08x\n", caller-
    >pid, mm->pgd[pgn]);
12.
```

```
13.     /* TODO: Play with your paging theory here */
14.     /* Find victim page in virtual mem */
15.     if(find_victim_page(caller->mram, &fifo_node) < 0)
16.     {
17. #ifdef MMDBG
18.       printf("-----[PID: %d] PAGE FAULT----
    - pg_getpage failed - no victim to swapoff\n", caller->pid);
19. #endif
20.       return -1;
21.     }
22.
23.     printf("-----[PID: %d] PAGE FAULT----- found victim: %08x\n", caller-
    >pid, *fifo_node->pgd_pgn);
24.
25.     /*victim in ram*/
26.     int ram_vicpgn = PAGING_FPN(*fifo_node->pgd_pgn);
27.
28.     /* Get free frame in MEMSWP */
29.     if(MEMPHY_get_freefp(caller->active_mswp, &swpfpn) < 0) return -1;
30.
31.     /* Do swap frame from MEMRAM to MEMSWP and vice versa*/
32.     /* Copy victim frame to swap */
33.     __swap_cp_page(caller->mram, ram_vicpgn, caller->active_mswp, swpfpn);
34.
35.     /* Copy target frame from swap to mem */
36.     __swap_cp_page(caller->active_mswp, tgtfpn, caller->mram, ram_vicpgn);
37.
38.     /* Make the tgtfpn in activeswap free again*/
39.     MEMPHY_put_freefp(caller->active_mswp, tgtfpn);
40.
41.     /* Update page table */
42.     pte_set_swap(fifo_node->pgd_pgn, 0, swpfpn);
43.     printf("-----[PID: %d] PAGE FAULT----- victim_present: %08x\n", caller-
    >pid, *fifo_node->pgd_pgn);
44.     free(fifo_node); //avoid mem leak
45.
46.     /* Update its online status of the target page */
47.     pte_set_fpn(&pte, ram_vicpgn);
48.     mm->pgd[pgn] = pte;
49.     printf("-----[PID: %d] PAGE FAULT----- target_put: %08x\n", caller->pid, mm-
    >pgd[pgn]);
50.
51.     enlist_pgn_node(caller->mram, pgn, &mm->pgd[pgn]);
52.   }
53.
54.   *fpn = PAGING_FPN(pte);
55.
56.   return 0;
57. }
```

Hàm này giúp lưu giá trị frame page number trong RAM tương ứng với page number ta yêu cầu vào biến fpn được truyền tham khảo phục vụ cho việc đọc ghi dữ liệu vào RAM. Nhưng không phải lúc nào frame page number ta cần tìm cũng nằm trong RAM đôi khi nó nằm trong bộ nhớ SWAP và bắt buộc phải thực hiện cơ chế thay trang (Page Replacement). Ở đây ta sẽ sử dụng cơ chế FIFO trang nào vào trước thì sẽ ra trước và thứ tự FIFO ở đây là trang cũ nhất sẽ nằm ở cuối danh sách FIFO.

Hình mô tả cơ chế này minh họa đưới đây:



- Hàm **__free():**

```
1.  int __free(struct pcb_t *caller, int vmaid, int rgid)
2.  {
3.    struct vm_rg_struct *rgnode;
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.
11.   /*enlist the obsoleted memory region */
12.   enlist_vm_freerg_list(caller->mm, rgnode);
13.
14.   printf("[PID: %d]----After enlist----\n", caller->pid);
15.   print_list_rg(caller->mm->mmap->vm_freerg_list);
16.
17.   return 0;
18. }
```

Hàm này chỉ đơn giản là đưa region cần free vào danh sách *vm_freerg_list* thông qua hàm **enlist_vm_freerg_list**.

- Hàm **enlist_vm_freerg_list():**

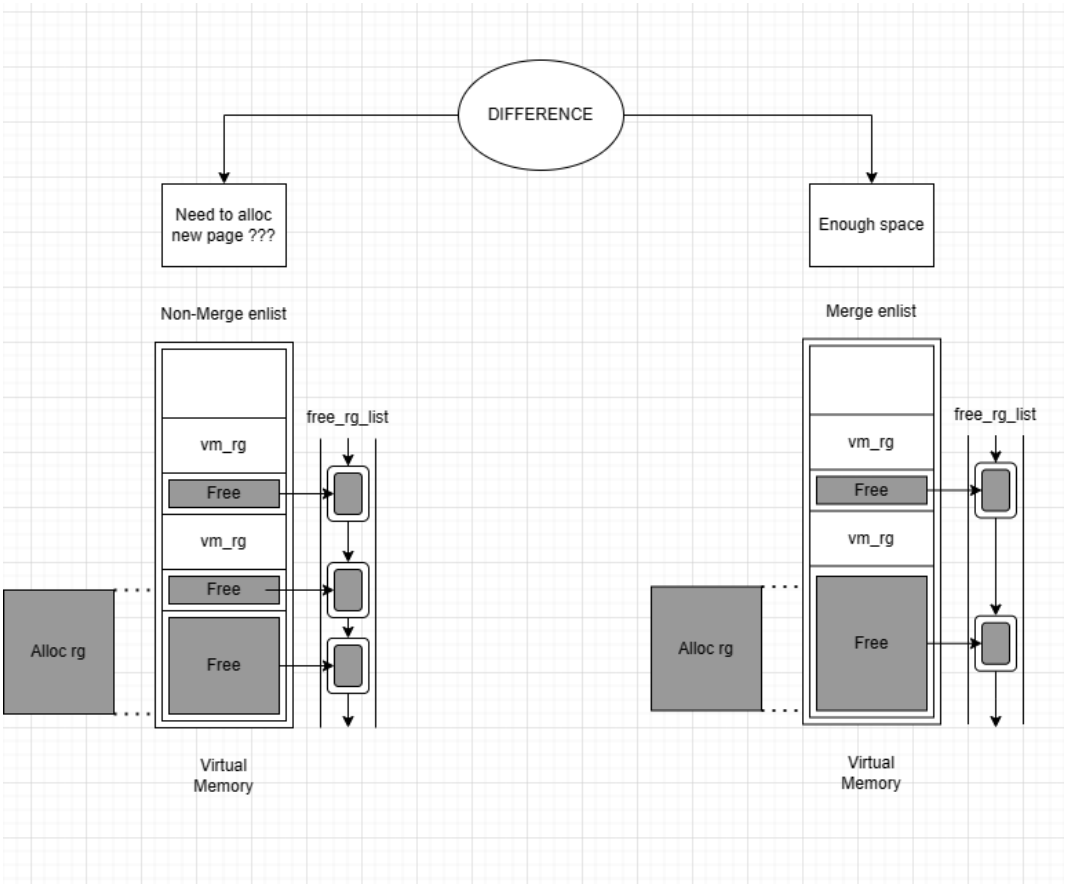
```
1.  int enlist_vm_freerg_list(struct mm_struct *mm, struct vm_rg_struct *rg_elmt)
2.  {
3.    struct vm_rg_struct *rg_node = mm->mmap->vm_freerg_list;
4.
5.    if (rg_elmt->rg_start >= rg_elmt->rg_end)
6.      return -1;
7.    /*Create newNode same value as rg_elmt*/
8.    struct vm_rg_struct *newNode = malloc(sizeof(struct vm_rg_struct));
9.    newNode->rg_start = rg_elmt->rg_start;
10.   newNode->rg_end = rg_elmt->rg_end;
11.   newNode->rg_next = NULL;
12.
13.   /*enlist at head of linkedlist*/
14.   if(rg_node == NULL || newNode->rg_end < rg_node->rg_start)
15.   {
16.     newNode->rg_next = rg_node;
17.     mm->mmap->vm_freerg_list = newNode;
18.     return 0;
19.   }
20.   struct vm_rg_struct *curr = mm->mmap->vm_freerg_list;
21.   struct vm_rg_struct *prev = NULL;
22.   int merged = 0;
23.
24.   /*Merging front*/
25.   while(curr != NULL)
26.   {
27.     if(newNode->rg_start > curr->rg_end)
28.     {
29.       prev = curr;
30.       curr = curr->rg_next;
31.     }
32.
33.     else if(newNode->rg_end >= curr->rg_start || newNode->rg_start <= curr-
    >rg_end)
34.     {
35.       curr->rg_start = (newNode->rg_start < curr->rg_start) ? newNode-
    >rg_start : curr->rg_start;
36.       curr->rg_end = (newNode->rg_end > curr->rg_end) ? newNode->rg_end : curr-
    >rg_end;
37.
38.       merged = 1;
39.       free(newNode);
40.       break;
41.     }
42.     break;
43.   }
44.
45.   /*Merged check*/
46.   if(!merged)
47.   {
48.     prev->rg_next = newNode;
49.     newNode->rg_next = curr;
50.
51.     curr = newNode; //Move to the newNode position
52.   }
53.
54.   /*Merging behind*/
55.   struct vm_rg_struct *curr_next = curr->rg_next;
56.   if(curr_next != NULL && curr_next->rg_start <= curr->rg_end)
57.   {
58.     curr->rg_end = (curr_next->rg_end > curr->rg_end) ? curr_next->rg_end : curr-
    >rg_end;
59.
60.     /*clone curr_next*/
```

