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

BÁO CÁO BÀI TẬP LỚN
HỆ ĐIỀU HÀNH (MỞ RỘNG) (CO201D)

THIẾT KẾ HỆ ĐIỀU HÀNH ĐƠN GIẢN

GVHD: Lê Thanh Vân
SV: Nguyễn Tuấn Minh – 2110359
Hoàng Đức Nguyên – 2110393

TP. HỒ CHÍ MINH, THÁNG 5/2023

Mục lục

1 Danh sách thành viên & Phân công công việc

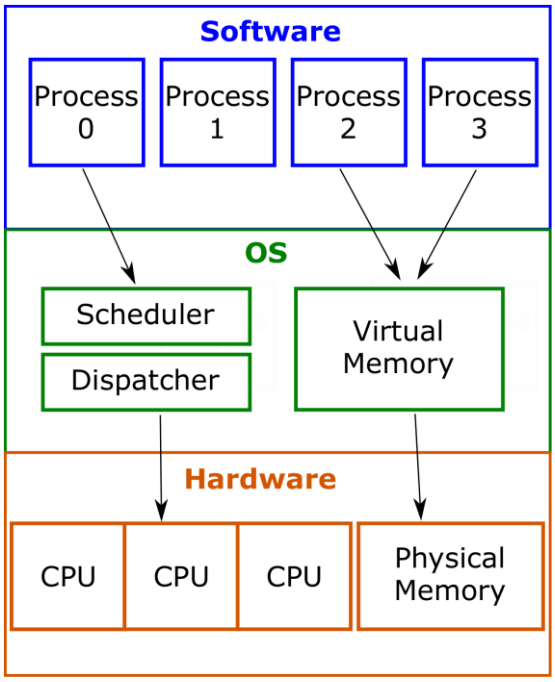
STT	Tên đầy đủ	MSSV	Phân công công việc	% hoàn thành
1	Nguyễn Tuấn Minh	2110359	Memory Management & Report	100%
2	Hoàng Đức Nguyên	2110393	Scheduling & Report	100%

2 Giới thiệu đề tài

Mục đích của bài tập lớn này là mô phỏng một hệ điều hành đơn giản, giúp sinh viên hiểu kiến thức cơ bản về định thời (Scheduling), quá trình đồng bộ (Synchronization) và quản lý bộ nhớ (Memory Management). Hình 1 thể hiển tổng quan cấu trúc của hệ điều hành mà chúng ta sẽ hiện thực.

Về cơ bản, hệ điều hành sẽ quản lý 2 tài nguyên ảo: CPU(s) và RAM, sử dụng 2 thành phần:

- **Scheduler (và Dispatcher):** quyết định quá trình nào sẽ được thực thi trên CPU nào đó
- **Virtual Memory Engine (VME):** cô lập không gian bộ nhớ của mỗi quá trình khỏi những quá trình khác. Mặc dù RAM được chia sẻ bởi nhiều quá trình, mỗi quá trình không biết sự tồn tại của các quá trình khác. Điều này được thực hiện bằng cách cho phép mỗi tiến trình có không gian bộ nhớ ảo riêng và công cụ bộ nhớ ảo sẽ ánh xạ và dịch các địa chỉ logic được cung cấp bởi các quy trình sang các địa chỉ vật lý tương ứng.



Hình 1: Tổng quan về các module chính trong bài tập lớn này

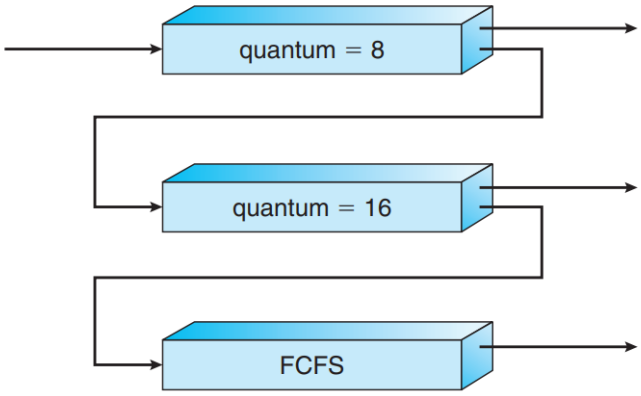
Thông qua các module trên, hệ điều hành cho phép nhiều quá trình được tạo ra bởi người dùng chia sẻ và sử dụng các tài nguyên. Do đó, ở trong bài tập lớn lần này, chúng ta sẽ tiến hành hiện thực các thành phần Scheduler/Dispatcher và VME.

