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

HỆ ĐIỀU HÀNH (CO2017)

Bài tập lớn (học kì: 232)

"Simple Operating System"

Instructor(s): GV. Nguyễn Phương Duy

Student(s):

Trịnh Thị Anh Thư	2213412	*(Lớp L09 - Nhóm 7,* ***Leader****)*
Trương Nguyễn Minh Nhiên	2212452	*(Lớp L09 - Nhóm 7)*
Phạm Thị Tố Như	2212478	*(Lớp L09 - Nhóm 7)*
Trần Thị Ngọc Huyền	2211311	*(Lớp L09 - Nhóm 7)*
Nguyễn Duy Thông	2213331	*(Lớp L09 - Nhóm 7)*
Nguyễn Hoàng Nam	2212142	*(Lớp L09 - Nhóm 7)*

TP. HỒ CHÍ MINH, 2024

Contents

1 Thành viên và phân chia công việc

STT	Fullname	Student ID	Problems	Đóng góp
1	Trịnh Thị Anh Thư	2213412	Code Memory&TLB	100 %
2	Trương Nguyễn Minh Nhiên	2212452	Lý thuyết Memory	100 %
3	Phạm Thị Tố Như	2212478	Code Memory&TLB	100 %
4	Trần Thị Ngọc Huyền	2211311	Code và lý thuyết Scheduler	100 %
5	Nguyễn Duy Thông	2213331	Code nền Scheduler	50 %
6	Nguyễn Hoàng Nam	2212142	Lý thuyết TLB&Wrapping	100 %

2 Tổng quan

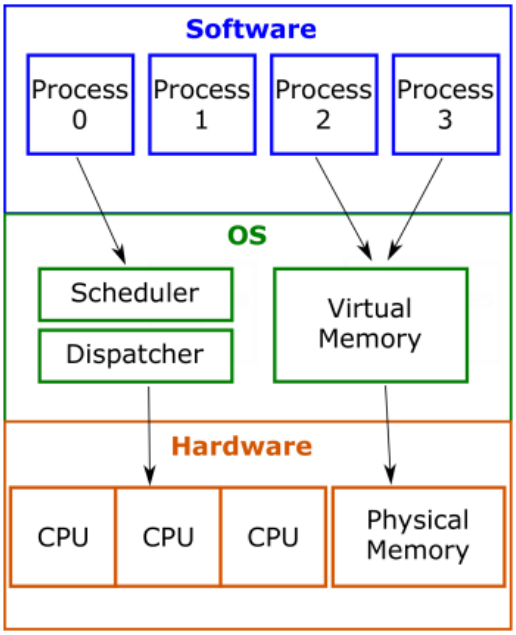
2.1 Giới thiệu

Nhiệm vụ: Ở bài tập lớn này, chúng ta mô phỏng một hệ điều hành đơn giản dựa trên các thành phần chính của nó: Định thời (Scheduler), đồng bộ hóa (Synchronization), sự vận hành của bộ nhớ ảo và bộ nhớ vật lí (Memory management).

Mục tiêu: Hiểu được phần nào nguyên lí của một hệ điều hành giản đơn, các thành phần chính của hệ điều hành cùng vai trò của chúng.

Hình 1 mô tả cấu trúc hệ điều hành mà chúng ta sẽ hiện thực. Nhìn chung, hệ điều hành có nhiệm vụ quản lý hai tài nguyên *ảo (virtual)*: CPU(s) và RAM nhờ sử dụng:

- Scheduler (Dispatcher): Xác định process nào sẽ được chạy trên một CPU nào đó.
- Virtual Memory engine (VME): Cô lập không gian bộ nhớ của mỗi tiến trình với nhau. Bộ nhớ vật lý được chia sẻ bởi nhiều tiến trình nhưng mỗi tiến trình không biết sự tồn tại của tiến trình khác. Điều này được thực hiện bằng cách cho phép mỗi tiến trình có không gian bộ nhớ ảo riêng. VME sẽ ánh xạ và dịch các địa chỉ ảo do các tiến trình cung cấp sang các địa chỉ vật lý tương ứng.



Hình 1: *Tổng quan về các modules trong bài tập lớn này.*

Qua đó cho phép nhiều chương trình có thể chia sẻ, sử dụng tài nguyên ảo trên máy tính. Ở bài tập lớn này chúng ta sẽ tập trung hiện thực **Scheduler/Dispatcher và Virtual Memory engine.**

2.2 Processes

Process (tiến trình) là chương trình đang được thực thi. Mỗi process được biểu diễn trong hệ điều hành bởi khối điều khiển process (Process Control Block hay PCB).
Process trong bài tập lớn này:

- Nội dung của mỗi tiến trình là một bản sao của một chương trình được lưu trữ trên đĩa. Một chương trình được xác định bởi một tệp duy nhất có định dạng sau:

```
[priority] [N = number of instructions]
instruction 0
instruction 1
...
instruction N-1
```

Hình 2: *Content of process.*

Với N là số instructions, priority là priority default, giá trị này sau đó có thể bị ghi đè bởi live priority khi tiến trình được thực thi.

- Nội dung của file input:

```
[time slice] [N = Number of CPU] [M = Number of Processes to be run]
[time 0] [path 0] [priority 0]
[time 1] [path 1] [priority 1]
...
[time M-1] [path M-1] [priority M-1]
```

Hình 3: *Content of input file.*

Với time slice là thời gian được phép chạy của một process, N là số CPU, M là số tiến trình được phép chạy. Tham số priority là live priority, sẽ ghi đè lên default priority theo cơ chế ***dual priority mechanism***.

- Nội dung của input file và process sẽ được loader đọc và gán và PCB tương ứng, nội dung của PCB như sau:

```
#ifdef MLQ_SCHED
  struct pcb_t {
        uint32_t pid;
        uint32_t priority;
 value depend on process itself
        struct code_seg_t * code;
        addr_t regs[10];
        uint32_t pc;
 #ifdef MLQ_SCHED
        uint32_t prio;
 #endif
 #ifdef CPU_TLB
        struct memphy_struct *tlb;
 #endif
```

```
#ifdef MM_PAGING
        struct mm_struct *mm;
        struct memphy_struct *mram;
        struct memphy_struct **mswp;
        struct memphy_struct *active_mswp;
#endif
        struct page_table_t * page_table; // Page table
        uint32_t bp;    // Break pointer
};
```

Trong đó:

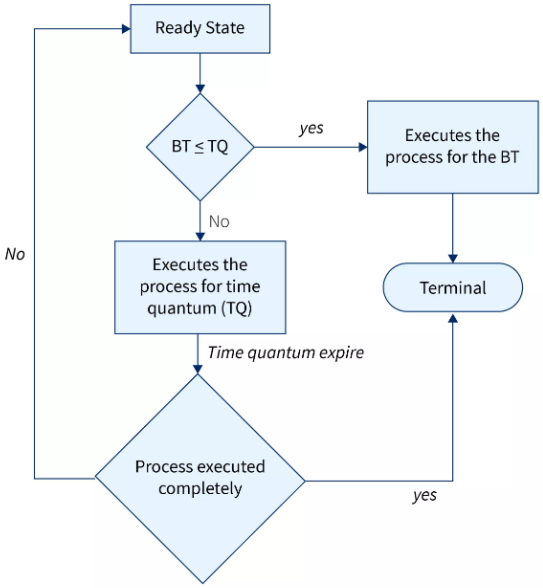
- PID: PID của một process, phân biệt giữa các process.
- priority: Độ ưu tiên của một process (priority default).
- code:Con trỏ đến khối các instructions.
- regs: Thanh ghi, mỗi tiến trình được sử dụng tối đa 10 thanh từ 0 đến 9.
- prio: Độ ưu tiên khi thực thi chương trình (live priority).
- pc: Địa chỉ instruction cuối cùng của process được program counter trỏ đến trước khi dừng tiến trình.

3 Scheduler

Đối với Hệ thống đơn chương trình, khi một tiến trình sử dụng các thiết bị O/I, CPU trở nên nhàn rỗi, trong khi đó các tiến trình khác phải chờ tiến trình đang chạy kết thúc để được thực thi. Điều này dẫn đến tốn thời gian và lãng phí tài nguyên. Hệ thống đa chương trình xử lý vấn đề trên bằng cách cho các tiến trình thay phiên nhau sử dụng tài nguyên (context switching) mà không cần chờ tiến trình đang chạy hoàn tất. Vai trò của hệ điều hành lúc này là điều phối tài nguyên nhằm tối ưu hệ thống, một trong những hoạt động điều phối là quyết định tiến trình nào sẽ được chạy trên CPU và thời gian chạy nhờ CPU scheduling (định thời). Có nhiều giải thuật định thời, ở bài tập lớn này, chúng ta sử dụng giải thuật Multi Level Queue (MLQ) để xác định tiến trình nào sẽ được thực thi.

3.1 Cơ sở lý thuyết

Sơ lược về giải thuật Round Robin: Chỉ cho phép các tiến trình được thực hiện trên CPU trong một thời gian xác định - Time Quantum (slice). Khi một tiến trình chạy hết thời gian một Quantum mà chưa hoàn thành, tiến trình sẽ bị đẩy lại vào ready queue. Thuật toán lặp lại đến khi tiến trình hoàn tất.



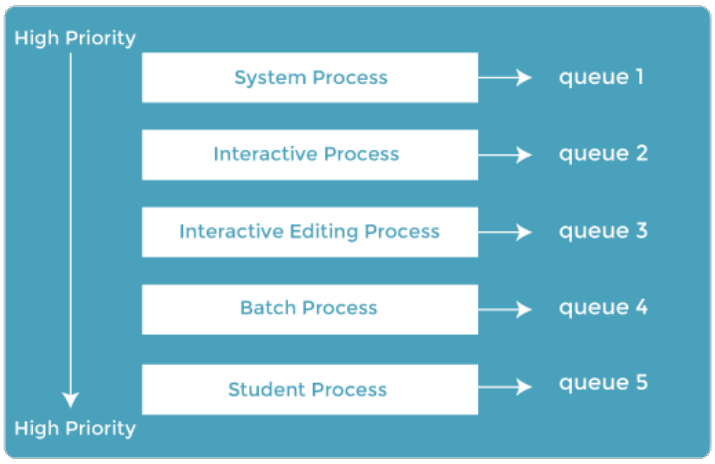
Hình 4: *Giải thuật Round Robin*

Sơ lược về giải thuật MLQ: Cho phép phân các tiến trình khác nhau vào các ready queue khác nhau dựa trên độ ưu tiên của tiến trình. . Các tiến trình ưu tiên cao được đặt trong các hàng đợi có mức độ ưu tiên cao hơn, trong khi các tiến trình ưu tiên thấp được đặt trong các hàng đợi có mức độ ưu tiên thấp hơn.