```
61.     curr->rg_next = curr_next->rg_next;
62.     free(curr_next);
63.   }
64.
65.   return 0;
66. }
```

Hàm này sẽ thực việc đưa region free vào danh sách vm_freerg_list bằng một phần giải thuật insertion sort, tức là ví region như một đoạn thẳng có giá trị start-end và sẽ nhét region này vào một danh sách đã sắp xếp sẵn theo thứ tự start tăng dần sau đó gộp node nếu node đưa vào có biên start-end bằng với end-start của node trước-sau. Minh họa việc này thông qua hình dưới đây:

```
Start: 0
End: 300
Insert (0, 300)
After insert: (0, 300) -> NULL
-----------------
Start: 600
End: 900
Insert (600, 900)
After insert: (0, 300) -> (600, 900) -> NULL
-----------------
Start: 300
End: 600
Insert (300, 600)
After insert: (0, 900) -> NULL
-----------------
```

Sự giải thích cho việc làm merge và sort này là vì bộ nhớ ảo được alloc theo cơ chế đó là continugous allocation. Nếu ta cứ đưa region free vào đầu danh sách thì sẽ có hiện tương như hình sau:



Hình bên trái là enlist không có sự sort và merge, còn hình bên phải thì có tất cả. Tuy cách làm này không hoàn hảo vì độ phức tạp của nó là O(N) chậm hơn so với cách không merge – O(1) nhưng ta có thể thấy được lợi ích của việc làm này là nó tối ưu được bộ nhớ tránh cấp phát dư thừa và hiện tượng hết bộ nhớ.

3.2.1.3 Tệp mm.c

- Hàm **vm_map_ram():**

```
1.  int vm_map_ram(struct pcb_t *caller, int astart, int aend, int mapstart, int incp
    gnum, struct vm_rg_struct *ret_rg)
2.  {
3.    struct framephy_struct *frm_lst = NULL;
4.    int ret_alloc;
5.
6.    /*@bksysnet: author provides a feasible solution of getting frames
7.     *FATAL logic in here, wrong behaviour if we have not enough page
8.     *i.e. we request 1000 frames meanwhile our RAM has size of 3 frames
9.     *Don't try to perform that case in this simple work, it will result
10.    *in endless procedure of swap-off to get frame and we have not provide
11.    *duplicate control mechanism, keep it simple
12.    */
13.   ret_alloc = alloc_pages_range(caller, incpgnum, &frm_lst);
14.
15.   if (ret_alloc < 0 && ret_alloc != -3000)
16.   {
17. #ifdef MMDBG
18.     printf("[PID: %d] ----OOR: vm_map_ram failed - no victim to swapoff----
    \n", caller->pid);
19. #endif
20.     return -1;
21.   }
22.
```

```
  /* Out of memory */
  if (ret_alloc == -3000)
  {
#ifdef MMDBG
    printf("[PID: %d] ----
OOM: vm_map_ram out of memory - no freeframe in swapper----\n", caller->pid);
#endif
    return -1;
  }

  /* it leaves the case of memory is enough but half in ram, half in swap
   * do the swaping all to swapper to get the all in ram */
  vmap_page_range(caller, mapstart, incpgnum, frm_lst, ret_rg);

  return 0;
}
```

Mục tiêu hàm này là sẽ map các pages được yêu cầu alloc với các frames trong RAM mà đang free. Về hai ngoại lệ sẽ được giải thích chi tiết ở hàm sau.

- Hàm **alloc_pages_range():**

```
int alloc_pages_range(struct pcb_t *caller, int req_pgnum, struct framephy_struct
** frm_lst)
{
  int pgit, fpn;

  for(pgit = 0; pgit < req_pgnum; pgit++)
  {
    if(MEMPHY_get_freefp(caller->mram, &fpn) < 0)
    {
      //RAM is full, move some frames in RAM to swap to get more space in RAM
      int swpfpn;
      struct pgn_t *fifo_node = malloc(sizeof(struct pgn_t));

      /* Find victim page in RAM */
      if(find_victim_page(caller->mram, &fifo_node) < 0)
       return -1;


      printf("-----[PID: %d] victim_OOR: %08x\n", caller->pid, *fifo_node-
>pgd_pgn);

      /* Get free frame in MEMSWP */
      if(MEMPHY_get_freefp(caller->active_mswp, &swpfpn) < 0)
      {

        /*put the victim back to the RAM*/
        enlist_pgn_node(caller->mram, fifo_node->pgn, fifo_node->pgd_pgn);
        free(fifo_node); //avoid mem leak

        return -3000;
      }

      /*victim in ram*/
      int ram_vicpgn = PAGING_FPN(*fifo_node->pgd_pgn);

      /* Copy victim frame to swap */
      __swap_cp_page(caller->mram, ram_vicpgn, caller->active_mswp, swpfpn);