3 Cơ sở lý thuyết

3.1 Scheduler

Trong bài tập lớn này, chúng ta sẽ sử dụng giải thuật **MLFQ (Multilevel Feedback Queue)** để xác định tiến trình (process) sẽ được thực hiện trong quá trình định thời.

Multilevel Feedback Queue (MLFQ) là một giải thuật định thời cho hệ thống máy tính, trong đó các tiến trình được phân vào các hàng đợi khi nhập hệ thống và có thể di chuyển giữa các hàng đợi. Ý tưởng của MLFQ là phân biệt các tiến trình dựa trên đặc điểm thời gian thực hiện CPU của chúng. Nếu một tiến trình sử dụng quá nhiều thời gian CPU, nó sẽ được chuyển xuống hàng đợi ưu tiên thấp hơn. Tương tự, một tiến trình chờ quá lâu trong hàng đợi ưu tiên thấp có thể được chuyển lên hàng đợi ưu tiên cao hơn. Điều này giúp ngăn chặn tình trạng "starvation" (tiến trình không bao giờ được thực thi).



Hình 2: Multilevel feedback queues

Các thông số cần được xác định trong giải thuật MLFQ:

1. **Số lượng hàng đợi:** Xác định số lượng hàng đợi sử dụng trong hệ thống.
2. **Giải thuật định thời cho mỗi hàng đợi:** Đây là phương pháp được sử dụng để quyết định tiến trình nào sẽ được lựa chọn để thực hiện trên CPU khi hàng đợi đó được đến lượt. Có thể sử dụng các giải thuật định thời như FCFS (First-Come, First-Served), SJF (Shortest Job First), Round Robin, Priority Scheduling, etc. Trong Bài tập lớn này, nhóm sử dụng giải thuật Round Robin cho các hàng đợi, tuy nhiên ở mỗi hàng đợi thời gian quantum time lại khác nhau, quantum time của mỗi hàng đợi được tính bằng công thức **quantum time = time slice * (priority + 1)** riêng hàng đợi có độ ưu tiên thấp nhất, quantum time sẽ chính là time slice.
3. **Phương pháp xác định khi nào một tiến trình sẽ được nâng cấp lên hàng đợi ưu tiên cao hơn (upgrade):** Đây là quy tắc để quyết định khi nào một tiến trình sẽ được chuyển từ hàng đợi ưu tiên thấp lên hàng đợi ưu tiên cao hơn. Ví dụ, có thể xác định nếu một tiến trình đã chờ đợi quá lâu trong hàng đợi thấp thì nó sẽ được nâng cấp lên hàng

đợi cao hơn. Phương pháp để xác định khi nào nên nâng cấp tiến trình thường liên quan đến độ ưu tiên của hàng đợi hiện tại, và có thể sử dụng các chiến lược như preemption hay time slicing. Ở bài tập lớn lần này, không có cơ chế cho một tiến trình có thể nâng cấp lên hàng đợi ưu tiên cao hơn.

4. **Phương pháp xác định khi nào một tiến trình sẽ bị giảm xuống hàng đợi ưu tiên thấp hơn (demote):** Đây là quy tắc để quyết định khi nào một tiến trình sẽ bị giảm xuống hàng đợi ưu tiên thấp hơn. Ví dụ, có thể xác định nếu một tiến trình đã sử dụng quá nhiều thời gian CPU trong hàng đợi cao hơn thì nó sẽ bị giảm xuống hàng đợi thấp hơn khác. Phương pháp để xác định khi nào nên giảm xuống hàng đợi ưu tiên thấp hơn thường liên quan đến đánh giá thời gian chờ đợi của tiến trình hay tổng thời gian thực thì của tiến trình. Ở bài tập lớn lần này, một tiến trình sẽ bị giảm xuống hàng đợi có độ ưu tiên thấp hơn sau mỗi lần thực hiện hết quantum time của hàng đợi hiện tại trừ khi tiến trình đang ở hàng đợi có độ ưu tiên thấp nhất (do không thể giảm độ ưu tiên nữa)
5. **Phương pháp xác định tiến trình sẽ vào hàng đợi nào khi tiến trình đó cần được phục vụ:** Đây là quy tắc để quyết định tiến trình sẽ được đưa vào hàng đợi nào khi nó cần được thực hiện trên CPU. Ví dụ, có thể sử dụng các tiêu chí như thời gian chờ, độ ưu tiên, hoặc loại công việc của tiến trình để xác định hàng đợi phù hợp cho nó.