Đặc điểm của giải thuật MLQ:

- Các process được cố định vào hàng đợi đến khi tiến trình được hoàn tất.

- Có nhiều tiêu chuẩn để phân process vào queue dựa trên yêu cầu tiến trình, loại tiến trình (system process, interactive process hay batch process,..).
- Các ready queue khác nhau có thể có phương pháp định thời khác nhau.
- Không chỉ các process mà các queue cũng được hệ điều hành định thời.



Hình 5: *Giải thuật MLQ dựa trên loại tiến trình*

Ưu điểm:

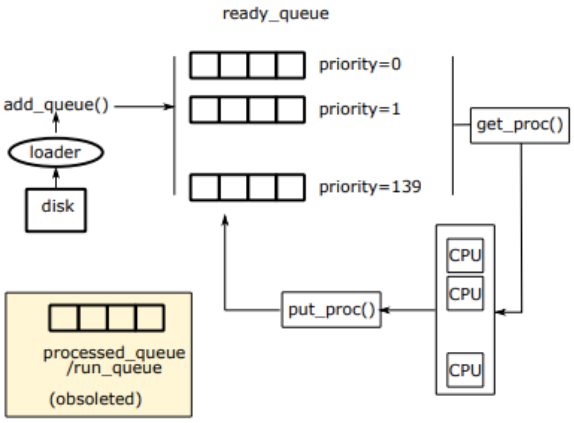
- Đảm bảo những process với độ ưu tiên cao được thực thi trước.
- Tính linh hoạt: Cho phép các ready queue với mỗi queue chứa các process có sự tương đồng (về yêu cầu thực thi, về chủng loại,..) sử dụng các giải thuật định thời khác nhau, nhờ đó cá biệt hóa từng loại process.

Hạn chế:

- Với số lượng queue lớn, việc xác định phương pháp định thời riêng từng queue và xử lý chúng gây mất thời gian và phức tạp.
- Vấn đề Starvation: Các process với độ ưu tiên thấp có thể bị chặn và phải đợi một khoảng thời gian rất dài cho đến khi các process với độ ưu tiên cao hơn hoàn tất.

Trong bài tập lớn này:

- Đối với mỗi chương trình mới, Loader sẽ tạo một process và PCB tương ứng. Sau đó, nó sao chép nội dung của chương trình vào con trỏ code của PCB (xem phần 2.2). PCB được đưa vào ready queue có prio(ready queue) = priority (process).
- CPU sử dụng giải thuật Round Robin để quản lý ready queue và thực thi process, mỗi process chỉ được phép chạy trong time slice xác định.
- Giải thuật MLQ: Độ ưu tiên của các ready queue được xác định dựa trên chỉ số **prio** và có số lượng process được thực thi mỗi lần xác định slot = (MAX_PRIO - prio).



Hình 6: *Giải thuật MLQ trong bài tập này.*

3.2 Các câu hỏi về Scheduler

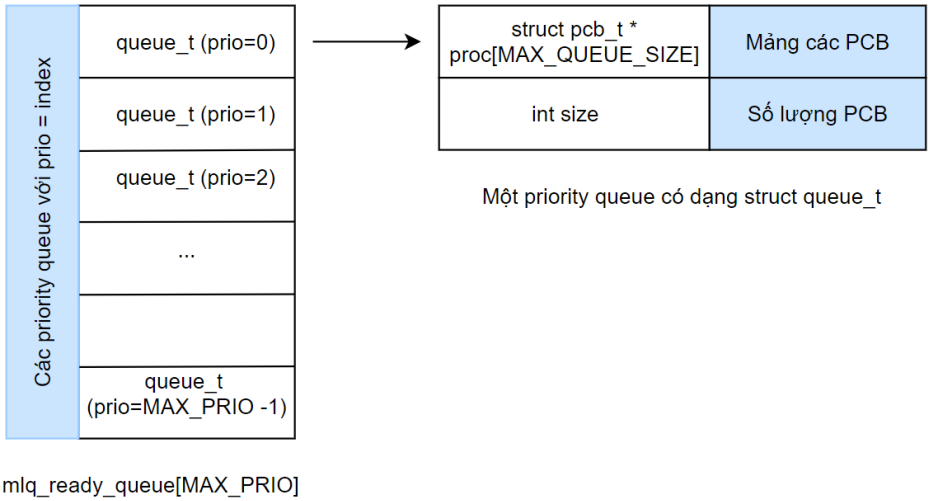
Câu hỏi: Điểm mạnh của giải thuật định thời trong bài tập lớn này so với các giải thuật khác đã học?

Trả lời: Ở bài tập lớn này, chúng ta kết hợp sử dụng hai giải thuật là Round Robin và Multi Level Queue để định thời.

- Nhìn chung, ưu điểm của giải thuật này là:
  - MLQ tạo điều kiện thực hiện các process với độ ưu tiên (sự cần thiết) cao hơn trước.
  - Giảm vấn đề Starvation nhờ yêu cầu time slice đối với process của giải thuật RR và slot đối với ready queue.
  - Sự kết hợp giữa hai giải thuật trên giúp tối ưu hóa tài nguyên bằng cách cung cấp dịch vụ cho tác vụ cần thiết đồng thời đảm bảo các tác vụ đó độ ưu tiên thấp vẫn được thực thi.
- So với các giải thuật khác:
  - Tránh gây ra hiệu ứng đầu xe lửa (convoy effect - khi tiến trình được thực hiện trước chiếm quá nhiều thời gian, dẫn đến các tiến trình sau phải chờ rất lâu) và Starvation so với First Come First Serve (định thời dựa trên thứ tự đến của các process) nhờ mỗi process chỉ được chạy trong time slice.
  - So với Shortest Job First (định thời dựa trên process nào có burst time ngắn nhất) và Longest Job First (định thời dựa trên công việc có burst time dài nhất), việc thực hiện các process dựa trên sự ưu tiên của nó có thể có hiệu quả quả hơn do đảm bảo các công việc có độ ưu tiên cao hay cần thiết hơn được thực hiện trước.
  - So với thực hiện riêng MLQ, kết hợp Round Robin mang ưu thế của của các giải thuật định thời preemptive, giảm waiting time, đảm bảo không tiến trình nào độc chiếm CPU. Đồng thời, MLQ cũng góp phần tối ưu Round Robin nhờ việc cho các tiến trình có độ ưu tiên cao thực hiện trước.

3.3 Hiện thực

Trong bài tập lớn này, để hiện thực giải thuật MLQ, chúng ta sử dụng mảng **struct queue_t mlq_ready_queue[MAX_PRIO]** để quản lý các **struct queue_t priority queue**. Độ ưu tiên của một priority queue chính bằng index của priority queue đó trong mlq_ready_queue. Định dạng của struct queue_t và mô tả hiện thực MLQ được minh họa dưới đây.



4

Hình 7: *Multi level queue.*

Có thể hiểu đơn giản mlq_ready_queue hoạt động giống như một bảng băm với key = index = prio, tại mỗi vị trí index là một hàng đợi first in first out (queue).

3.3.1 Hiện thực file queue.c

a. Hàm en_queue: Hàm có chức năng thêm một process mới (struct pcb_t * proc) vào queue. Áp dụng cho hai trường hợp, đối với MLQ, process được thêm vào cuối priority queue; trong trường hợp còn lại, process được đưa vào queue dựa trên thứ tự priority từ nhỏ đến lớn (nếu priority bằng nhau, process đến sau sẽ ở sau).

```
void enqueue(struct queue_t * q, struct pcb_t * proc) {
    #ifdef MLQ_SCHED
        q->proc[q->size]=proc;
        q->size++;
    #else
        unsigned index=0;
        for(int i=(q->size -1); i>0; i--){
            if(proc->priority < q->proc[i]->priority) continue;
            else {
              index=i+1;
               if(index==q->size) q->proc[q->size]=proc;
               else {
                   for(int j=q->size; j> index; j--){
                   q->proc[j]=q->proc[j-1];}
                   q->proc[index]=proc;
               }
               q->size++;
            return;
            }
        }
    #endif
}
```

b. Hàm de_queue: Hàm có chức năng lấy một process có độ ưu tiên cao nhất (giá trị priority nhỏ nhất) ra khỏi queue.

```
struct pcb_t * dequeue(struct queue_t * q) {
    if(empty(q)){
                return NULL;
    }
    struct pcb_t * Highest_Priority = q->proc[0];
    for(int i=0; i<(q->size); i++){
        q->proc[i]=q->proc[i+1];
    }
    q->size--;
    return Highest_Priority;
}
```

3.3.2 Hiện thực file sched.c

a. Hàm get_mlq_proc: Trả về một process từ **mlq_ready_queue**. Dựa trên giải thuật định thời yêu cầu trong bài tập lớn này, ý tưởng hàm như sau: Tại mỗi vị trí index (tham số curr_queue), kiểm tra priority queue, nếu queue khác rỗng (1) và có slot > 0 (2), ta lấy một process ra khỏi queue và giảm số slot của queue đó xuống một đơn vị. Nếu vi phạm điều kiện (1) hoặc (2), ta đặt lại slot của queue đó về ban đầu và nhảy đến queue tiếp theo.

```
//Declare slot array, and set  curr_queue value: curr_queue=0, slot[i]=
    MAX_PRIO-i;
#ifdef MLQ_SCHED
static struct queue_t mlq_ready_queue[MAX_PRIO];
```

```
static int slot[MAX_PRIO];
static int curr_queue = 0;
#endif
void init_scheduler(void) {
#ifdef MLQ_SCHED
    int i ;
    curr_queue = 0;
    for (i = 0; i < MAX_PRIO; i ++){
        slot[i]=MAX_PRIO-i;
        mlq_ready_queue[i].size = 0;
    }
#endif
    ready_queue.size = 0;
    run_queue.size = 0;
    pthread_mutex_init(&queue_lock, NULL);
}

//get_mlq_proc function
struct pcb_t * get_mlq_proc(void) {
    struct pcb_t * proc = NULL;
    pthread_mutex_lock(&queue_lock);
    while (queue_empty()==-1) {
        if (!empty(&mlq_ready_queue[curr_queue]) && slot[curr_queue]>0) {
            proc = dequeue(&mlq_ready_queue[curr_queue]);
            slot[curr_queue]-=1;
            //when a priority queue used up all its slot, set it's
            slot =( MAX_PRIO - prio of queue) and jump to next queue.
            if ( slot[curr_queue] <1) {
                slot[curr_queue]=MAX_PRIO-curr_queue;
                curr_queue=(curr_queue +1) % MAX_PRIO;
            }
            break;
        } else {
            curr_queue=(curr_queue +1) % MAX_PRIO;
        }
    }
    pthread_mutex_unlock(&queue_lock);
    return proc;
}
```

b. Hàm get_proc: Trả về một process từ ready queue.

```
#ifdef MLQ_SCHED
struct pcb_t * get_proc(void) {return get_mlq_proc();}
#else
struct pcb_t * get_proc(void) {
        struct pcb_t * proc = NULL;
        pthread_mutex_lock(&queue_lock);
        if(!empty(&ready_queue)) {
                proc=dequeue(&ready_queue);}
        pthread_mutex_unlock(&queue_lock);
        return proc;
}
```

3.4 Kết quả thực thi

Thử một số testcase sched, thu được kết quả như sau:

a. Testcase 1: Chạy với time slice bằng 4, số CPU là 1 và số process là 3 và MAX_PRIO=2.