      /* Update page table */
      pte_set_swap(fifo_node->pgd_pgn, 0, swpfpn);
      free(fifo_node); //avoid mem leak

      fpn = ram_vicpgn;
    }
```

```
43.
44.     /*Collect the freeframe*/
45.     //Create new framepage node (which is the freefp we just collected) with node
    's fpn = fpn
46.     struct framephy_struct *frm_node = malloc(sizeof(struct framephy_struct));
47.     frm_node->fpn = fpn;
48.
49.     //Connect the new framepage node to the frm_lst
50.     frm_node->fp_next = *frm_lst;
51.     *frm_lst = frm_node;
52.   }
53.
54.   return 0;
55. }
```

Mục tiêu hàm này là thu thập các free frames trong RAM lưu thành danh sách và lưu bởi biến tham chiếu frm_lst. Trong lúc thu thập free frames trong RAM ta sẽ dính phải trường hợp là lượng frames trống đã hết. Đến đây ta sẽ giải thích cho việc tại sao phải khai báo thuộc tính fifo_pgn trong **mram** mà không phải trong **mmap** nữa. Vì mỗi process sẽ có một **mmap** riêng chỉ xài chung mram, nếu khai báo trong **mmap** thì với process mới (chưa alloc gì và danh sách fifo rỗng) khi tìm kiếm victim để thực hiện việc thay trang khi frame trống trong RAM đã hết (vì bị các process khác sử dụng) thì sẽ không tìm thấy gì cả và thất bại trong việc thu thập frame trống.

Giải pháp cho việc này là khai báo fifo_pgn trong **mram.** Và khi khi tìm victim nó sẽ thay thế thẳng victim của một process khác và cập nhật luôn cả page directory của victim của process đó thông qua thuộc tính con trỏ pgd_pgn trỏ tới đúng địa chỉ page directory cần cập nhật.

Nếu tìm không thấy victim thì sẽ trả về -1 (ngoại lệ OOR), còn nếu trong SWAP hết cả free frame để swap victim qua thì trả về -3000 (ngoại lệ OOM).

- Hàm **vmap_page_range():**

```
1.  int vmap_page_range(struct pcb_t *caller, // process call
2.                              int addr, // start address which is aligned to pagesz

3.                              int pgnum, // num of mapping page
4.             struct framephy_struct *frames,// list of the mapped frames
5.          struct vm_rg_struct *ret_rg)// return mapped region, the real mapped fp
6.  {                                     // no guarantee all given pages are mapped

7.
8.    struct framephy_struct *fpit = frames;
9.
10.   int pgit = 0;
11.   int pgn = PAGING_PGN(addr);
12.
13.   ret_rg->rg_end = ret_rg->rg_start = addr;
14. // at least the very first space is usable
15.
16.   /* TODO map range of frame to address space
17.    *      [addr to addr + pgnum*PAGING_PAGESZ
18.    *      in page table caller->mm->pgd[]
19.    */
20.
21.  while(fpit != NULL){
22.
23.     uint32_t pte = 0;//caller->mm->pgd[pgn + pgit];
24.     pte_set_fpn(&pte, fpit->fpn); //update page table
25.     caller->mm->pgd[pgn + pgit] = pte;
26.
27.     enlist_pgn_node(caller->mram, pgn + pgit, &caller->mm->pgd[pgn + pgit]);
28. //Enqueue new usage page
29.
30.     // Update the range.
31.     ret_rg->rg_end += PAGING_PAGESZ;
32.     fpit = fpit->fp_next;
```

```
    pgit++;
  }

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

 return 0;
}
```

Hàm này chỉ đơn giản là lấy frame page number của các frames trống vừa thu thập được cập nhật cho page directory của các pages bắt đầu từ addr thông qua hàm **pte_set_fpn** và nhét các pages đấy vào danh sách FIFO.

3.2.2 Test

Sử dụng make file: make mem

3.2.2.1 Test 1

- Input:
Process: p0s

```
1 14
calc
alloc 300 0
alloc 300 4
free 0
alloc 100 1
write 100 1 20
read 1 20 20
write 102 2 20
read 2 20 20
write 103 3 20
read 3 20 20
calc
free 4
calc
```

Configuration: paging_4KB

```
4096 16777216 0 0 0
```

- Output:
./mem p0s paging_4KB

```
[PID: 1]----After enlist----
print_list_rg:
rg[0->300]

[PID: 1] write region=1 offset=20 value=100
print_list_using_fpn_RAM:
ra[80000002]-
ra[80000000]-
ra[80000001]-

print_pgtbl: 0 - 768
00000000: 80000001
00000004: 80000000
00000008: 80000002
Memory content [addr]-[offset,value]: | max size = 4096

[PID: 1] read region=1 offset=20 value=100
print_list_using_fpn_RAM:
ra[80000002]-
```

```
ra[80000000]-
ra[80000001]-

print_pgtbl: 0 - 768
00000000: 80000001
00000004: 80000000
00000008: 80000002
Memory content [addr]-[offset,value]:
                [80000001]-[20,100]| max size = 4096

[PID: 1] write region=2 offset=20 value=102
print_list_using_fpn_RAM:
ra[80000002]-
ra[80000000]-
ra[80000001]-

print_pgtbl: 0 - 768
00000000: 80000001
00000004: 80000000
00000008: 80000002
Memory content [addr]-[offset,value]:
                [80000001]-[20,100]| max size = 4096

[PID: 1] read region=2 offset=20: INVALID REGION OFFSET
print_list_using_fpn_RAM:
ra[80000002]-
ra[80000000]-
ra[80000001]-

print_pgtbl: 0 - 768
00000000: 80000001
00000004: 80000000
00000008: 80000002
Memory content [addr]-[offset,value]:
                [80000001]-[20,100]| max size = 4096

[PID: 1] write region=3 offset=20 value=103
print_list_using_fpn_RAM:
ra[80000002]-
ra[80000000]-
ra[80000001]-

print_pgtbl: 0 - 768
00000000: 80000001
00000004: 80000000
00000008: 80000002
Memory content [addr]-[offset,value]:
                [80000001]-[20,100]| max size = 4096

[PID: 1] read region=3 offset=20: INVALID REGION OFFSET
print_list_using_fpn_RAM:
ra[80000002]-
ra[80000000]-
ra[80000001]-

print_pgtbl: 0 - 768
00000000: 80000001
00000004: 80000000
00000008: 80000002
Memory content [addr]-[offset,value]:
                [80000001]-[20,100]| max size = 4096