Giải thuật MLFQ cho phép tùy chỉnh phù hợp với hệ thống cụ thể được thiết kế. Tuy nhiên, nó cũng đòi hỏi các tham số phải được chọn để định nghĩa lập lịch tốt nhất. Khi các thông số này được thiết lập một cách hợp lý, giải thuật MLFQ có thể cải thiện hiệu suất hệ thống và giảm độ trễ cho các tiến trình. Nó cũng đảm bảo tính công bằng và ưu tiên cho các tiến trình ưu tiên cao hơn, đồng thời cho phép tiến trình di chuyển giữa các hàng đợi để phù hợp với yêu cầu của nó. MLFQ là một giải thuật lập lịch phổ quát nhất, nó cũng là giải thuật phức tạp nhất.

Giải thuật MLFQ có một số ưu điểm như:

- Tính linh hoạt cao
- Cho phép các tiến trình khác nhau di chuyển giữa các hàng đợi khác nhau.
- Ngăn chặn hiện tượng "starvation" (tiến trình có độ ưu tiên thấp không bao giờ được thực thi) bằng cách di chuyển tiến trình chờ quá lâu trong hàng đợi ưu tiên thấp lên hàng đợi ưu tiên cao hơn.

Tuy nhiên, Giải thuật MLFQ cũng có những hạn chế như:

- Việc lựa chọn giải thuật định thời tốt nhất đòi hỏi một số phương pháp khác để chọn các giá trị phù hợp.
- Tạo ra nhiều tải CPU hơn.
- Là một trong những giải thuật phức tạp nhất

3.2 Memory Management

Trong Bài tập lớn này, Kỹ thuật Paging được sử dụng để quản lý bộ nhớ (Memory Management). Paging là kỹ thuật quản lý bộ nhớ (memory), trong đó máy tính lưu trữ và nhận dữ liệu từ bộ nhớ thứ cấp để sử dụng trong bộ nhớ chính.

Để hiện thực Paging trong bài tập lớn này, ta có mô tả về các thành phần sử dụng cho kỹ thuật Paging như sau:

3.2.1 Bộ nhớ ảo (Virtual Memory) cho mỗi tiến trình

Không gian vùng nhớ ảo (virtual memory space) được thiết kế sử dụng phương pháp ánh xạ bộ nhớ (memory mapping) cho mỗi tiến trình được quản lý bởi PCB (process control block).

Mỗi tiến trình được cấp phát một vùng nhớ ảo **Virtual Memory Area (VMA)** liên tục độc lập, trong đó có những vùng nhớ con liên tục, có thể rời rạc hoặc không, được sử dụng để lưu nội dung của tiến trình (biến, đoạn mã, v.v..) được gọi là **Memory Region** và được quản lý bởi **Symbol Table** . Vùng nhớ ảo được chia thành các khối có kích thước cố định và bằng nhau gọi là Page (trang) mỗi Page này sẽ được ánh xạ đến một vùng nhớ vật lý có cùng kích thước .

Mỗi tiến trình còn có một **page address (hay logical address)** được cấp bởi CPU (trong đề bài gọi là **CPU address**) để truy cập vào một vị trí vùng nhớ nhất định, được mô tả ở hình dưới đây, cụ thể gồm:

- **Page number (p):** là chỉ số của một bảng phân trang (page table) đang lưu trữ địa chỉ cơ sở của mỗi vùng nhớ tiến trình (page) trong bộ nhớ vật lý.
- **Page offset (d)**: kết hợp với địa chỉ cơ sở để xác định địa chỉ bộ nhớ vật lý được gửi tới Khối Quản lý Bộ nhớ (Memory Management Unit).

Hình 3: CPU Address

3.2.2 Bộ nhớ chính (physical memory) của hệ thống

Tương tự như virtual memory, bộ nhớ chính được chia thành các khối nhớ nhỏ có kích thước cố định gọi là **frame** hay **page frame**. Mỗi frame trong bài tập này được lưu **frame number** sẵn có trong mã nguồn định nghĩa về cấu trúc của frame.

Như đã đề cập, tất cả vùng nhớ có vùng nhớ ảo được ánh xạ, và chúng độc lập với nhau. Tuy nhiên, tất cả ánh xạ đó đều có chỉ tới một thiết bị nhớ vật lý đơn lẻ (singleton physical device). Trong bài tập lớn này có hai thiết bị vật lý:

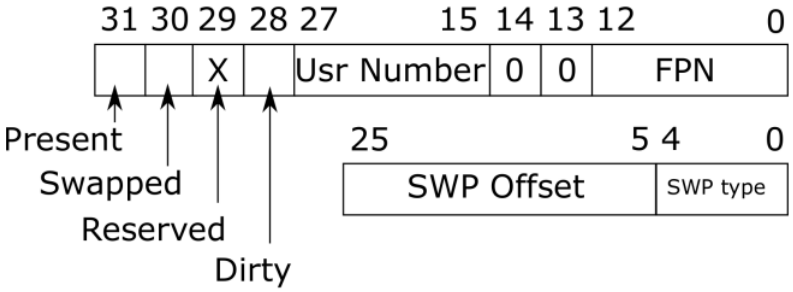
- **RAM**: có thể được truy cập trực tiếp từ CPU address bus (có thể đọc/viết với các lệnh CPU).
- **SWAP**: thiết bị nhớ thứ cấp, không thể truy cập trực tiếp từ CPU. Để thực thi các chương trình và dữ liệu lưu ở thiết bị này, chúng phải được chuyển lên bộ nhớ chính (main memory) để thực thi.