Testcase:

```
4 1 3
0 p1s 1
1 p1s 1
2 p1s 0
```

Kết quả thực thi:

```
Time slot   0
        ld_routine
                Loaded a process at input/proc/p1s, PID: 1 PRIO: 1
        Time slot   1
                CPU 0: Dispatched process  1
                Loaded a process at input/proc/p1s, PID: 2 PRIO: 1
        Time slot   2
                Loaded a process at input/proc/p1s, PID: 3 PRIO: 0
        Time slot   3
        Time slot   4
        Time slot   5
                CPU 0: Put process  1 to run queue
                CPU 0: Dispatched process  3
        Time slot   6
        Time slot   7
        Time slot   8
        Time slot   9
                CPU 0: Put process  3 to run queue
                CPU 0: Dispatched process  3
        Time slot  10
        Time slot  11
        Time slot  12
        Time slot  13
                CPU 0: Put process  3 to run queue
                CPU 0: Dispatched process  2
        Time slot  14
        Time slot  15
        Time slot  16

        Time slot  17
                CPU 0: Put process  2 to run queue
                CPU 0: Dispatched process  3
        Time slot  18
        Time slot  19
                CPU 0: Processed  3 has finished
                CPU 0: Dispatched process  1
        Time slot  20
        Time slot  21
        Time slot  22
        Time slot  23
                CPU 0: Put process  1 to run queue
```

```
        CPU 0: Dispatched process  2
Time slot  24
Time slot  25
Time slot  26
Time slot  27
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  1
Time slot  28
Time slot  29
        CPU 0: Processed  1 has finished
        CPU 0: Dispatched process  2
Time slot  30
Time slot  31
        CPU 0: Processed  2 has finished
        CPU 0 stopped
```

Giải thích kết quả thực thi (xem biểu đồ bên dưới):

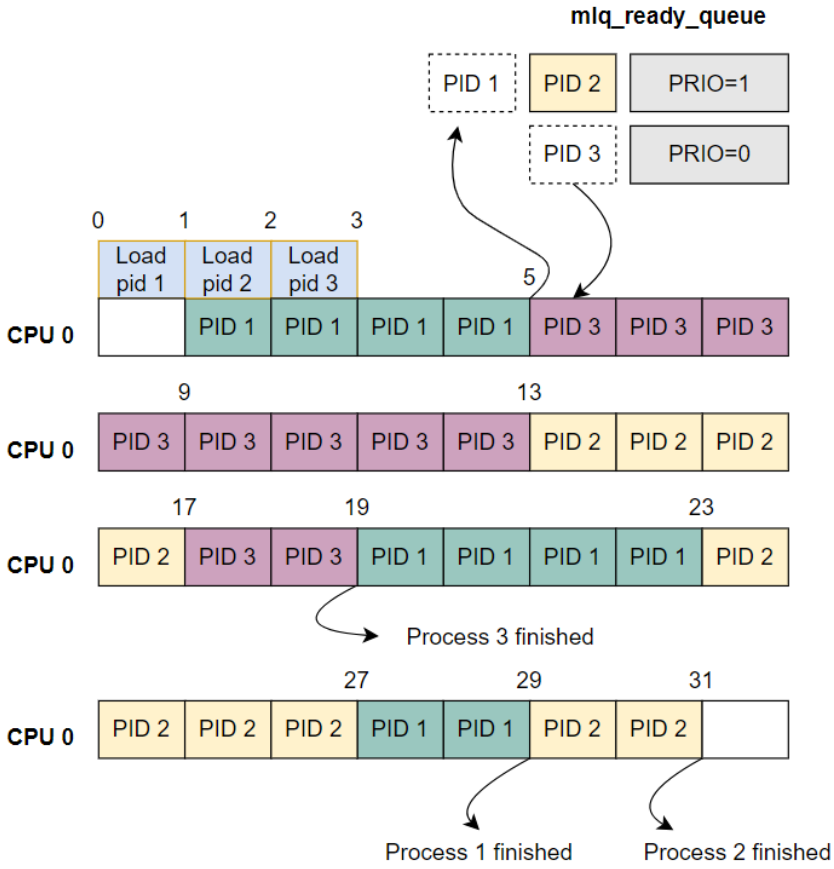
- **Time slot 0:** process đầu tiên được loader khởi tạo PCB tương ứng (PID=1) và được đẩy vào mlq_ready_queue, vì process có live priority là 1 nên được đẩy vào ready queue nằm ở index 1 trong mlq_ready_queue. Ở các Time slot tiếp theo (1, 2), process 2 (PID=2, priority=1) và 3 (PID=3, priority=0) tương tự. *Vì để mặc định MAX_PRIO=2, chúng ta chỉ có 2 priority queue: PRIO_0 (index=prio=0, slot=2) và PRIO_1(index=prio=1, slot=1).*

- **Time slot 1-4:** Ở time slot 1, vì ready queue có độ ưu tiên cao hơn (prio=index=0) rỗng, process nằm ở queue có độ ưu tiên thấp hơn được đẩy vào CPU làm việc. CPU hoạt động theo Round Robin style, mỗi process chỉ được chạy trong time slice cố định (ở đây là 4).

- **Time slot 5:** PID 1 chạy hết time slice và bị đẩy vào priority queue tương ứng. Vì mỗi queue trong multi level queue chỉ được chạy với slot= MAX_PRIO - prio, nên PRIO_1 đã đã chạy hết slot của mình, lượt sử dụng CPU được nhường cho PRIO_0. PID được thực thi.

- **Time slot 5 -13:** PID 3 chạy trên CPU hết 4 time slice, ở time slot 9, PID 3 bị đẩy trở về queue PRIO_0, tuy nhiên do queue PRIO_0 chưa hết lượt chạy, nên PID 3 vẫn tiếp tục thực thi.

- **Time slot 13-31:** Khi queue PRIO_0 chạy hết 2 slot, quyền chạy trên CPU mới được chuyển đến queue PRIO_1, cứ thế luôn phiên cho đến khi tiến trình kết thúc (không còn bị đẩy vào queue).



Hình 8: *Biểu đồ Gantt testcase 1.*

4 Memory management

4.1 Cơ sở lý thuyết

Mỗi Process được thực thi bởi CPU được cấp phát một vùng nhớ trên bộ nhớ thực. Cơ chế phân trang (paging) là kỹ thuật quản lý bộ nhớ (memory management):

- Chia bộ nhớ vật lý thành các frame có kích thước cố định.
- Chia bộ nhớ luận lý thành các page cùng kích thước.

Sau đó, hệ thống sẽ ánh xạ các page của không gian vùng nhớ ảo của mỗi Process với các frame tương ứng trong bộ nhớ vật lý. Để máy tính lưu trữ dữ liệu của các Process trong bộ nhớ thứ cấp và truy cập dữ liệu từ bộ nhớ chính nhờ vào các cơ chế chuyển đổi địa chỉ bởi MMU (Memory Management Unit).

4.1.1 Bộ nhớ ảo (Virtual Memory) ánh xạ trong mỗi Process

Trong mỗi PCB của mỗi tiến trình, không gian vùng nhớ ảo (virtual memory space) sẽ được thiết kế theo phương pháp ánh xạ bộ nhớ **Memory Mapping**, tổ chức thành các vùng nhớ liên tục gọi là **Memory Area**.

Mỗi vùng nhớ **Memory Area** chứa các vùng nhớ con **Memory Region** có kích thước khác nhau để lưu trữ nội dung của Process. Trong không gian vùng nhớ ảo của hệ thống Paging, mỗi Process đều có **CPU Address** để truy cập vào vị trí vùng nhớ nhất định, được chia thành 2 phần để xác định vị trí của dữ liệu trong bộ nhớ vật lý khi có yêu cầu truy cập từ CPU:

Hình 9: CPU Address format.

- Page number (p): chỉ số dùng để xác định page trong bảng phân trang (page table), lưu trữ địa chỉ cơ sở của mỗi Process trong bộ nhớ vật lý.

- Page offset (d): vị trí cụ thể của địa chỉ trong page, kết hợp với Page number để xác định địa chỉ bộ nhớ vật lý được gửi tới khối quản lý bộ nhớ (MMU - memory management unit).

Câu hỏi: Trong hệ điều hành đơn giản này, hiện thực thiết kế nhiều phân đoạn bộ nhớ hoặc vùng bộ nhớ trong khai báo mã nguồn. Ưu điểm của thiết kế nhiều phân đoạn được đề xuất là gì?

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

Hình 10: Various CPU address bus configuration value.

Trả lời: Ưu điểm của thiết kế nhiều phân đoạn bộ nhớ hoặc vùng bộ nhớ trong khai báo mã nguồn của hệ điều hành đơn giản này là:

- Tối ưu hóa bộ nhớ và tốc độ thực thi: Thay vì tải toàn bộ quá trình vào bộ nhớ khi khởi động, thì chỉ những phần cần thiết được tải vào, giúp việc truy cập và thực thi nhanh hơn, tiết kiệm bộ nhớ hiệu quả và cải thiện được hiệu suất hệ thống.

- Quản lý linh hoạt: Thiết kế nhiều phân đoạn có thể có kích thước khác nhau và khu vực bộ nhớ được phân biệt, quản lý và giám sát độc lập. Nhờ đó mà thao tác quản lý bộ nhớ, các tài nguyên hệ thống trở nên linh hoạt và dễ dàng hơn.

- Bảo mật và toàn vẹn dữ liệu, hệ thống: Các phân đoạn độc lập giúp ngăn chặn khỏi sự xâm nhập từ chương trình hoặc người dùng khác.

- Linh động về vị trí bộ nhớ: Các phân đoạn có thể ánh xạ sang bộ nhớ thực với địa chỉ khác nhau và không bị giới hạn bởi một khối nhất định.

Tuy nhiên, việc thiết kế nhiều phân đoạn bộ nhớ có những nhược điểm như tăng khối lượng công việc, phức tạp hóa việc quản lý bộ nhớ, cần xem xét mức độ phù hợp với yêu cầu và mục đích cụ thể của hệ thống.