[PID: 1]----After enlist----
print_list_rg:
rg[100->600]
```

- Giải thích input-output:

+ Với file input p0s như trên thì ta sẽ chỉ tập trung vào các lệnh liên quan đến cấp phát, đọc – ghi, giải phóng bộ nhớ.
+ File cấu hình (configuration) như trên thì kích thước RAM sẽ là 4KB, SWAP[0] sẽ là 16MB các SWAP còn lại có kích thước là bằng 0 và kích thước mỗi trang là 256B. Nên trong RAM ban đầu sẽ có 4KB/256B = 16 trang trống và trong SWAP[0] có 16MB/256B = 65536 trang trống.
+ Phân tích:
  - Lệnh 1: calc (Lệnh tính toán)
  - Lệnh 2: alloc 300 0
    Lúc này region 0 sẽ được cấp phát một lượng vùng nhớ có kích thước là 300B như vậy RAM sẽ cấp phát tương ứng là 2 trang (2x256B) và sẽ map với vùng nhớ ảo tương ứng trang 0-1 có địa chỉ vật lí là 80000001-80000000 . Với vùng nhớ ảo, sbrk = 300B, bắt đầu tại 0 và kết thúc tại 512B.
  - Lệnh 3: alloc 300 4
    Region 4 yêu cầu 300B, mà khoảng cách giữa sbrk và vm_end là 512B – 300B = 212B, nhỏ hơn so với yêu cầu là 88B vậy nên RAM sẽ cấp phát thêm 1 trang nữa (256B) và map trang này với địa chỉ vật lí tương ứng là 80000002. Với vùng nhớ ảo, sbrk = 600B, bắt đầu tại 0 và kết thúc tại 768B.
  - Lệnh 4: free 0
    Region 0 sẽ được giải phóng và đưa vào danh sách liên kết các vùng đã được giải phóng vùng nhớ ảo và in ra dánh sách ở file từ dòng 1 – 3.
  - Lệnh 5: alloc 100 1
    Region 1 yêu cầu 100B và lần này thì trong danh sách các vùng ảo free có vùng vừa được thêm vào ở lệnh 4 có kích thước 0- >300B đủ để vùng 1 thêm vào, sau khi xong lệnh này danh sách free hiện tại có 1 region 100->300B.
  - Lệnh 6 - 7: write 100 1 20 – read 1 20 20
    Region 1 được yêu cầu viết vào giá trị 100 với offset của vùng là 20 nên địa chỉ viết vào của vùng ảo là 20. Mà region 1 hiện tại đang map với địa chỉ trong RAM từ 80000001 tức là tại địa chỉ vật lí là 256 mà offset là 20 nên địa chỉ cần viết trong RAM là 276. Ta có thể quan sát file output của quá trình này từ dòng 5 – 28.
  - Lệnh 8 – 9: write 102 2 20 – read 2 20 20
    Hai lệnh này yêu cầu ghi và đọc ở region 2 mà region 2 chưa được cấp phát gì cả nên thao tác write sẽ không ghi gì vào bộ nhớ vật lí và khi yêu cầu đọc thì sẽ in ra giá trị đọc được là "INVALID REGION OFFSET". Theo dõi file output từ dòng 30 -54.
  - Lệnh 10 – 11: write 103 3 20 – read 3 20 20
    Trạng thái của hai lệnh này giải thích tương tự lệnh 8 – 9. Theo dõi từ dòng 56 – 80.
  - Lệnh 12: calc (Lệnh tính toán)
  - Lệnh 13: free 4
    Region 4 [300->600] được yêu cầu giải phóng. Danh sách liên kết các vùng đã giải phóng hiện tại chỉ có 1 node [100->300]. Khi đưa region 4 này vào thì danh sách sẽ thành [100->300]->[300->600] nhưng với cơ chế tối ưu alloc đã được minh họa ở trang 25 thì danh sách này sẽ là [100->600]. Theo dõi file output ở dòng 82 – 84.
  - Lệnh 14: calc (Lệnh tính toán)

+ Trong output ở trên có in ra địa chỉ các frames đang được sử dụng ở các dòng bắt đầu bằng "`print_list_using_fpn_RAM:`", và pages nào tương ứng với frames nào ở các dòng bắt đầu bằng "`print_pgtbl:`" cho dễ dàng quan sát.

3.2.2.2 Test 2

- Input:
Process: m0s

```
1. 1 7
```

```
alloc 300 0
alloc 100 1
free 0
alloc 100 2
write 102 1 20
write 1 0 40
read 0 40 40
```

Configuration: paging_1KB

```
1024 16777216 0 0 0
```

- Output:
./mem m0s paging_1KB

```
[PID: 1]----After enlist----
print_list_rg:
rg[0->300]

[PID: 1] write region=1 offset=20 value=102
print_list_using_fpn_RAM:
ra[80000000]-
ra[80000001]-

print_pgtbl: 0 - 512
00000000: 80000001
00000004: 80000000
Memory content [addr]-[offset,value]: | max size = 1024

[PID: 1] write region=0 offset=40 value=1
print_list_using_fpn_RAM:
ra[80000000]-
ra[80000001]-

print_pgtbl: 0 - 512
00000000: 80000001
00000004: 80000000
Memory content [addr]-[offset,value]:
                [80000000]-[64,102]| max size = 1024

[PID: 1] read region=0 offset=40 value=1
print_list_using_fpn_RAM:
ra[80000000]-
ra[80000001]-