Bài tập lớn lần này được hỗ trợ một thiết bị RAM và 4 thiết bị SWAP. Trong RAM phân thành các frame gọi là MEMRAM, SWAP phân thành các frame gọi là MEMSWAP.

3.2.3 Paging-based address translation scheme

Mỗi một virtual page của tiến trình đều có một **bảng phân trang (page table)**. Bảng phân trang giúp userspace process tìm ra được physical frame của mỗi virtual page được ánh xạ tới. Trong mỗi **page table**, có các **Page Table Entries (PTE)**. Trong các page table entries bao gồm một giá trị 32-bit cho mỗi virtual page, có định nghĩa về data và cấu trúc một PTE như sau:

```
* Bits 0-12  page frame number (PFN) 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
```



Hình 4: Page Table Entry Format

Memory Swapping: Khi thực thi một tiến trình, các nội dung trong page của tiến trình tương ứng sẽ được tải vào bất kỳ frame nào trong bộ nhớ chính, cụ thể ở đây là MEMRAM. Khi máy tính vượt quá lược bộ nhớ chính cho phép, tức là không còn frame trống trong MEMRAM, hệ điều hành sẽ di chuyển những nội dung trong page không mong muốn đến bộ nhớ thứ cấp, cụ thể là vào frame MEMSWAP (swapping out).

Ngược lại, cơ chế swapping in sẽ di chuyển các nội dung của page ta cần dùng để thực thi tiến trình tương ứng, nhưng hiên tại đang ở trong MEMSWAP, vào các frame MEMRAM bộ nhớ chính. Cơ chế thực hiện swapping out trong mã nguồn Bài tập Lớn này được thực hiện theo quy tắc **First In First Out (FIFO)**, tức là nội dung trong page được tải vào frame MEMRAM sớm nhất trong số các page đang ở trong MEMRAM, sẽ được sao chép hoặc chuyển vào frame trong MEMSWAP, sau đó giải phóng bộ nhớ trong frame MEMRAM đó.

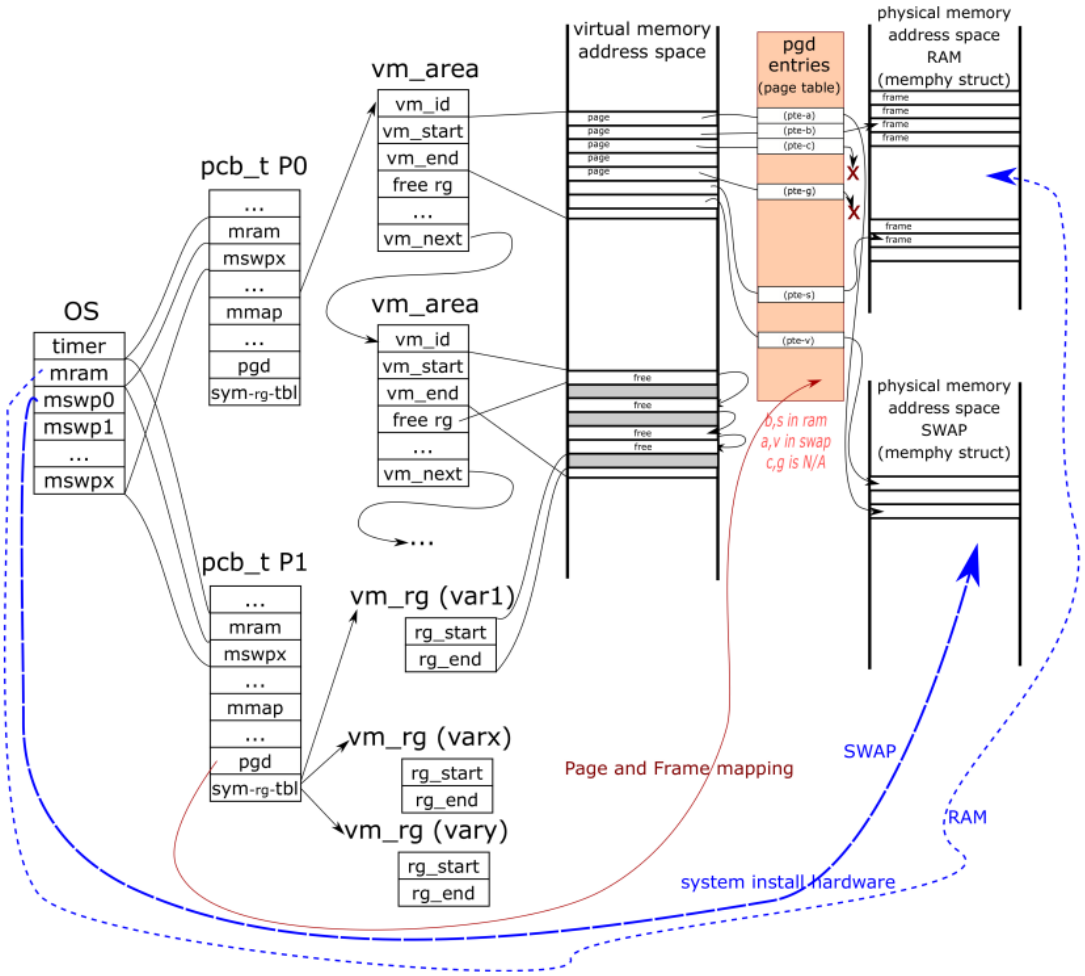
Với ý tưởng ban đầu, do việc truy cập trực tiếp từ CPU lên vùng nhớ thứ cấp là không thể xảy ra, do đó đòi hỏi cần có cơ chế để chuyển nội dung cần truy cập lên bộ nhớ chính. Khi