4.1.2 Bộ nhớ vật lý (Physical Memory) của hệ thống

Bộ nhớ vật lý được chia thành các khối nhỏ có kích thước cố định được gọi là **Frame**. Mỗi Process được ánh xạ vào không gian vùng nhớ ảo riêng biệt, nhưng tất cả ánh xạ này đều hướng tới một thiết bị nhớ vật lý đơn lẻ (singleton physical device), nơi dùng để lưu trữ Frame. Có hai loại thiết bị bộ nhớ vật lý:

- RAM (Ramdon Access Memory): loại bộ nhớ có thể truy cập trực tiếp từ CPU Address Bus, tức là CPU có thể đọc hoặc ghi dữ liệu trực tiếp vào RAM bằng các lệnh CPU.

- SWAP: thiết bị bộ nhớ thứ cấp, không thể truy cập trực tiếp từ CPU. Dữ liệu cần được truy cập từ SWAP phải được chuyển vào bộ nhớ chính (Main Memory) trước khi thực thi bất kỳ thao tác xử lý dữ liệu. SWAP thường được dùng khi RAM không đủ để chứa tất cả dữ liệu cần thiết cho các Process đang sử dụng.

Bài tập lớn này hỗ trợ phần cứng với 01 thiết bị RAM (được phân thành các frame là **MEMRAM**) và tối đa 4 thiết bị SWAP (được phân thành các frame là **MEMSWP**).

Câu hỏi: Điều gì xảy ra khi chia địa chỉ thành nhiều hơn 2 cấp trong hệ thống quản lý bộ nhớ phân trang?

Trả lời: Trong hệ thống quản lý bộ nhớ phân trang, địa chỉ bộ nhớ được phân thành hai cấp là page và page offset. Khi chia địa chỉ thành nhiều hơn 2 cấp trong hệ thống quản lý bộ nhớ phân trang gây ra các vấn đề sau:

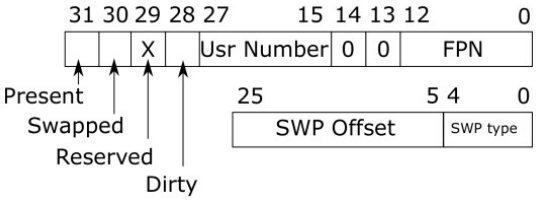
- Tăng thời gian truy cập bộ nhớ: Khi chia địa chỉ thành nhiều hơn 2 cấp, việc ánh xạ địa chỉ đòi hỏi phải sử dụng nhiều page table. Điều này dẫn đến tăng thời gian truy cập bộ nhớ do cần phải thực hiện nhiều phép truy xuất page table để có thể ánh xạ được địa chỉ, làm giảm hiệu suất của hệ thống.

- Lãng phí bộ nhớ: Do cần sử dụng nhiều page table mà mỗi page table cần quản lý và lưu trữ thông tin về ánh xạ địa chỉ, gây ra lãng phí bộ nhớ.

- Phức tạp hóa quản lý bộ nhớ: Việc dùng nhiều cấp trong quản lý bộ nhớ làm tăng độ phức tạp của hệ thống, đặc biệt khi thiếu bộ nhớ hay tốc độ truy cập bộ nhớ bị chậm trễ.

4.1.3 Phương pháp dịch địa chỉ Paging-based

Mỗi Process đều có một bảng phân trang **Page Table**, cấu trúc page table cho phép userspace process tìm ra **physical frame** mà mỗi trang ảo đều có và được ánh xạ tới.



Hình 11: Page Table Entry (PTE) format.

Mỗi bảng phân trang chứa các **Page Table Entries (PTEs)**, mỗi PTE thường bao gồm một giá trị 32-bit cho mỗi trang ảo, chứa các thông tin liên quan đến việc ánh xạ giữa trang ảo và địa chỉ vật lý trong bộ nhớ. Cấu trúc của PTE tùy thuộc vào kiến trúc của hệ điều hành và cách thức triển khai.

```
* Bits 0-12  page frame number (FPN) if present
* Bits 13-14 zero if present
* Bits 15-27 user-defined numbering if present
* Bits 0-4   swap type if swapped
* Bits 5-25  swap offset if swapped
* Bit  28    dirty
* Bits 29    reserved
* Bit  30    swapped
* Bit  31    presented
```

Cơ chế chuyển đổi bộ nhớ **Memory Swapping** là chuyển đổi các frame của bộ nhớ giữa RAM và SWAP. Cơ chế **swapping in** di chuyển nội dung của frame từ SWAP **(MEMSWP)** vào các frame của RAM **(MEMRAM)** khi cần truy cập nội dung để Process thực thi tương ứng. Ngược lại, cơ chế **swapping out** sẽ di chuyển nội dung từ MEMRAM sang MEMSWP, để có được MEMRAM trống do kích thước của SWAP thường đủ lớn.

Cơ chế này hỗ trợ được lượng Process lớn hơn so với dung lượng RAM cho phép, bằng cách dùng SWAP để lưu trữ dữ liệu khi không còn đủ không gian trong RAM. Tuy nhiên, cơ chế này có thể làm giảm hiệu suất do tốc độ truy cập vào SWAP thường chậm hơn so với RAM.

Câu hỏi: Ưu điểm và nhược điểm của phân đoạn theo phân trang là gì?

Trả lời: Phân đoạn theo trang (Segmentation with paging) là kỹ thuật kết hợp giữa phân đoạn và phân trang trong quản lý không gian bộ nhớ.

- Ưu điểm của phân đoạn theo trang:

+ **Tiết kiệm không gian bộ nhớ:** Page table chỉ dùng thông tin các page trong các phân đoạn riêng biệt và kích thước của page table có thể được xác định dựa vào kích thước của các phân đoạn. Điều này giúp kích thước của page table giảm đi và tiết kiệm không gian bộ nhớ so với single-level paging.

+ **Tránh phân mảnh ngoài (external fragmentation):** Việc kết hợp giữa phân đoạn với phân trang giúp duy trì được tính liên tục của không gian bộ nhớ, tránh được hiện tượng phân mảnh ngoài.

+ **Quản lý bộ nhớ hiệu quả:** Cho phép kích thước các phân đoạn thay đổi và có thể chuyển đổi trang giữa bộ nhớ chính và thứ cấp để việc quản lý bộ nhớ tốt hơn, bảo mật hơn.

- Nhược điểm của phân đoạn theo trang:

+ **Tạo ra phân mảnh nội (internal fragmentation):** Page có kích thước cố định nên có thể gây ra phân mảnh nội trong các phân đoạn.

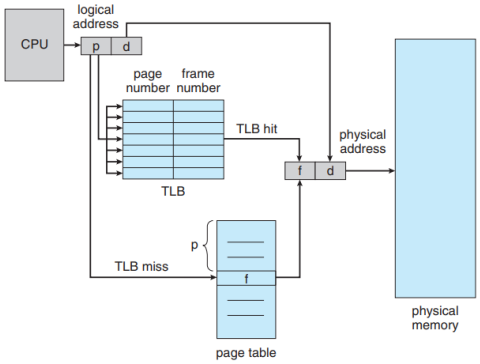
+ **Độ phức tạp cao hơn:** Cần xác định kích thước của phân đoạn, số lượng page cần cấp phát và phải quản lý cả phân đoạn, các page trong từng phân đoạn, đòi hỏi thuật toán và cấu trúc dữ liệu phức tạp hơn.

+ **Tăng thời gian truy cập bộ nhớ:** Khi số lượng các phân đoạn tăng lên dẫn đến yêu cầu lưu trữ bảng phân trang và truy cập đến các thành phần trong page table diễn ra liên tục.

4.1.4 Translation Lookaside Buffer (TLB)

Hệ điều hành và hệ thống con quản lý bộ nhớ có bảng trang (page table) riêng, bảng này chứa mục nhập bảng trang (page table entry) cung cấp frame number. Vấn đề xảy ra nằm ở việc tối ưu hóa thời gian truy cập cho các mục này, thời gian truy cập lâu hơn khi đọc page table và

truy cập dữ liệu bộ nhớ trong MEMPHY (Bộ nhớ vật lý). Để giải quyết các vấn đề trên ta sử dụng TLB, TLB là bộ nhớ có tính liên kết và tốc độ cao vì mỗi mục trong TLB bao gồm hai phần: khóa (hoặc thẻ) và giá trị. Khi bộ nhớ liên kết được hiển thị với một mục (thông tin muốn truy xuất,...), mục này được so sánh với tất cả các khóa cùng một lúc. Nếu mục được tìm thấy, trường giá trị phản hồi sẽ được trả về.



Hình 12: Paging hardware with TLB

Dưới đây là các công việc cơ bản của TLB:
Phương thức truy cập TLB: TLB là phần cứng với sự hỗ trợ của cơ chế ánh xạ để xác định cách thức truy cập nội dung được liên kết với thông tin định dạng.
Thiết lập TLB: TLB chứa các mục trong page table được sử dụng gần đây. Khi CPU tạo ra một địa chỉ ảo, MMU sẽ kiểm tra page number trong TLB:
– Nếu có tồn tại trong page table (TLB hit), frame number tương ứng sẽ được truy xuất.
– Nếu không tìm thấy trong page table (TLB miss), page number sẽ được sử dụng làm index để truy cập page table trong bộ nhớ chính. Nếu page không có trong bộ nhớ chính thì sẽ xảy ra lỗi trang và TLB sẽ được cập nhật page entry mới.

Ta có công thức tính thời gian truy cập bộ nhớ hiệu quả (Effective memory access time-EMAT):

$$\mathbf{EMAT = h\times(c+m)+(1\text{-}h)\times(c+n\times m)}$$

Trong đó:
+ h là tỷ lệ hit của TLB (hit ratio).
+ m là thời gian truy cập bộ nhớ.
+ c là thời gian truy cập TLB.
+ n là hệ số biểu thị cấp độ hệ thống.
Câu hỏi: Điều gì sẽ xảy ra nếu hệ thống đa lõi có mỗi lõi CPU có thể chạy trong một bối cảnh khác nhau, mỗi lõi có MMU riêng và một phần của lõi (TLB)?
Trả lời:
- Ưu điểm:
+ Nâng cao hiệu suất làm việc của máy vì các lõi CPU có thể cùng lúc xử lí nhiều công việc khác nhau, làm giảm thời gian chờ từ đó nâng cao hiệu suất làm việc.
+ Nâng cao tính bảo mật vì mỗi lõi CPU sẽ chạy trong một không gian bộ nhớ riêng biệt, làm giảm nguy cơ xâm nhập từ các lỗ hổng.
+ Quản lí bộ nhớ tốt hơn vì mỗi lõi đều có MMU và TLB riêng có thể duy trì bộ nhớ riêng của

từng lõi, giảm sự cạnh tranh tài nguyên của hệ thống và giảm thời gian truy cập bộ nhớ.

- Nhược điểm:

+ Hiệu suất không đồng đều, dù mỗi lõi có thể có TLB riêng, nhưng sự phân phối không đồng đều của công việc giữa các lõi có thể dẫn đến hiệu suất không đồng đều.