print_pgtbl: 0 - 512
00000000: 80000001
00000004: 80000000
Memory content [addr]-[offset,value]:
                [80000000]-[64,102]
                [80000001]-[40,1]| max size = 1024
```

3.2.3 Answer the question

3.2.3.1 Question 1

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

Answer:

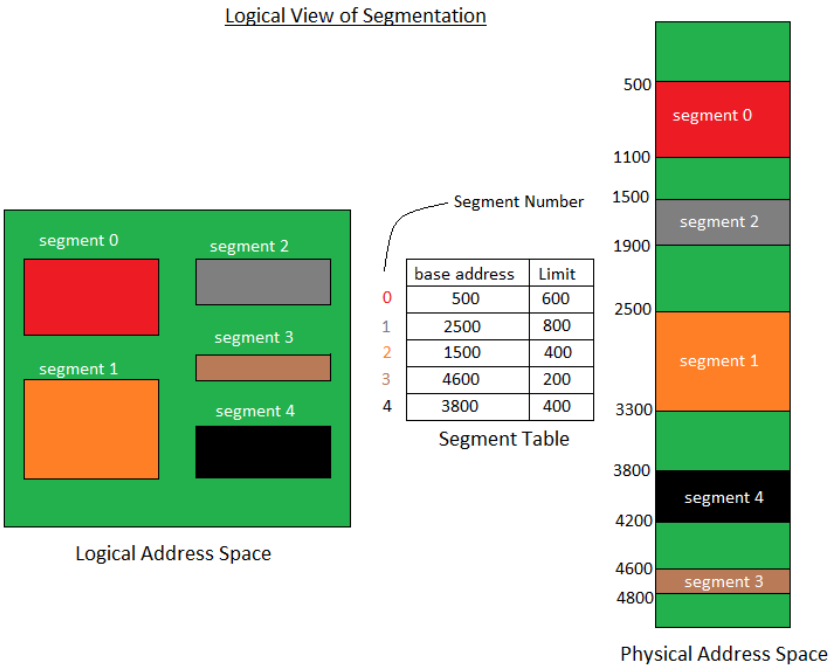
Trong hệ điều hành, chương trình và dữ liệu được lưu trữ trong bộ nhớ. Nhưng không phải toàn bộ bộ nhớ đều có thể được sử dụng cho mỗi chương trình. Thay vào đó, bộ nhớ được chia thành nhiều phân đoạn khác nhau để phục vụ cho mục đích khác nhau.

Ví dụ, một hệ điều hành có thể chia bộ nhớ thành các phân đoạn sau:

- Code segment: Lưu trữ mã máy của chương trình
- Data segment: Lưu trữ dữ liệu được sử dụng bởi chương trình
- Heap segment: Lưu trữ dữ liệu động được tạo ra trong khi chương trình đang chạy
- Stack segment: Lưu trữ các biến cục bộ và các giá trị của hàm được gọi trong chương trình

Với việc chia bộ nhớ thành các phân đoạn như vậy, chương trình có thể được quản lý tốt hơn và giảm thiểu việc tràn bộ nhớ (memory overflow) hoặc xung đột bộ nhớ (memory conflicts).

Hơn nữa, việc phân chia bộ nhớ thành các phân đoạn cho phép hệ điều hành cung cấp quyền truy cập khác nhau cho mỗi phân đoạn. Ví dụ, chương trình không được phép ghi đè vào mã máy của chính nó trong Code segment, nhưng có thể ghi vào Data segment.



Ưu điểm khác của việc sử dụng nhiều đoạn bộ nhớ là nó cho phép quản lý bộ nhớ hiệu quả hơn. Các đoạn khác nhau có thể được phân bổ và giải phóng độc lập, cho phép quản lý bộ nhớ linh hoạt hơn, phân bổ bộ nhớ chi tiết hơn. Ví dụ, trong một chương trình yêu cầu cả stack và heap, stack có thể được phân bổ trong một đoạn và heap trong một đoạn khác. Điều này cho phép quản lý riêng biệt hai vùng nhớ này, có thể cải thiện hiệu suất tổng thể và giảm nguy cơ lỗi liên quan đến bộ nhớ.

Nhiều phân đoạn còn cho phép chia sẻ các phân đoạn bộ nhớ giữa các quy trình. Điều này có thể hữu ích cho giao tiếp giữa các quá trình hoặc để chia sẻ thư viện mã.

Ngoài ra, nhiều đoạn bộ nhớ cũng cung cấp cải tiến về bảo mật và ổn định hệ thống. Bằng cách cô lập các đoạn khác nhau, có thể ngăn chặn không cho 1 tiến trình truy cập hoặc sửa đổi trái phép vào các thành phần hệ thống quan trọng hay bất kì phân đoạn bộ nhớ của tiến trình khác.

Tổng thể, việc sử dụng nhiều đoạn bộ nhớ có thể cung cấp nhiều lợi ích, bao gồm quản lý bộ nhớ hiệu quả hơn, cải thiện hiệu suất, giảm nguy cơ lỗi liên quan đến bộ nhớ và cải thiện bảo mật. Tuy nhiên, nó cũng đòi hỏi các thuật toán quản lý bộ nhớ phức tạp hơn và có thể tăng chi phí liên quan đến phân bổ và giải phóng bộ nhớ.

3.2.3.1 Question 2

What will happen if we divide the address to more than 2-levels in the paging memory management system?

Answer:

Trong hệ thống quản lý bộ nhớ phân trang, chúng ta có thể chia địa chỉ logic thành nhiều cấp để quản lý bộ nhớ hợp lý hơn. Các mục của bảng trang cấp 1 là con trỏ tới bảng trang cấp 2 và các mục của bảng trang cấp 2 là con trỏ tới bảng trang cấp 3, v.v. Các mục của bảng trang mức cuối cùng lưu trữ thông tin khung thực tế. Cấp 1 chứa

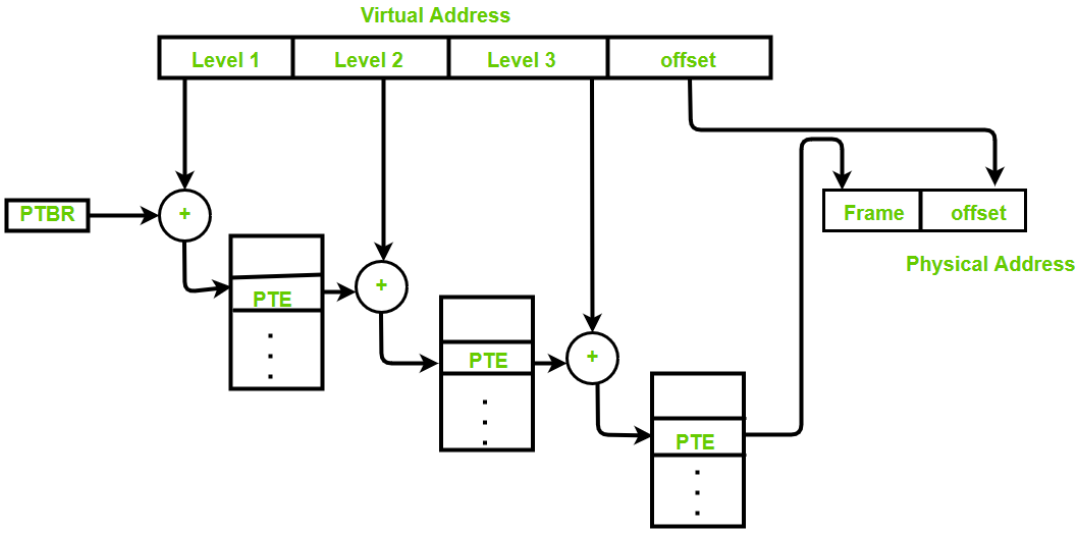
một bảng trang đơn và địa chỉ của bảng đó được lưu trữ trong PTBR (Thanh ghi cơ sở bảng trang).

Thông thường, hệ thống quản lý bộ nhớ phân trang được thiết kế với 2 hoặc 3 cấp địa chỉ, tuy nhiên, nếu chúng ta chia địa chỉ thành nhiều hơn 2 cấp, sẽ có một số ảnh hưởng đến hiệu suất của hệ thống.

Khi chia địa chỉ thành nhiều hơn 2 cấp, số bit được sử dụng cho mỗi cấp sẽ giảm đi. Điều này có thể dẫn đến hiện tượng internal fragmentation (phân mảnh nội) trong bộ nhớ, nghĩa là một số phần của frame sẽ không được sử dụng, điều này có thể làm giảm hiệu suất tổng thể của hệ thống. Ví dụ, nếu chúng ta chia địa chỉ thành 4 cấp, mỗi cấp chỉ sử dụng 4 bit, khi đó mỗi frame chỉ được sử dụng 16 byte thay vì 4KB như ở 2 cấp, dẫn đến lãng phí bộ nhớ.

Việc phân trang đa cấp còn tăng thêm độ phức tạp cho hệ thống quản lý bộ nhớ, điều này có thể gây khó khăn hơn cho việc thiết kế, triển khai và gỡ lỗi.

Mặc dù phân trang đa cấp có thể giảm chi phí bộ nhớ liên quan đến bảng trang, nhưng nó cũng có thể tăng chi phí liên quan đến tra cứu bảng trang. Điều này là do mỗi cấp độ phải được duyệt qua để tìm mục nhập bảng trang mong muốn.



3 Level paging system

Tuy nhiên, việc chia địa chỉ thành nhiều cấp cũng có một số lợi ích. Khi sử dụng nhiều cấp địa chỉ, hệ thống sẽ có thể quản lý bộ nhớ tốt hơn và sử dụng tài nguyên bộ nhớ hiệu quả hơn. Ngoài ra, còn giúp cứu bảng trang nhanh hơn với số lượng mục nhỏ hơn trên mỗi cấp, sẽ mất ít thời gian hơn để thực hiện tra cứu bảng trang. Điều này có thể dẫn đến hiệu suất hệ thống tổng thể nhanh hơn.

Tóm lại, việc chia địa chỉ thành nhiều cấp đều có những mặt lợi và hạn chế riêng. Cần xem xét nhu cầu cũng như mục tiêu hiện thực để lựa chọn số cấp phân chia địa chỉ vùng nhớ phù hợp.

3.2.3.3 Question 3

What is the advantage and disadvantage of segmentation with paging?

Answer:

Phân đoạn (segmentation) kết hợp với phân trang (paging) là một phương pháp quản lý bộ nhớ trong hệ điều hành. Phương pháp này kết hợp cả ưu điểm của phân đoạn và phân trang. Tuy nhiên, nó cũng có những ưu điểm và nhược điểm riêng.

Ưu điểm:

- Phân đoạn cho phép quản lý các khối bộ nhớ lớn hơn, còn phân trang cho phép quản lý các khối nhỏ hơn. Kết hợp cả hai giúp giảm thiểu hao phí bộ nhớ, sử dụng bộ nhớ hiệu quả, khắc phục việc bảng phân trang có kích thước quá lớn
- Chống phân mảnh ngoại

- Dễ cấp phát phát bộ nhớ vật lý
- Bảo vệ dữ liệu và chia sẻ phân đoạn
- Việc chia sẻ dữ liệu giữa các tiến trình linh hoạt hơn qua chia sẻ 1 page hay 1 segment table
- Khi kết hợp phân đoạn và phân trang, mỗi phân đoạn và trang sẽ được gán một quyền truy cập khác nhau. Điều này giúp tăng tính bảo mật của hệ thống bằng cách ngăn chặn việc truy cập bất hợp pháp đến các vùng nhớ quan trọng.

Nhược điểm:

- Khó hiện thực
- Còn tồn tại phân mảnh nội
- Hiệu suất truy cập không cao
- Size của process bị giới hạn bởi size của physical memory
- Khó có thể bảo trì nhiều process cùng lúc trong memory, và như vậy khó nâng cao mức độ multiprogramming của OS.
- Nếu segment table không được sử dụng thì không còn lưu page table. Page table còn nhiều memory space, nên sẽ không tốt cho system có RAM nhỏ.

-> Giảm memory space cần thiết khi lưu page table.

3.3 Module 3 – Put it all together

3.3.1 Test

- Input: Sử dụng make file: make all

Configuration: os_1_mlq_paging_small_1K