đó đòi việc lựa chọn **victim page** để thực hiện swapping out nhằm tạo ra khoảng trống trên MEMRAM khi MEMRAM đã đầy. Ở đây ban đầu cơ chế này được hiện thực tuân theo giải thuật thay trang cục bộ (Local Page Replacement) nghĩa là khi muốn truy cập vào vùng nhớ đang thuộc bộ nhớ thứ cấp, chỉ chọn những page nào thuộc quản lý của chính tiến trình đang thực thi làm **victim page**. Tuy nhiên nếu một tiến trình chưa hề có page nào được ánh xạ vào bộ nhớ vật lý, ta sẽ không tìm được **victim page** nào để thay để, dẫn đến việc tiến trình đó sẽ phải chờ đến khi có tiến trình khác đã hoàn thành việc thực thi và để lại khoảng trống trên MEMRAM. Để khắc phục điều này, giải thuật thay trang toàn cục (Global Page Replacement) nghĩa là tiến trình có thể lựa chọn bất kì frame nào đã được ánh xạ và thuộc quản lý của tiến trình khác nhằm tìm kiếm **victim page** điều này sẽ giúp tối ưu hiệu quả sử dụng bộ nhớ chính tuy nhiên cũng sẽ làm xuất hiện sự phụ thuộc của các tiến trình với nhau, ngược lại với việc mong muốn các tiến trình thực thi độc lập và không phụ thuộc lẫn nhau.

Chính vì vậy, nhóm có một giải pháp là kết hợp cả hai giải thuật thay trang toàn cục và cục bộ. Khi tiến trình cần tìm **victim page** để thực hiện cơ chế swapping out, trước hết nó sẽ tìm trong chính các page của nó đã được ánh xạ. Nếu không có page nào thuộc tiến trình đã được ánh xạ trước đó, **victim page** sẽ được chọn là page đã được ánh xạ của một tiến trình nào đó khác, việc page nào thuộc tiến trình nào được chọn sẽ vẫn được thực hiện theo cơ chế FIFO - page nào được ánh xạ vào bộ nhớ vật lý trước sẽ được chọn trước.

3.3 Put It All Together

Sau khi tổng hợp hai phần trên, ta đã có được một hệ điều hành đơn giản với một góc nhìn khái quát qua biểu đồ sau:


OS
timer
mram
mswp0
mswp1
...
mswpx
pcb_t P0
...
mram
mswpx
...
mmap
...
pgd
sym-rg-tbl
pcb_t P1
...
mram
mswpx
...
mmap
...
pgd
sym-rg-tbl
vm_area
vm_id
vm_start
vm_end
free rg
...
vm_next
vm_area
vm_id
vm_start
vm_end
free rg
...
vm_next
...
vm_rg (var1)
rg_start
rg_end
vm_rg (varx)
rg_start
rg_end
vm_rg (vary)
rg_start
rg_end
virtual memory address space
page
free
pgd entries (page table)
(pte-a)
(pte-b)
(pte-c)
(pte-g)
(pte-s)
(pte-v)
b,s in ram
a,v in swap
c,g is N/A
physical memory address space RAM (memphy struct)
frame
physical memory address space SWAP (memphy struct)
Page and Frame mapping
SWAP
RAM
system install hardware

4 Scheduler

4.1 Trả lời câu hỏi

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

Priority Queue là một phiên bản nâng cấp của queue, trong đó mỗi process sẽ có một giá trị priority và các hoạt động của CPU đối với mỗi process sẽ dựa trên số priority đó, thông thường các số có priority thấp sẽ được ưu tiên thực thi bởi CPU trước (số priority thấp tương đương với mức độ ưu tiên cao hơn).