+ Thiết kế hệ thống với mỗi lõi có MMU và TLB riêng đòi hỏi chi phí và phức tạp cao hơn vì cần đảm bảo rằng các lõi hoạt động hài hòa với nhau và không gây ra xung đột tài nguyên hoặc hiệu suất.

Câu hỏi: Trong CPU hiện đại, TLB 2 cấp rất phổ biến hiện nay, tác động của các cấu hình phần cứng bộ nhớ mới này đến các sơ đồ dịch thuật của chúng ta là gì?

Trả lời:

- Ưu điểm:

+ Tăng lượng thông tin, dữ liệu lưu trữ: TLB 2 cấp sẽ gồm có cấp độ lớn (Level 1 - L1 TLB) và cấp độ nhỏ (Level 2 - L2 TLB), nhờ vậy nên ta sẽ có thể chứa được thêm một lượng thông tin, dữ liệu.

+ Giảm tỉ lệ TLB Miss: vì chứa được nhiều thông tin, dữ liệu hơn nên khi TLB ở cấp độ nhỏ bị trượt, CPU vẫn có thể kiểm tra TLB ở cấp độ lớn để xem xét trước các bản ghi TLB bị lỡ hoặc ngược lại, từ đó nâng cao hiệu suất của máy nhờ việc giảm TLB Miss.

- Nhược điểm:

+ Chi phí phần cứng: vì có đến hai nơi lưu trữ nên có thể tốn thêm chi phí sản xuất, chi phí thiết kế.

+ Phức tạp trong thiết kế và quản lý: chúng ta cần đảm bảo tính nhất quán và hiệu quả giữa hai cấp độ TLB để không gây ra xung đột trong qua trình trao đổi dữ liệu.

4.1.5 Wrapping-up all paging-oriented implementations

Giới thiệu về điều khiển cấu hình bằng cách sử dụng định nghĩa hằng số: để làm giảm áp lực trong việc can thiệp giữa các mô-đun, ta sử dụng cách tách từng tính năng thông qua hệ thống cấu hình. Nhờ vậy từng hệ thống con sẽ được duy trì một cách tách biệt nhau, tất cả sẽ tồn tại trong một đoạn mã. Chúng ta có thể kiểm soát cấu hình được sử dụng thông qua include/os-cfg.h .

```
#define MLQ_SCHED 1
#define MAX_PRIO 140
#define CPU_TLB
#define CPUTLB_FIXED_TLBSZ
#define MM_PAGING
#define MM_FIXED_MEMSZ
```

Ví dụ về MM PAGING setting: trong cấu trúc PCB, có một phần nằm trong điều kiện tiền xử lí "#ifdef MM_PAGING", muốn sử dụng thì phải gọi #Define. Điều này có nghĩa là các trường trong phần này chỉ được định nghĩa và sử dụng khi hằng số MM_PAGING đã được định nghĩa trước đó.

```
// From include/common.h
struct pcb_t {
...
#ifdef MM_PAGING
    struct mm_struct *mm;
    struct memphy_struct *mram;
    struct memphy_struct **mswp;
    struct memphy_struct *active_mswp;
```

```
#endif
...
}
```

Ví dụ khác về MM FIXED MEMSZ setting: khi sử dụng một phiên bản cấu trúc mới thì tệp mô tả đầu vào vẫn được giữ nguyên các cài đặt thông qua câu lệnh #define MM_FIXED_MEMSZ và tệp này vẫn hoạt động được trong chế độ mới và có tính tương thích ngược (tương thích ngược là trong khi dùng phiên bản mới thì hệ thống có thể hiểu và xử lí các dữ liệu từ phiên cũ).

Ví dụ về CPU TLB setting: Với mô-đun này của cấu trúc dữ liệu thì ta có nhiều thao tác có thể làm với trang như phân bố, giải phóng, đọc và ghi, được bọc hoặc mã hóa cứng (viết trực tiếp các giá trị, hằng số hoặc logic vào mã nguồn của một chương trình mà không sử dụng các tham số hoặc cấu hình mềm dẻo để điều chỉnh hoặc thay đổi chúng) bởi hệ thống.

```
// From src/cpu.c
int run(struct pcb_t * proc) {
...
#ifdef CPU_TLB
    stat = tlbfree_data(proc, ins.arg_0);
#elif defined(MM_PAGING)
    stat = pgfree_data(proc, ins.arg_0);
#else
    stat = free_data(proc, ins.arg_0);
#endif
...
};
```

Cấu trúc hỗ trợ TLB CPU trong cấu trúc PCB được kích hoạt bằng cách sử dụng TLB CPU liên quan đến dòng cấu hình được định nghĩa trong tệp include/os-cfg.h:

```
// From include/common.h
struct pcb_t {
...
#ifdef CPU_TLB
    struct memphy_struct *tlb;
#endif
...
};
```

Cấu hình mới với khai báo rõ ràng về kích thước bộ nhớ: Nếu ở trong chế độ này, thì chương trình mô phỏng sẽ lấy thêm các dòng từ tệp đầu vào. Những dòng đầu vào này chứa kích thước của TLB và các kích thước bộ nhớ vật lý của hệ thống: một MEMRAM và tối đa 4 MEMSWP. Giá trị kích thước yêu cầu là một số nguyên không âm. Chúng ta có thể đặt kích thước bằng 0, nhưng điều đó có nghĩa là trao đổi bị vô hiệu hóa. Để giữ cho tham số kích thước bộ nhớ hợp lệ, chúng ta phải có một MEMRAM và ít nhất 1 MEMSWAP, các giá trị đó phải là số nguyên dương, các giá trị còn lại *(là giá trị còn lại của MEMSWP khi đã có MEMRAM và MEMSWAP)* có thể được đặt bằng 0.

```
[time slice] [N = Number of CPU] [M = Number of Processes to be run]
[CPU_TLB_SZ]
[MEM_RAM_SZ] [MEM_SWP_SZ_0] [MEM_SWP_SZ_1] [MEM_SWP_SZ_2] [MEM_SWP_SZ_3]
[time 0] [path 0] [priority 0]
[time 1] [path 1] [priority 1]
```

```
...
[time M-1] [path M-1] [priority M-1]
```

[time slice]: thời gian chia cho mỗi slice thời gian, tức là khoảng thời gian mà mỗi CPU hoặc tiến trình sẽ được thực hiện trước khi chuyển đổi sang tiến trình khác.

[N = Number of CPU]: số lượng CPU trong hệ thống, tức là số lượng tiến trình có thể được thực thi đồng thời.

[M = Number of Processes to be run]: số lượng các tiến trình hoặc luồng được mô phỏng trong hệ thống. Mỗi tiến trình sẽ có một lịch trình thực thi riêng. [CPU_TLB_SZ]: Kích thước của bộ TLB của CPU.

[MEM_RAM_SZ] [MEM_SWP_SZ_0] [MEM_SWP_SZ_1] [MEM_SWP_SZ_2] [MEM_SWP_SZ_3]: Kích thước của bộ nhớ vật lý (RAM) và các phân vùng bộ nhớ trao đổi (swap memory) trong hệ thống. MEM_RAM_SZ là kích thước của bộ nhớ RAM chính, trong khi MEM_SWP_SZ_0 đến MEM_SWP_SZ_3 là kích thước của các phân vùng bộ nhớ trao đổi (nếu có).

[time i] [path i] [priority i]: mô tả thông tin về từng tiến trình hoặc luồng cụ thể, bao gồm thời gian bắt đầu thực thi của tiến trình (time i), đường dẫn tới tệp thực thi của tiến trình (path i), và ưu tiên của tiến trình (priority i).

4.2 Hiện thực

4.2.1 Hiện thực mm-memphy.c

Hiện thực hàm MEMPHY_dump

Chức năng: Hàm MEMPHY_dump có tác dụng in các nội dung khác 0 từ bộ nhớ vật lý ra màn hình.

Code

```
int MEMPHY_dump(struct memphy_struct *mp)
{
/*TODO dump memphy contnt mp->storage
 *     for tracing the memory content
 */
   printf("===== PHYSICAL MEMORY DUMP =====\n");
   for (int i = 0; i < mp->maxsz; ++i)
   {
      if (mp->storage[i] != 0)
      {
         printf("BYTE %08x: %d\n", i, mp->storage[i]);
      }
   }

   printf("===== PHYSICAL MEMORY END-DUMP =====\n");
   printf("
       ================================================================\n"
       );
   return 0;
}
```

4.2.2 Hiện thực mm-vm.c

Hiện thực hàm __alloc

Chức năng: Hàm __alloc được sử dụng để cấp phát bộ nhớ ảo cho một tiến trình trong hệ điều hành. Đầu tiên, hàm tiến hành kiểm tra kích thước được yêu cầu cấp phát có hợp lệ không. Nếu hợp lệ mới được tiếp tục, ngược lại hàm báo lỗi và kết thúc.

Tiếp theo, gọi hàm tìm kiếm vùng nhớ ảo trống có kích thước đủ lớn (get_free_vmrg_area), nếu có thì tiến hành cấp phát vùng nhớ ảo mới cho tiến trình. Nếu không, ta tiến hành mở rộng vùng nhớ ảo hiện có của tiến trình (inc_vma_limit).

Cuối cùng, kiểm tra xem có phần dư thừa nào sau khi cấp phát hay không. Nếu có phần dư thừa, ta cấp phát một cấu trúc vm_rg_struct mới (rg_free) để lưu trữ thông tin về phần dư thừa và thêm nó vào danh sách các vùng nhớ ảo trống (enlist_vm_freerg_list).

Code

```
int __alloc(struct pcb_t *caller, int vmaid, int rgid, int size, int *
    alloc_addr)
{
  if(size <= 0) return -1;
  struct vm_rg_struct rgnode;
  if (get_free_vmrg_area(caller, vmaid, size, &rgnode) == 0)
  {
    caller->mm->symrgtbl[rgid].rg_start = rgnode.rg_start;
    caller->mm->symrgtbl[rgid].rg_end = rgnode.rg_end;
    *alloc_addr = rgnode.rg_start;
    return 0;
  }
  struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);
  int inc_sz = PAGING_PAGE_ALIGNSZ(size);
  int old_sbrk;
  old_sbrk = cur_vma->sbrk;
  inc_vma_limit(caller, vmaid, inc_sz);

  caller->mm->symrgtbl[rgid].rg_start = old_sbrk;
  caller->mm->symrgtbl[rgid].rg_end = old_sbrk + size;
  *alloc_addr = old_sbrk;

  caller->mm->symrgtbl[rgid].rg_start = old_sbrk;
  caller->mm->symrgtbl[rgid].rg_end = old_sbrk + size;
  *alloc_addr = old_sbrk;

   struct vm_area_struct *remain_rg = get_vma_by_num(caller->mm, vmaid);
  if (old_sbrk + size < remain_rg->sbrk)
  {
    struct vm_rg_struct *rg_free = malloc(sizeof(struct vm_rg_struct));
    rg_free->rg_start = old_sbrk + size;
    rg_free->rg_end = remain_rg->sbrk;
    rg_free->rg_next = NULL;
    enlist_vm_freerg_list(caller->mm, rg_free);
  }
  return 0;
}
```

Hiện thực hàm __free

Chức năng: Hàm __free được sử dụng để giải phóng không gian lưu trữ được liên kết với một vùng nhớ ảo nhất định trong một tiến trình Đầu tiên, ta kiểm tra rgid có nằm trong vùng hợp lệ không, nếu không hợp lệ thì báo lỗi và kết thúc hàm.