```
2 4 8
1024 16777216 0 0 0
1 p0s 130
2 s3 39
4 m1s 15
6 s2 120
7 m0s 120
9 p1s 15
11 s0 38
16 s1 0
```

Process:

Process	p0s	s3	m1s	s2	m0s	p1s	s0	s1
Description	1 14 calc alloc 300 0 alloc 300 4 free 0 alloc 100 1 write 100 1 20 read 1 20 20 write 102 2 20 read 2 20 20 write 103 3 20 read 3 20 20 calc free 4 calc	7 17 calc calc calc calc calc calc calc calc calc calc calc calc calc calc calc calc calc calc	1 6 alloc 300 0 alloc 100 1 free 0 alloc 100 2 free 2 free 1	20 13 calc calc calc calc calc calc calc calc calc calc calc calc calc	1 7 alloc 300 0 alloc 100 1 free 0 alloc 100 2 write 102 1 20 write 1 0 40 read 0 40 40	1 11 calc calc calc calc calc calc calc calc calc calc	12 15 calc calc calc calc calc calc calc calc calc calc calc calc calc calc calc calc	20 7 calc calc calc calc calc calc calc calc

- Output: ./os os_1_mlq_paging_small_1K

```
Time slot   0
```

```
ld_routine
        Loaded a process at input/proc/p0s, PID: 1 PRIO: 130 at time 1
        CPU 1: Dispatched process  1 at time 1
Time slot   1
        Loaded a process at input/proc/s3, PID: 2 PRIO: 39 at time 2
Time slot   2
        CPU 0: Dispatched process  2 at time 2
        CPU 1: Put process  1 to run queue at time 3
        CPU 1: Dispatched process  1 at time 3
Time slot   3
        Loaded a process at input/proc/m1s, PID: 3 PRIO: 15 at time 4
Time slot   4
        CPU 0: Put process  2 to run queue at time 4
        CPU 0: Dispatched process  3 at time 4
[PID: 1]----After enlist----
print_list_rg:
rg[0->300]

-----[PID: 3] victim_OOR: 80000001
        CPU 3: Dispatched process  2 at time 5
Time slot   5
        CPU 1: Put process  1 to run queue at time 5
        CPU 1: Dispatched process  1 at time 5
        Loaded a process at input/proc/s2, PID: 4 PRIO: 120 at time 6
Time slot   6
        CPU 0: Put process  3 to run queue at time 6
        CPU 0: Dispatched process  3 at time 6
[PID: 3]----After enlist----
print_list_rg:
rg[0->300]

[PID: 1] write region=1 offset=20 value=100
print_list_using_fpn_RAM:
ra[80000003]-
ra[80000001]-
ra[80000002]-
ra[80000000]-

print_pgtbl: 0 - 768
00000000: 40000000
00000004: 80000000
00000008: 80000002
Memory content [addr]-[offset,value]: | max size = 1024

-----[PID: 1] PAGE FAULT----- target page in swap: 40000000
-----[PID: 1] PAGE FAULT----- found victim: 80000000
-----[PID: 1] PAGE FAULT----- victim_present: 40000020
-----[PID: 1] PAGE FAULT----- target_put: 80000000
Time slot   7
        Loaded a process at input/proc/m0s, PID: 5 PRIO: 120 at time 7
        CPU 3: Put process  2 to run queue at time 7
        CPU 3: Dispatched process  2 at time 7
        CPU 1: Put process  1 to run queue at time 7
        CPU 1: Dispatched process  4 at time 7
        CPU 2: Dispatched process  5 at time 7
-----[PID: 5] victim_OOR: 80000002
-----[PID: 5] victim_OOR: 80000001
Time slot   8
        CPU 0: Put process  3 to run queue at time 8
        CPU 0: Dispatched process  3 at time 8
[PID: 3]----After enlist----
print_list_rg:
rg[0->300]

        Loaded a process at input/proc/p1s, PID: 6 PRIO: 15 at time 9
        CPU 3: Put process  2 to run queue at time 9
```

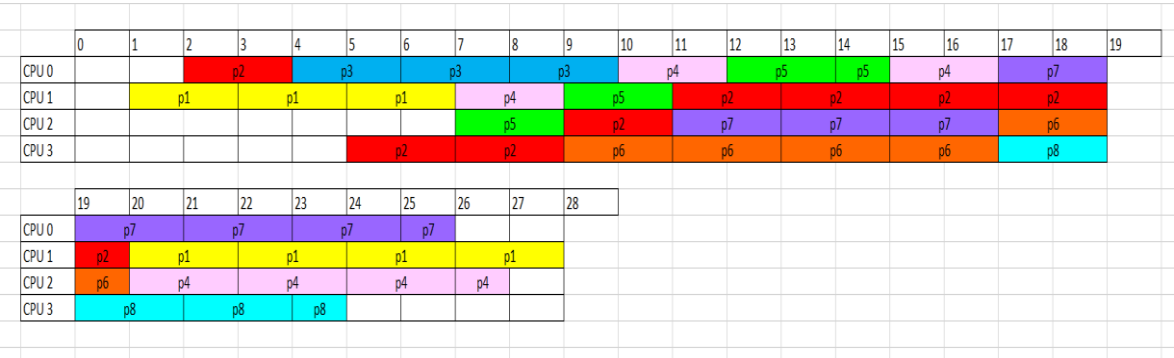
```
        CPU 3: Dispatched process  6 at time 9
        CPU 2: Put process  5 to run queue at time 9
        CPU 2: Dispatched process  2 at time 9
        CPU 1: Put process  4 to run queue at time 9
        CPU 1: Dispatched process  5 at time 9
[PID: 5]----After enlist----
print_list_rg:
rg[0->300]