Những ưu điểm của priority queue:

- Vì cơ chế priority thấp được ưu tiên thực hiện trước, thế nên việc thực thi priority queue đảm bảo rằng các process tuy rằng tới sau nhưng được ưu tiên hơn thì nó sẽ được ưu tiên thực hiện trước, phù hợp với các hệ thống thời gian thực hoặc các hệ thống nhúng hoặc các hệ thống hướng user trong đó các ràng buộc về deadline là cực kì quan trọng.
- Trong priority Queue các process sẽ được sắp xếp dựa vào priority, thế nên mỗi khi CPU tiến hành thực thi thì quá trình dequeue sẽ diễn ra nhanh chóng mà không cần duyệt hết toàn bộ queue (process có priority thấp sẽ nằm ở đầu queue).
- Priority Queue cho phép độ ưu tiên của các công việc được điều chỉnh động, có nghĩa là các công việc có thể được gán các ưu tiên khác nhau dựa trên nhu cầu thay đổi của hệ thống. Điều này làm cho nó trở thành một thuật toán định thời linh hoạt có thể thích ứng với các tình huống khác nhau.

4.2 Hiện thực

4.2.1 Hiện thực Queue

4.2.1.a Hiện thực hàm enqueue

Hàm enqueue có chức năng thêm một tiến trình mới vào hàng đợi (queue) đã cho.

Hàm có hai đối số: *struct queue_t *q* - con trỏ tới hàng đợi và *struct pcb_t *proc* - con trỏ tới tiến trình cần được thêm vào hàng đợi.

Trong hàm, đầu tiên kiểm tra xem hàng đợi đã đạt đến kích thước tối đa (*MAX_QUEUE_SIZE*) chưa. Nếu kích thước hiện tại của hàng đợi (*q->size*) nhỏ hơn *MAX_QUEUE_SIZE*, tiến trình được thêm vào hàng đợi bằng cách gán con trỏ tiến trình (*proc*) vào phần tử tiếp theo trong mảng proc của hàng đợi (*q->proc[q->size]*), sau đó tăng kích thước của hàng đợi lên một đơn vị (*q->size++*).

```
1 void enqueue(struct queue_t * q, struct pcb_t * proc) {
2         /* TODO: put a new process to queue [q] */
3         if(q->size<MAX_QUEUE_SIZE)
4         {
5                 q->proc[q->size]= proc;
6                 q->size++;
7         }
8 }
```

4.2.1.b Hiện thực hàm dequeue

Hàm dequeue có chức năng lấy tiến trình có độ ưu tiên cao nhất từ hàng đợi (queue) đã cho và loại bỏ nó khỏi hàng đợi.

Hàm có một đối số là *struct queue_t *q* - con trỏ tới hàng đợi.

Trong hàm, đầu tiên kiểm tra xem hàng đợi có kích thước là 0 không. Nếu kích thước của hàng đợi *(q->size)* là 0, tức là hàng đợi trống, hàm trả về con trỏ *NULL*, không có tiến trình để lấy.

Nếu hàng đợi không rỗng, tiến trình được lấy từ phần tử đầu tiên của mảng *proc* của hàng đợi *(q->proc[0])*. Sau đó, kích thước của hàng đợi được giảm đi một đơn vị *(q->size–)* để chỉ ra rằng một tiến trình đã được loại bỏ khỏi hàng đợi.

Tiếp theo, các phần tử còn lại trong mảng *proc* được dịch chuyển lên một vị trí để lấp đầy vị trí của tiến trình đã bị loại bỏ. Việc này được thực hiện bằng cách sao chép giá trị của phần tử tiếp theo *(q->proc[i+1])* vào phần tử hiện tại *(q->proc[i])*. Quá trình này được lặp lại cho tất cả các phần tử từ 0 đến *q->size - 1*.