Tiếp theo, kiểm tra con trỏ tới vùng nhớ được xác định bởi rgid trong vùng nhớ ảo của tiến trình (caller→mm) có hợp lệ không và có được cấp phát trước đây không, nếu không cũng báo lỗi và kết thúc. Cấp phát một cấu trúc vm_rg_struct mới (freerg_node) để lưu trữ thông tin về vùng nhớ ảo được giải phóng, gán các giá trị của rgnode cho freerg_node, trong đó freerg_node→rg_next=NULL,

Khởi tạo lại rg_start và rg_end của rgnode bằng 0, gọi hàm enlist_vm_freerg_list để thêm freerg_node vào danh sách các vùng nhớ ảo trống được quản lý bởi tiến trình (caller->mm).

Code

```
int __free(struct pcb_t *caller, int vmaid, int rgid)
{
  if(rgid < 0 || rgid > PAGING_MAX_SYMTBL_SZ)
    return -1;
  struct vm_rg_struct *rgnode = get_symrg_byid(caller->mm, rgid);
  if(rgnode->rg_start == 0 && rgnode->rg_end == 0) return -1;

  struct vm_rg_struct *freerg_node = malloc(sizeof(struct vm_rg_struct));
  freerg_node->rg_start = rgnode->rg_start;
  freerg_node->rg_end = rgnode->rg_end;
  freerg_node->rg_next = NULL;

  rgnode->rg_start = rgnode->rg_end = 0;

  enlist_vm_freerg_list(caller->mm, freerg_node);
  return 0;
}
```

Hiện thực hàm pg_getpage

Chức năng: Hàm pg_getpage được sử dụng để lấy thông tin về trang từ bộ nhớ vật lý (RAM) thông qua bảng phân trang của một tiến trình cụ thể. Đầu tiên, truy cập vào mục nhập ứng với trang mong muốn trong bảng phân trang để xem trang này có được load vào bộ nhớ vật lý chưa. Nếu chưa, chúng ta tiến hành tìm một trang nạn nhân trong bộ nhớ vật lý theo cơ chế FIFO và tiến hành swap với trang đang cần load vào bộ nhớ vật lý để tiến hành thực thi (trang này nằm trong bộ nhớ swap). Thiết lập frame number và page number mới cho entry của bảng trang, đồng thời đưa page này vào danh sách quản lý thay trang.

Code

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

  if (!PAGING_PAGE_PRESENT(pte))
  {
    int vicpgn, swpfpn;
    int vicfpn;
    uint32_t vicpte;
    int tgtfpn = PAGING_SWP(pte);

    if(find_victim_page(caller->mm, &vicpgn) == -1) return -1;

    if (MEMPHY_get_freefp(caller->active_mswp, &swpfpn) == -1) return -1;
```

```
    vicpte = mm->pgd[vicpgn];
    vicfpn = PAGING_FPN(vicpte);
    __swap_cp_page(caller->mram, vicfpn, caller->active_mswp, swpfpn);
    __swap_cp_page(caller->active_mswp, tgtfpn, caller->mram, vicfpn);

    pte_set_swap(&mm->pgd[vicpgn], 0, swpfpn);

    pte_set_fpn(&mm->pgd[pgn], vicfpn);
#ifdef CPU_TLB
    /* Update its online status of TLB (if needed) */
#endif

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

  *fpn = PAGING_FPN(pte);

  return 0;
}
```

Hiện thực hàm validate_overlap_vm_area

Chức năng: Hàm validate_overlap_vm_area được dùng để kiểm tra xem khu vực nhớ mới mở rộng có xung đột không gian bộ nhớ với các khu vực nhớ khác trong tiến trình không.

Code

```
int validate_overlap_vm_area(struct pcb_t *caller, int vmaid, int
    vmastart, int vmaend)
{

  if (vmastart >= vmaend)
  {
    return -1;
  }

  struct vm_area_struct *vma = caller->mm->mmap;
  if (vma == NULL)
  {
    return -1;
  }

  /* TODO validate the planned memory area is not overlapped */
  struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);
  if (!cur_vma)
  {
    return -1;
  }

  while (!vma)
  {
    if (vma != cur_vma && vma->vm_start < vmaend && vma->vm_end >
        vmastart)
    {
      return -1;
    }
```

```
    vma = vma->vm_next;
  }
  return 0;
}
```

Hiện thực hàm find_victim_page

Chức năng: Hàm find_victim_page được dùng để tìm trang thay thế bằng thuật toán FIFO và giải phóng vùng nhớ mà trang đó giữ trong bộ nhớ ảo để có thể tiến hành thay trang ở các bước tiếp theo.

Code

```
int find_victim_page(struct mm_struct *mm, int *retpgn)
{
  struct pgn_t *pg = mm->fifo_pgn;
  if(!pg) return -1;

  struct pgn_t *prev = NULL;
  while (pg->pg_next)
  {
    prev = pg;
    pg = pg->pg_next;
  }
  *retpgn = pg->pgn;
  prev->pg_next = NULL;
  /* TODO: Implement the theorical mechanism to find the victim page */

  free(pg);

  return 0;
}
```

4.2.3 Hiện thực mm.c

Hiện thực hàm vmap_page_range

Chức năng: Hàm vmap_page_range được sử dụng để ánh xạ một phạm vi trang (page range) của bộ nhớ vật lý vào không gian địa chỉ của một tiến trình thông qua bảng trang (page table).

Hàm sử dụng vòng lặp để ánh xạ mỗi khung trang (frame) từ danh sách frames vào không gian địa chỉ của quy trình và cập nhật frame number tương ứng vào bảng phân trang. Các trang mới đã được thêm vào bảng trang thông qua việc đưa chúng vào hàng đợi FIFO (fifo_pgn).

Code

```
int vmap_page_range(struct pcb_t *caller,
                    int addr,
                    int pgnum,
                    struct framephy_struct *frames,
                    struct vm_rg_struct *ret_rg)
{
  struct framephy_struct *fpit;
  int pgit = 0;
  int pgn = PAGING_PGN(addr);
```

```
  ret_rg->rg_start = addr;
  ret_rg->rg_end = ret_rg->rg_start + pgnum * PAGING_PAGESZ;

  /* TODO map range of frame to address space
   *      [addr to addr + pgnum*PAGING_PAGESZ
   *      in page table caller->mm->pgd[]
   */
  for (; pgit < pgnum; ++pgit)
  {
    fpit = frames;
    pte_set_fpn(&caller->mm->pgd[pgn + pgit], fpit->fpn);
    frames = frames->fp_next;
    free(fpit);

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

  return 0;
}
```

Hiện thực hàm alloc_pages_range

Chức năng: Hàm alloc_pages_range có chức năng là cấp phát một dải liên tục các khung bộ nhớ vật lý (trang) cho một tiến trình. Trong trường hợp không có khung trống, hàm sẽ thực hiện thao tác thay trang. Cập nhật danh sách frame được phân bổ frm_lst.

Code

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

  for (pgit = 0; pgit < req_pgnum; pgit++)
  {
    newfp_str = (struct framephy_struct *)malloc(sizeof(struct
        framephy_struct));
    if (MEMPHY_get_freefp(caller->mram, &fpn) == 0)
    {
      newfp_str->fpn = fpn;
    }
    else
    { // ERROR CODE of obtaining somes but not enough frames
      int vicpgn, swpfpn;
      if (find_victim_page(caller->mm, &vicpgn) == -1 ||
          MEMPHY_get_freefp(caller->active_mswp, &swpfpn) == -1)
      {
        if (*frm_lst == NULL)
        {
          return -1;
        }
        else
        {
```

```
          struct framephy_struct *freefp_str;
          while (*frm_lst != NULL)
          {
            freefp_str = *frm_lst;
            *frm_lst = (*frm_lst)->fp_next;
            free(freefp_str);
          }
          return -3000;
        }
      }
      uint32_t vicpte = caller->mm->pgd[vicpgn];
      int vicfpn = PAGING_FPN(vicpte);
      __swap_cp_page(caller->mram, vicfpn, caller->active_mswp, swpfpn);
      pte_set_swap(&caller->mm->pgd[vicpgn], 0, swpfpn);
      newfp_str->fpn = vicfpn;
    }
    newfp_str->fp_next = *frm_lst;
    *frm_lst = newfp_str;
  }

  return 0;
}
```

4.2.4 Hiện thực cpu_tlbcache.c

Hiện thực hàm tlb_cache_read

Chức năng: Hàm tlb_cache_read có chức năng đọc dữ liệu từ bộ nhớ cache của TLB (Translation Lookaside Buffer) dựa trên thông tin về trang (page) và tiến trình (process) cụ thể.

Code

```
int tlb_cache_read(struct memphy_struct * mp, int pid, int pgnum, BYTE*
    value)
   /* TODO: the identify info is mapped to
    *      cache line by employing:
    *      direct mapped, associated mapping etc.
    */
   if (mp == NULL) return -1;

    int cache_index = pgnum % mp->maxsz;

   int kq = TLBMEMPHY_read(mp, cache_index, value);
   if(kq == -1) return -1;

   return 0;
}
```

Hiện thực hàm tlb_cache_write

Chức năng: Hàm tlb_cache_write được sử dụng để ghi dữ liệu vào cache của TLB.

Code

```
int tlb_cache_write(struct memphy_struct *mp, int pid, int pgnum, BYTE
    value)
{
```

```
    /* TODO: the identify info is mapped to
     *      cache line by employing:
     *      direct mapped, associated mapping etc.
     */
    if (mp == NULL) return -1;

    int cache_index = pgnum % mp->maxsz;
    mp->pro_id[cache_index] = pid;
    int kq = TLBMEMPHY_write(mp, cache_index, value);
    if(kq == -1) return -1;

    return 0;
}
```

Hiện thực hàm TLBMEMPHY_dump

Chức năng: Hàm TLBMEMPHY_dump có chức năng chính là in ra nội dung bộ nhớ đệm TLB.