Time slot   9
[PID: 3]----After enlist----
print_list_rg:
rg[0->400]

Time slot  10
        CPU 0: Processed  3 has finished at time 10
        CPU 0: Dispatched process  4 at time 10
        Loaded a process at input/proc/s0, PID: 7 PRIO: 38 at time 11
Time slot  11
        CPU 2: Put process  2 to run queue at time 11
        CPU 2: Dispatched process  7 at time 11
        CPU 3: Put process  6 to run queue at time 11
        CPU 3: Dispatched process  6 at time 11
        CPU 1: Put process  5 to run queue at time 11
        CPU 1: Dispatched process  2 at time 11
Time slot  12
        CPU 0: Put process  4 to run queue at time 12
        CPU 0: Dispatched process  5 at time 12
[PID: 5] write region=1 offset=20 value=102
print_list_using_fpn_RAM:
ra[80000002]-
ra[80000001]-
      ra[80000000]-
      ra[80000003]-

      print_pgtbl: 0 - 512
      00000000: 80000001
      00000004: 80000002
      Memory content [addr]-[offset,value]:
                     [80000000]-[20,100]| max size = 1024

              CPU 3: Put process  6 to run queue at time 13
              CPU 1: Put process  2 to run queue at time 13
              CPU 1: Dispatched process  2 at time 13
      Time slot  13
              CPU 2: Put process  7 to run queue at time 13
              CPU 2: Dispatched process  7 at time 13
              CPU 3: Dispatched process  6 at time 13
      [PID: 5] write region=0 offset=40 value=1
      print_list_using_fpn_RAM:
      ra[80000002]-
      ra[80000001]-
      ra[80000000]-
      ra[80000003]-

      print_pgtbl: 0 - 512
      00000000: 80000001
      00000004: 80000002
      Memory content [addr]-[offset,value]:
                     [80000000]-[20,100]
                     [80000002]-[64,102]| max size = 1024

      Time slot  14
              CPU 0: Put process  5 to run queue at time 14
              CPU 0: Dispatched process  5 at time 14
      [PID: 5] read region=0 offset=40 value=1
```

```
print_list_using_fpn_RAM:
ra[80000002]-
ra[80000001]-
ra[80000000]-
ra[80000003]-

print_pgtbl: 0 - 512
00000000: 80000001
00000004: 80000002
Memory content [addr]-[offset,value]:
                [80000000]-[20,100]
                [80000001]-[40,1]
                [80000002]-[64,102]| max size = 1024

        CPU 3: Put process  6 to run queue at time 15
        CPU 1: Put process  2 to run queue at time 15
        CPU 1: Dispatched process  2 at time 15
Time slot  15
        CPU 2: Put process  7 to run queue at time 15
        CPU 2: Dispatched process  7 at time 15
        CPU 3: Dispatched process  6 at time 15
        CPU 0: Processed  5 has finished at time 15
        CPU 0: Dispatched process  4 at time 15
        Loaded a process at input/proc/s1, PID: 8 PRIO: 0 at time 16
Time slot  16
        CPU 3: Put process  6 to run queue at time 17
        CPU 3: Dispatched process  8 at time 17
        CPU 2: Put process  7 to run queue at time 17
        CPU 2: Dispatched process  6 at time 17
        CPU 1: Put process  2 to run queue at time 17
        CPU 0: Put process  4 to run queue at time 17
        CPU 0: Dispatched process  7 at time 17
        CPU 1: Dispatched process  2 at time 17
Time slot  17
Time slot  18
        CPU 3: Put process  8 to run queue at time 19
        CPU 3: Dispatched process  8 at time 19
        CPU 1: Put process  2 to run queue at time 19
        CPU 1: Dispatched process  2 at time 19
        CPU 2: Put process  6 to run queue at time 19
        CPU 2: Dispatched process  6 at time 19
        CPU 0: Put process  7 to run queue at time 19
        CPU 0: Dispatched process  7 at time 19
Time slot  19
        CPU 2: Processed  6 has finished at time 20
        CPU 2: Dispatched process  4 at time 20
        CPU 1: Processed  2 has finished at time 20
        CPU 1: Dispatched process  1 at time 20
[PID: 1] read region=1 offset=20 value=100
print_list_using_fpn_RAM:
ra[80000002]-
ra[80000001]-
ra[80000000]-
ra[80000003]-

print_pgtbl: 0 - 768
00000000: 80000000
00000004: 40000020
00000008: 40000000
Memory content [addr]-[offset,value]:
                [80000000]-[20,100]
                [80000001]-[40,1]
                [80000002]-[64,102]| max size = 1024

Time slot  20
        CPU 3: Put process  8 to run queue at time 21
```

```
200.                CPU 3: Dispatched process  8 at time 21
201.        [PID: 1] write region=2 offset=20 value=102
202.        print_list_using_fpn_RAM:
203.        ra[80000002]-
204.        ra[80000001]-
205.        ra[80000000]-
206.        ra[80000003]-
207.
208.        print_pgtbl: 0 - 768
209.        00000000: 80000000
210.        00000004: 40000020
211.        00000008: 40000000
212.        Memory content [addr]-[offset,value]:
213.                       [80000000]-[20,100]
214.                       [80000001]-[40,1]
215.                       [80000002]-[64,102]| max size = 1024
216.
217.        Time slot  21
218.                CPU 0: Put process  7 to run queue at time 21
219.                CPU 0: Dispatched process  7 at time 21
220.                CPU 1: Put process  1 to run queue at time 22
221.                CPU 2: Put process  4 to run queue at time 22
222.                CPU 2: Dispatched process  4 at time 22
223.        Time slot  22
224.                CPU 1: Dispatched process  1 at time 22
225.        [PID: 1] read region=2 offset=20: INVALID REGION OFFSET
226.        print_list_using_fpn_RAM:
227.        ra[80000002]-
228.        ra[80000001]-
229.        ra[80000000]-
230.        ra[80000003]-
231.
232.        print_pgtbl: 0 - 768
233.        00000000: 80000000
234.        00000004: 40000020
235.        00000008: 40000000
236.        Memory content [addr]-[offset,value]:
237.                       [80000000]-[20,100]
238.                       [80000001]-[40,1]
239.                       [80000002]-[64,102]| max size = 1024
240.
241.                CPU 3: Put process  8 to run queue at time 23
242.                CPU 0: Put process  7 to run queue at time 23
243.                CPU 0: Dispatched process  7 at time 23
244.        [PID: 1] write region=3 offset=20 value=103
245.        print_list_using_fpn_RAM:
246.        ra[80000002]-
247.        ra[80000001]-
248.        ra[80000000]-
249.        ra[80000003]-
250.
251.        print_pgtbl: 0 - 768
252.        00000000: 80000000
253.        00000004: 40000020
254.        00000008: 40000000
255.        Memory content [addr]-[offset,value]:
256.                       [80000000]-[20,100]
257.                       [80000001]-[40,1]
258.                       [80000002]-[64,102]| max size = 1024
259.
260.        Time slot  23
261.                CPU 3: Dispatched process  8 at time 23
262.        Time slot  24
263.                CPU 3: Processed  8 has finished at time 24
264.                CPU 3 stopped at time 24
265.                CPU 2: Put process  4 to run queue at time 24
```