Cuối cùng, phần tử cuối cùng của mảng proc được gán giá trị *NULL (q->proc[q->size] = NULL)* để đảm bảo rằng phần tử cuối cùng trong mảng không giữ giá trị không xác định. Hàm trả về con trỏ tới tiến trình có độ ưu tiên cao nhất mà đã được loại bỏ khỏi hàng đợi.

```
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;
        struct pcb_t * temp = q->proc[0];
        q->size--;
        int i;
        for(i=0;i<q->size;i++)
                q->proc[i]=q->proc[i+1];
        q->proc[q->size]=NULL;
   return temp;
}
```

Do được phát triển lên từ giải thuật định thời MLQ, mặc dù tên gọi các hàm, biến, macro,... có liên quan đến MLQ nhưng phần hiện thực thực chất là dành cho MLFQ.

4.2.2 Hiện thực Scheduler

4.2.2.a Hiện thực hàm get_mlq_proc

Hàm get_mlq_proc có chức năng lấy một tiến trình từ hàng đợi ưu tiên (ready_queue) của thuật toán Multilevel Feedback Queue Scheduling (MLFQ). Hàng đợi này chứa các hàng đợi con, mỗi hàng đợi con tương ứng với một mức ưu tiên.

Hàm sử dụng một khóa (queue_lock) để bảo vệ việc truy cập đồng thời vào hàng đợi.

Trong hàm, đầu tiên, một con trỏ proc được khởi tạo với giá trị NULL, đại diện cho tiến trình sẽ được trả về từ hàng đợi.

Sau đó, một khóa mutex (queue_lock) được sử dụng để khóa hàng đợi trước khi thực hiện lấy tiến trình từ hàng đợi ưu tiên.

Hàm duyệt qua từng hàng đợi con trong mảng mlq_ready_queue (được xem như các mức ưu tiên từ cao đến thấp). Nếu hàng đợi con không trống (!empty(&mlq_ready_queue[i])), tức

là có tiến trình trong hàng đợi con này, tiến trình đầu tiên trong hàng đợi con được lấy ra bằng cách gọi hàm dequeue(&mlq_ready_queue[i]). Tiến trình này được gán cho con trỏ proc.

Nếu độ ưu tiên (priority) của tiến trình lấy ra nhỏ hơn giá trị MAX_PRIO - 1 (giá trị ưu tiên thấp nhất), tiến trình đó được tăng giá trị độ ưu tiên lên 1 (proc->priority++). Điều này đảm bảo rằng tiến trình sẽ được đưa vào hàng đợi ưu tiên thấp hơn trong lần lấy tiếp theo.

Sau khi lấy tiến trình từ hàng đợi và cập nhật độ ưu tiên (nếu cần), khóa mutex được giải phóng (pthread_mutex_unlock(&queue_lock)).

Cuối cùng, hàm trả về con trỏ proc, đại diện cho tiến trình đã lấy ra từ hàng đợi ưu tiên. Nếu hàng đợi ưu tiên là rỗng, con trỏ proc sẽ có giá trị NULL.

```
struct pcb_t * get_mlq_proc(void) {
   struct pcb_t * proc = NULL;
   /*TODO: get a process from PRIORITY [ready_queue].
    * Remember to use lock to protect the queue.
    * */
   pthread_mutex_lock(&queue_lock);
   for(int i=0; i < MAX_PRIO; i++){
      if(!empty(&mlq_ready_queue[i])){
         proc=dequeue(&mlq_ready_queue[i]);
         if (proc->priority < MAX_PRIO - 1) proc->priority++;
         break;
      }
   }
   pthread_mutex_unlock(&queue_lock);
   return proc;
}
```

4.2.2.b Hiện thực hàm add_mlq_proc

Hàm add_mlq_proc có chức năng thêm một tiến trình vào hàng đợi ưu tiên cao nhất (mức ưu tiên 0) do theo giải thuật MLFQ khi một tiến trình được nạp vào nó sẽ nằm ở hàng đợi độ ưu tiên cao nhất.

```
void add_mlq_proc(struct pcb_t * proc) {
   pthread_mutex_lock(&queue_lock);
   enqueue(&mlq_ready_queue[0], proc);
   pthread_mutex_unlock(&queue_lock);
}
```

4.2.2.c Hiện thực hàm put_mlq_proc

Hàm put_mlq_proc() được sử dụng để đặt tiến trình vào hàng đợi ưu tiên tương ứng với mức ưu tiên của tiến trình (proc->priority).

```
void put_mlq_proc(struct pcb_t * proc) {
   pthread_mutex_lock(&queue_lock);
   enqueue(&mlq_ready_queue[proc->priority], proc);
```

```
    pthread_mutex_unlock(&queue_lock);
}
```