Code

```
int TLBMEMPHY_dump(struct memphy_struct * mp)
{
    printf("===== TLB_MEMPHY DUMP =====\n");
    for (int i = 0; i < mp->maxsz; ++i) {
       if (mp->storage[i] != 0) {
          printf("BYTE %08x: %d\n", i, mp->storage[i]);
       }
    }
    printf("===== TLB_MEMPHY END-DUMP =====\n");
    return 0;
}
```

4.2.5 Hiện thực cpu_tlb.c

Hiện thực hàm tlb_change_all_page_tables_of

Chức năng: Hàm tlb_change_all_page_tables_of thực hiện việc cập nhật tất cả các bảng trang của một tiến trình.

Code

```
int tlb_change_all_page_tables_of(struct pcb_t *proc,  struct
    memphy_struct * mp)
{
  if (proc == NULL || mp == NULL) {
        return -1;
    }

    for (int i = 0; i < mp->maxsz; ++i) {
      int cache_index = i % mp->maxsz;
      int kq = tlb_cache_write(proc->tlb, proc->pid, i, mp->storage[
          cache_index]);
      if(kq == -1) return -1;
    }
  return 0;
```

```
}
```

Hiện thực hàm tlb_flush_tlb_of

Chức năng: Hàm tlb_flush_tlb_of được sử dụng để xóa hoàn toàn các dữ liệu trong bộ nhớ cache TLB của một tiếntrình.

Code

```
int tlb_flush_tlb_of(struct pcb_t *proc, struct memphy_struct * mp)
{


   if (proc == NULL || mp == NULL) {
        return -1;
    }

    for (int i = 0; i < mp->maxsz; ++i) {
        int kq = tlb_cache_write(proc->tlb, proc->pid, i, (BYTE)-1);
        if(kq == -1) return -1;
    }

  return 0;
}
```

Hiện thực hàm tlballoc

Chức năng: Hàm tlballoc có chức năng là cấp phát một phần của không gian bộ nhớ ảo cho một tiến trình và cập nhật thông tin về các page và frame tương ứng của process vào TLB (Translation Lookaside Buffer) của process.

Code

```
int tlballoc(struct pcb_t *proc, uint32_t size, uint32_t reg_index)
{
printf("===== Alloc =====\n");
  int addr, val;
   if (proc == NULL) {
        return -1;
    }
  val = __alloc(proc, 0, reg_index, size, &addr);
  if(val == -1) return -1;
  printf("===== PHYSICAL MEMORY AFTER ALLOCATION =====\n");
  printf("PID=%d - Region=%d - Address=%08x - Size=%d byte\n", proc->pid,
      reg_index, addr, size);
#ifdef PAGETBL_DUMP
  print_pgtbl(proc, 0, -1); // print max TBL
#endif
  return val;
}
```

Hiện thực hàm tlbfree_data

Chức năng: Hàm tlbfree_data có chức năng là giải phóng vùng nhớ của không gian bộ nhớ ảo của một tiến trình và cập nhật bảng TLB (Translation Lookaside Buffer) để loại bỏ ánh xạ các trang ảo tương ứng.

Code

```
int tlbfree_data(struct pcb_t *proc, uint32_t reg_index)
{

  int val = __free(proc, 0, reg_index);
  if(val == -1) return -1;

  /* TODO update TLB CACHED frame num of freed page(s)*/
  /* by using tlb_cache_read()/tlb_cache_write()*/
  struct vm_rg_struct *currg = get_symrg_byid(proc->mm, reg_index);
  int index = PAGING_PGN((currg->rg_start))% proc->tlb->maxsz;
  proc->tlb->pro_id[index] = -1;
  return 0;
}
```

Hiện thực hàm tlbread

Chức năng: Hàm tlbread có chức năng đọc dữ liệu từ một vùng bộ nhớ ảo của một tiến trình tại một vị trí cụ thể và sao chép dữ liệu đó vào một vị trí cụ thể trong bộ nhớ vật lý. Kiểm tra địa chỉ cần đọc đã được load vào tlb chưa.

Nếu miss gọi tới __read để đọc dữ liệu từ page table, hàm này gọi tới pg_getval rồi gọi tới pg_getpage để thay trang nếu trang này chưa được load vào bộ nhớ vật lý và dùng MEMPHY_read để đọc dữ liệu từ trang vừa được load vào thanh ghi destination. Sau đó cập nhật lại page và frame này trong tlb qua tlb_cache_write.

Nếu hit, hàm lấy thông tin frame number tương ứng với trang qua hàm tlb_cache_read và đọc dữ liệu bằng MEMPHY_read với frame number đó vào thanh ghi destination.

Code

```
int tlbread(struct pcb_t * proc, uint32_t source,
            uint32_t offset,    uint32_t destination)
{
  BYTE data, frmnum = -1;

  /* TODO retrieve TLB CACHED frame num of accessing page(s)*/
  /* by using tlb_cache_read()/tlb_cache_write()*/
  /* frmnum is return value of tlb_cache_read/write value*/
  struct vm_rg_struct *currg = get_symrg_byid(proc->mm, source);
  int pgnum = PAGING_PGN((currg->rg_start + offset));
  int cache_index = pgnum % proc->tlb->maxsz;
  if(proc->tlb->pro_id[cache_index] == -1 || proc->tlb->pro_id[
      cache_index] != proc->pid)
  {
  #ifdef IODUMP
        printf("TLB miss at read region=%d offset=%d\n", source, offset);
        #ifdef PAGETBL_DUMP
        print_pgtbl(proc, 0, -1);
        #endif
        MEMPHY_dump(proc->mram);
  #endif
        int val = __read(proc, 0, source, offset, &data);
        if(val == -1) return -1;
  }
  else{
  #ifdef IODUMP
```

```
        printf("TLB hit at read region=%d offset=%d\n", source, offset);
        #ifdef PAGETBL_DUMP
        print_pgtbl(proc, 0, -1); //print max TBL
        #endif
        MEMPHY_dump(proc->mram);
   #endif
        int kq = tlb_cache_read(proc->tlb, proc->pid, pgnum, &frmnum);
        if(kq == -1) return -1;
        int phyaddr = (frmnum * PAGING_PAGESZ) + offset;
        int val = MEMPHY_read(proc->mram, phyaddr, &data);
        if(val == -1) return -1;
  }
    destination = (uint32_t) data;
    #ifdef IODUMP
      printf("read region=%d offset=%d value=%d\n", source, offset, data)
          ;
    #ifdef PAGETBL_DUMP
      print_pgtbl(proc, 0, -1);
    #endif
      MEMPHY_dump(proc->mram);
    #endif

  if(proc->tlb->pro_id[cache_index] == -1 || proc->tlb->pro_id[
      cache_index] != proc->pid) {
      if(pg_getpage(proc->mm, pgnum, (int *)&frmnum, proc) != 0)
      return -1;
      int kq = tlb_cache_write(proc->tlb, proc->pid, pgnum, frmnum);
      if(kq == -1) return -1;
      proc->tlb->pro_id[cache_index] = proc->pid;
  }
  return 0;
}
```

Hiện thực hàm tlbwrite

Chức năng: Hàm tlbwrite có chức năng ghi dữ liệu vào một vùng bộ nhớ ảo của một tiến trình tại một vị trí cụ thể. Cơ chế tương tự như tlbread

Code

```
int tlbwrite(struct pcb_t * proc, BYTE data,
             uint32_t destination, uint32_t offset)
{
  int val;
  BYTE frmnum = -1;
  struct vm_rg_struct *currg = get_symrg_byid(proc->mm, destination);
  int pgnum = PAGING_PGN((currg->rg_start + offset));
  int cache_index = pgnum % proc->tlb->maxsz;
  if(proc->tlb->pro_id[cache_index] == -1 || proc->tlb->pro_id[
      cache_index] != proc->pid)
  {
  #ifdef IODUMP
        printf("TLB miss at write region=%d offset=%d value=%d\n",
            destination, offset, data);
        #ifdef PAGETBL_DUMP
        print_pgtbl(proc, 0, -1);
```

```
        #endif
        MEMPHY_dump(proc->mram);
    #endif
        val = __write(proc, 0, destination, offset, data);
        if(val == -1) return -1;
  }
  else{
  #ifdef IODUMP
        printf("TLB hit at write region=%d offset=%d value=%d\n",
            destination, offset, data);
        #ifdef PAGETBL_DUMP
        print_pgtbl(proc, 0, -1);
        #endif
        MEMPHY_dump(proc->mram);
   #endif
        int kq = tlb_cache_read(proc->tlb, proc->pid, pgnum, &frmnum);
        if(kq == -1) return -1;
        int phyaddr = (frmnum * PAGING_PAGESZ) + offset;
        int val = MEMPHY_write(proc->mram, phyaddr, data);
        if(val == -1) return -1;
  }

  if(proc->tlb->pro_id[cache_index] == -1 || proc->tlb->pro_id[
      cache_index] != proc->pid) {
       if(pg_getpage(proc->mm, pgnum, (int *)&frmnum, proc) != 0)
       return -1;
       int kq = tlb_cache_write(proc->tlb, proc->pid, pgnum, frmnum);
       if(kq == -1) return -1;
       proc->tlb->pro_id[cache_index] = proc->pid;
  }
  return val;
}
```

4.3 Kết quả thực thi

```
Time slot   0
ld_routine
    Loaded a process at input/proc/p0s, PID: 1 PRIO: 139
Time slot   1
    CPU 0: Dispatched process  1
Time slot   2
===== Alloc =====
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=1 - Region=0 - Address=00000000 - Size=300 byte
print_pgtbl: 0 - 512
00000000: 80000001
00000004: 80000000
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
================================================================
Time slot   3
===== Alloc =====
```

```
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=1 - Region=4 - Address=00000200 - Size=300 byte
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
Time slot   4
Time slot   5
===== Alloc =====
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=1 - Region=1 - Address=00000190 - Size=100 byte
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
Time slot   6
TLB miss at write region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
===== PHYSICAL MEMORY END-DUMP =====
================================================================
Time slot   7
TLB hit at write region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
```

```
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 000000a4: 100
===== PHYSICAL MEMORY END-DUMP =====
================================================================
Time slot   8
TLB hit at read region=1 offset=20
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000014: 100
BYTE 000000a4: 100
===== PHYSICAL MEMORY END-DUMP =====
================================================================
read region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000014: 100
BYTE 000000a4: 100
===== PHYSICAL MEMORY END-DUMP =====
================================================================
Time slot   9
TLB hit at read region=1 offset=20
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000014: 100
BYTE 000000a4: 100
```

```
===== PHYSICAL MEMORY END-DUMP =====
================================================================
read region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000014: 100
BYTE 000000a4: 100
===== PHYSICAL MEMORY END-DUMP =====
================================================================
Time slot  10
TLB miss at write region=3 offset=20 value=103
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000014: 100
BYTE 000000a4: 100
===== PHYSICAL MEMORY END-DUMP =====
================================================================
Time slot  11
TLB hit at read region=3 offset=20
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000014: 100
BYTE 000000a4: 100
BYTE 00000114: 103
===== PHYSICAL MEMORY END-DUMP =====
================================================================
```