```
266.                CPU 2: Dispatched process  4 at time 24
267.                CPU 1: Put process  1 to run queue at time 24
268.                CPU 1: Dispatched process  1 at time 24
269.        [PID: 1] read region=3 offset=20: INVALID REGION OFFSET
270.        print_list_using_fpn_RAM:
271.        ra[80000002]-
272.        ra[80000001]-
273.        ra[80000000]-
274.        ra[80000003]-
275.
276.        print_pgtbl: 0 - 768
277.        00000000: 80000000
278.        00000004: 40000020
279.        00000008: 40000000
280.        Memory content [addr]-[offset,value]:
281.                       [80000000]-[20,100]
282.                       [80000001]-[40,1]
283.                       [80000002]-[64,102]| max size = 1024
284.
285.        Time slot  25
286.                CPU 0: Put process  7 to run queue at time 25
287.                CPU 0: Dispatched process  7 at time 25
288.        Time slot  26
289.                CPU 0: Processed  7 has finished at time 26
290.                CPU 0 stopped at time 26
291.                CPU 2: Put process  4 to run queue at time 26
292.                CPU 2: Dispatched process  4 at time 26
293.                CPU 1: Put process  1 to run queue at time 26
294.                CPU 1: Dispatched process  1 at time 26
295.        [PID: 1]----After enlist----
296.        print_list_rg:
297.        rg[100->600]
298.
299.                CPU 2: Processed  4 has finished at time 27
300.                CPU 2 stopped at time 27
301.        Time slot  27
302.        Time slot  28
303.                CPU 1: Processed  1 has finished at time 28
304.                CPU 1 stopped at time 28
```

- Sơ đồ Gantt



3.3.2 Answer the question

Question: What will happen if the synchronization is not handled in your simple OS? Illustrate by example the problem of your simple OS if you have any.

Answer:

Nếu đồng bộ hóa không được xử lý đúng cách trong hệ điều hành, nó có thể gây ra một loạt vấn đề, bao gồm race condition, deadlock và priority inversion. Những vấn đề này có thể khiến hệ thống hoạt động không đoán được, treo máy hoặc gây ra sự cố.

Ví dụ, giả sử rằng trong hệ điều hành đơn giản của tôi, hai quy trình cần truy cập vào một tài nguyên chia sẻ, chẳng hạn như một máy in. Nếu đồng bộ hóa không

được xử lý đúng cách, cả hai quy trình có thể cố gắng truy cập vào máy in đồng thời, dẫn đến race condition. Tình huống này có thể gây ra máy in hoạt động không đúng hoặc sản xuất kết quả không chính xác, vì các quy trình có thể ghi đè lên kết quả của nhau hoặc can thiệp vào các hoạt động của nhau.

Một ví dụ khác là vấn đề deadlock. Giả sử rằng hai quy trình cần truy cập vào hai tài nguyên, mỗi tài nguyên lại được giữ bởi quy trình kia. Nếu đồng bộ hóa không được xử lý đúng cách, các quy trình có thể vào trạng thái deadlock, nơi chúng đang chờ đợi nhau để giải phóng tài nguyên mà chúng cần. Tình huống này có thể làm treo hệ thống hoặc khiến nó không đáp ứng.

Cuối cùng, vấn đề priority inversion có thể xảy ra khi một quy trình ưu tiên thấp giữ một tài nguyên mà quy trình ưu tiên cao cần. Nếu đồng bộ hóa không được xử lý đúng cách, quy trình ưu tiên cao có thể bị chặn, đang chờ quy trình ưu tiên thấp giải phóng tài nguyên. Tình huống này có thể gây ra độ trễ và giảm hiệu suất tổng thể của hệ thống.

Để tránh những vấn đề này, các cơ chế đồng bộ hóa đúng cách, chẳng hạn như locks, semaphore và monitor, cần được thực hiện trong hệ điều hành. Những cơ chế này đảm bảo rằng các quy trình truy cập vào tài nguyên chia sẻ một cách kiểm soát và đồng bộ, ngăn ngừa race condition, deadlock và priority inversion.

4 PARTICIPANT AND PROJECT OUTPUTS

4.1 Role and contribution of each member 1

Nhóm trưởng: Võ Tấn Hưng
Các công việc được các thành viên hoàn thành như sau:
- Nguyễn Đức An: thiết kế và hiện thực phần định thời
- Võ Tấn Hưng, Đào Duy Long: thiết kế và hiện thực phần quản lý bộ nhớ
- Lê Đình Huy: viết báo cáo

4.2 Project output

Nhóm chúng em cơ bản đã giả lập được các thành phần lớn trong một hệ điều hành đơn giản. Ví dụ: định thời (scheduler), đồng bộ (synchronization), quan hệ giữa bộ nhớ vật lý (physical memory) và bộ nhớ ảo (virtual memory).

Ngoài ra, sau bài tập lớn, chúng em hiểu được phần nào đó nguyên lý hoạt động của một hệ điều hành cơ bản, hiểu vai trò và ý nghĩa của các mô-đun (key modules) ở hệ điều hành cũng như nó hoạt động như thế nào. Từ đó, có thể nắm được khái quát nội dung được học trên lớp lý thuyết cũng như học phần thí nghiệm thí nghiệm để áp dụng vào thực tiễn.

4.3 Project outcome

L.O.1	**Describe on how to apply fundamental knowledge of computing and mathematics in an operating system**
L.O.1.1	Define the functionality and structures that a modern operating system must deliver to meet a particular need.
L.O.1.2	Explain virtual memory and its realisation in hardware and software.
L.O.2	**Able to report the tradeoffs between the performance and the resource and technology constraints in a design of an operating system.**
L.O.2.1	Compare and contrast common algorithms used for both preemptive and non-preemptive scheduling of tasks in operating systems

L.O.2.2	Compare and contrast different approaches to file organisation, recognizing the strengths and weaknesses of each.
L.O.3	**Describe main operating system concepts and their aspects that are useful to realise concurrent systems and describe the benefits of each.**
L.O.3.1	Compare and contrast different methods for process synchronisation.

ANNEXES

1. Terms of Reference

- Linux Documentation: http://linux.die.net
- Geeksforgeeks: https://www.geeksforgeeks.org/
- Operating System Concepts, Peter Baer Galvin, Greg Gagne, Abraham Silberschatz, 10th edition