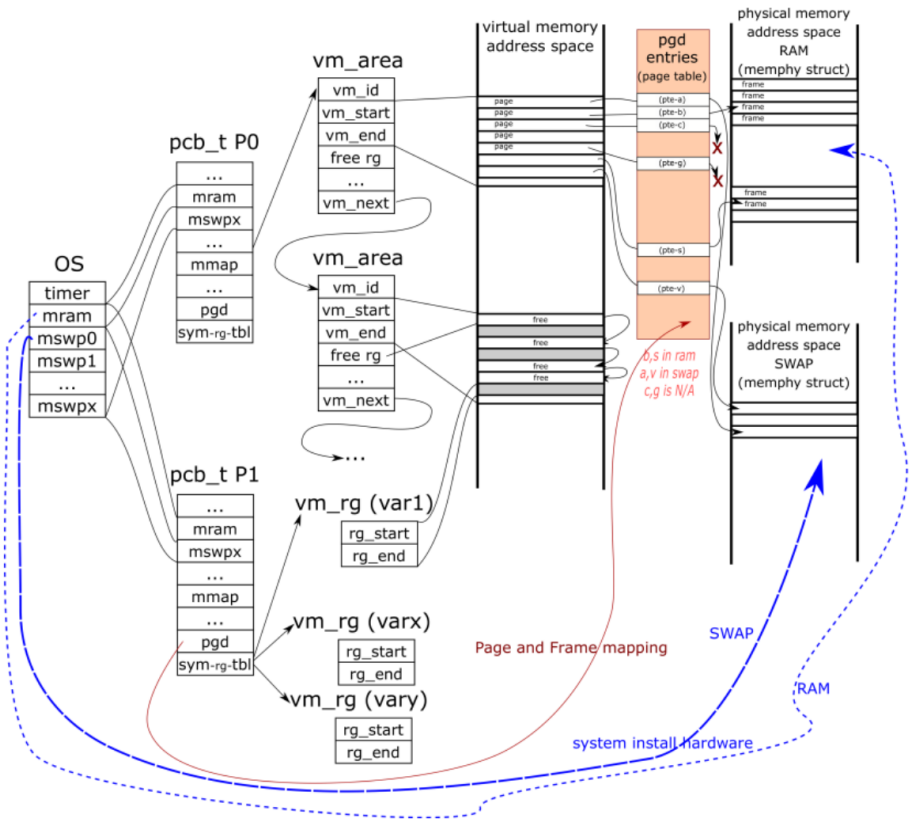
```
read region=3 offset=20 value=103
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000014: 100
BYTE 000000a4: 100
BYTE 00000114: 103
===== PHYSICAL MEMORY END-DUMP =====
================================================================
Time slot  12
TLB hit at read region=1 offset=20
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000014: 100
BYTE 000000a4: 100
BYTE 00000114: 103
===== PHYSICAL MEMORY END-DUMP =====
================================================================
read region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000014: 100
BYTE 000000a4: 100
BYTE 00000114: 103
===== PHYSICAL MEMORY END-DUMP =====
================================================================
Time slot  13
```

```
CPU 0: Processed  1 has finished
CPU 0 stopped
```

5 Put it all together

5.1 Cơ sở lý thuyết

Sau khi kết hợp bộ lập lịch(Scheduler) và quản lý bộ nhớ(Memory management) để tạo thành 1 hệ điều hành hoàn chỉnh, ta có cái nhìn khái quát qua biểu đồ sau.



Hình 13: Hệ điều hành

Vì hệ điều hành chạy trên nhiều bộ xử lý nên có thể có nhiều tiến trình cùng một lúc có thể truy cập đồng thời các tài nguyên chia sẻ. Sau đây là các phương pháp đồng bộ hóa để hệ điều hành trên có thể hoạt đồng đúng.

- **Synchronization Hardware:** Một số hệ điều hành cung cấp tính năng khóa (locking). Trong chức năng này, khi một tiến trình nhập vào phần quan trọng, nó lấy một khóa và khóa được gỡ bỏ khi tiến trình thoát khỏi phần quan trọng. Vì vậy, chức năng khóa này đảm bảo rằng chỉ có một tiến trình vào được phần quan trọng vào một thời điểm vì khi các tiến trình khác cố gắng nhập vào phần quan trọng nó bị khóa.
- **Mutex lock:** là một cơ chế đồng bộ hóa đảm bảo rằng chỉ có một luồng hoặc tiến trình có thể truy cập vào một tài nguyên được bảo vệ bởi mutex tại một thời điểm. Mutex lock là một biến hoặc cấu trúc dữ liệu được sử dụng để thực hiện việc này.

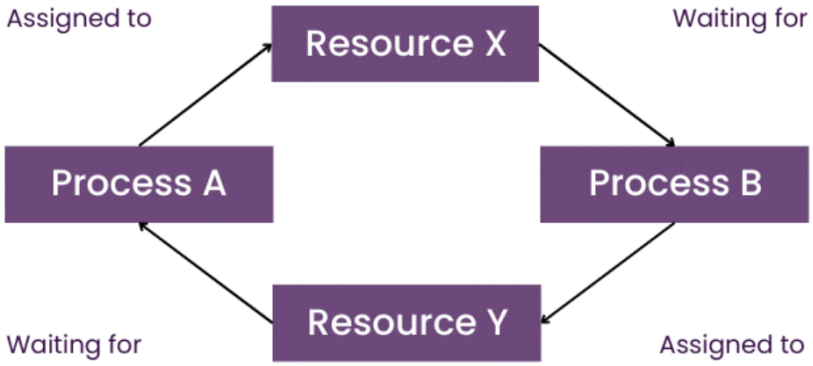
- **Semaphores:** Semaphores là cấu trúc dữ liệu đồng bộ hóa có thể sử dụng để kiểm soát quyền truy cập vào tài nguyên chia sẻ. Semaphores có thể là binary semaphore (2 giá trị: 0 hoặc 1) hoặc là counting semaphore (có thể có giá trị bất kỳ).

5.2 Trả lời câu hỏi

Câu hỏi? Điều gì sẽ xảy ra nếu việc đồng bộ hóa không được xử lý trong hệ điều hành đơn giản của bạn? Minh họa bằng ví dụ vấn đề của hệ điều hành đơn giản của bạn nếu có?

Trả lời: Do hệ điều hành sử dụng nhiều CPU, việc không xử lý đồng bộ sẽ dẫn đến nhiều rủi ro như sau:

- **Data race:** xảy ra khi các process hoặc thread cùng thực hiện đọc và ghi lên một khối dữ liệu, sẽ dẫn đến tính không nhất quán của dữ liệu.
- **Race Condition:** là một tình huống không mong muốn xảy ra khi một thiết bị hoặc hệ thống cố gắng thực hiện hai hoặc nhiều hoạt động cùng một lúc, nhưng do tính chất của thiết bị hoặc hệ thống, các hoạt động phải được thực hiện theo thứ tự đúng đắn để được thực hiện một cách chính xác.
- **Deadlock:** Deadlock trong hệ điều hành là tình huống mà hai hoặc nhiều tiến trình hoặc luồng không thể tiếp tục vì mỗi cái đang chờ đợi cái khác thả ra tài nguyên. Nói cách khác, đó là một trạng thái mà một nhóm các tiến trình bị mắc kẹt và không thể tiến triển. Deadlock thường xảy ra trong các hệ thống nơi nhiều tiến trình cạnh tranh cho tài nguyên hạn chế như thời gian CPU, bộ nhớ hoặc thiết bị nhập hoặc xuất.



Hình 14: Deadlock

5.3 Kết quả thực thi

Chạy thử với file test os_0_mlq_paging thu được kết quả như sau:

```
Time slot   0
ld_routine
```

```
    Loaded a process at input/proc/p0s, PID: 1 PRIO: 139
Time slot   1
    CPU 0: Dispatched process  1
Time slot   2
===== Alloc =====
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=1 - Region=0 - Address=00000000 - Size=300 byte
print_pgtbl: 0 - 512
00000000: 80000001
00000004: 80000000
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
================================================================
Time slot   3
===== Alloc =====
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=1 - Region=4 - Address=00000200 - Size=300 byte
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
Time slot   4
Time slot   5
===== Alloc =====
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=1 - Region=1 - Address=00000190 - Size=100 byte
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
Time slot   6
TLB miss at write region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
```

```
================================================================
===== PHYSICAL MEMORY DUMP =====
===== PHYSICAL MEMORY END-DUMP =====
================================================================
Time slot   7
TLB hit at write region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 000000a4: 100
===== PHYSICAL MEMORY END-DUMP =====
================================================================
Time slot   8
TLB hit at read region=1 offset=20
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000014: 100
BYTE 000000a4: 100
===== PHYSICAL MEMORY END-DUMP =====
================================================================
read region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000014: 100
BYTE 000000a4: 100
===== PHYSICAL MEMORY END-DUMP =====
================================================================
```

```
Time slot   9
TLB hit at read region=1 offset=20
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000014: 100
BYTE 000000a4: 100
===== PHYSICAL MEMORY END-DUMP =====
================================================================
read region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000014: 100
BYTE 000000a4: 100
===== PHYSICAL MEMORY END-DUMP =====
================================================================
Time slot  10
TLB miss at write region=3 offset=20 value=103
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000014: 100
BYTE 000000a4: 100
===== PHYSICAL MEMORY END-DUMP =====
================================================================
Time slot  11
TLB hit at read region=3 offset=20
print_pgtbl: 0 - 1024
```

```
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000014: 100
BYTE 000000a4: 100
BYTE 00000114: 103
===== PHYSICAL MEMORY END-DUMP =====
================================================================
read region=3 offset=20 value=103
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000014: 100
BYTE 000000a4: 100
BYTE 00000114: 103
===== PHYSICAL MEMORY END-DUMP =====
================================================================
Time slot  12
TLB hit at read region=1 offset=20
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000014: 100
BYTE 000000a4: 100
BYTE 00000114: 103
===== PHYSICAL MEMORY END-DUMP =====
================================================================
read region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000001
```

```
00000004: 80000000
00000008: 80000003
00000012: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 3
Page Number: 3 -> Frame Number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000014: 100
BYTE 000000a4: 100
BYTE 00000114: 103
===== PHYSICAL MEMORY END-DUMP =====
================================================================
Time slot  13
    CPU 0: Processed  1 has finished
    CPU 0 stopped
```

Tổng hợp kết quả các test được để ở đường dẫn: Tổng hợp kết quả testcase assignment os - nhóm 7, lớp L09

6 Phần mở rộng

7 Tài liệu tham khảo

References

[1] Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. (2018). *Operating System Concepts,* Tenth edition.

[2] David A. Patterson, John L. Hennessy. (2014). *Computer Organization And Design,* Fifth edition.