4.3 Biểu đồ Gantt

4.3.1 Sched_0

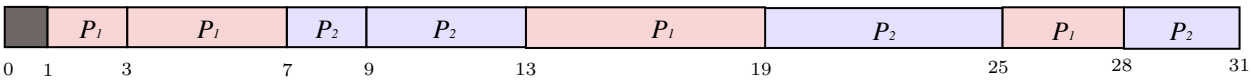
Input:

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

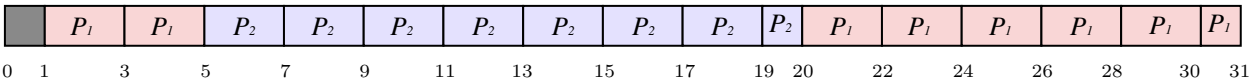
Time	Process	Path	Priority (for MLQ)	Burst time
0	1	s0	4	15
4	2	s0	0	15

Time slice: 2
Number of CPU: 1
Number of Process: 2

Biểu đồ Gantt khi chạy Multilevel Feedback Queue:
CPU 0:



Biểu đồ Gantt khi chạy Multilevel Queue:
CPU 0:



So sánh khi chạy MLQ và MLFQ:

	MLFQ	MLQ
Average Waiting Time	12.5	8.5
Average Turnaround Time	27.5	23.5

Output:

```
ld_routine
Time slot   0
        Loaded a process at input/proc/s0, PID: 1
Time slot   1
        CPU 0: Dispatched process  1 from queue with prio 0
Time slot   2
Time slot   3
        CPU 0: Put process  1 to run queue
```

```
        CPU 0: Dispatched process  1 from queue with prio 1
Time slot   4
        Loaded a process at input/proc/s0, PID: 2
Time slot   5
Time slot   6
Time slot   7
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2 from queue with prio 0
Time slot   8
Time slot   9
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2 from queue with prio 1
Time slot  10
Time slot  11
Time slot  12
Time slot  13
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  1 from queue with prio 2
Time slot  14
Time slot  15
Time slot  16
Time slot  17
Time slot  18
Time slot  19
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2 from queue with prio 2
Time slot  20
Time slot  21
Time slot  22
Time slot  23
Time slot  24
Time slot  25
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  1 from queue with prio 3
Time slot  26
Time slot  27
Time slot  28
        CPU 0: Processed  1 has finished
        CPU 0: Dispatched process  2 from queue with prio 3
Time slot  29
Time slot  30
Time slot  31
        CPU 0: Processed  2 has finished
        CPU 0 stopped
```

Giải thích:

- Thời điểm 0: hệ điều hành nạp process vào ready queue với priority = 0 để chuẩn bị nạp vào CPU
- Thời điểm 1 -> 3: hệ điều hành nạp process 1 vào CPU để xử lí, bởi vì time slice của chương trình là 2 nên sau khi chạy xong 2 khoảng thời gian, process 1 sẽ chuyển tới queue có độ ưu tiên thấp hơn (priority = 1)

- Thời điểm 3 -> 7: do trong ready queue với priority = 0 hiện tại không có process nào nên hệ điều hành chuyển tới ready queue với priority = 1 nên CPU tiếp tục lấy process 1 để xử lí tiếp, bởi vì time slice của ready queue với priority = 1 là 4 đơn vị nên sau khi chạy xong 4 khoảng thời gian, process 1 sẽ chuyển tới queue có priority = 2
- Thời điểm 4: hệ điều hành nạp process 2 vào ready queue với priority = 0 để chuẩn bị xử lí bởi CPU
- Thời điểm 7->9: thời điểm process 2 đã được nạp vào ready queue với priority = 0 (ưu tiên cao nhất) nên CPU ưu tiên xử lý process 2, vì time slice của ready queue với priority = 0 là 2 đơn vị nên sau khi chạy xong 3 khoảng thời gian , process 2 sẽ chuyển tới queue có priority = 1
- Thời điểm 9->13: thời điểm này, ready queue với priority = 0 không còn process nào nên hệ điều hành chuyển tới ready queue với priority = 1 nên CPU tiếp tục lấy process 2 để xử lý tiếp, bởi vì time slice của ready queue với priority = 1 là 4 đơn vị nên sau khi chạy xong 4 khoảng thời gian, process 2 sẽ chuyển tới queue có priority = 2
- Thời điểm 13->19: thời điểm này, 2 process đều ở ready queue với priority = 2, tuy nhiên CPU sẽ ưu tiên xử lý process enqueue trước (process 1) nên CPU sẽ lấy process 1 để xử lý, bởi vì time slice của ready queue với priority = 2 là 6 đơn vị nên sau khi chạy xong 6 khoảng thời gian, process 1 sẽ chuyển tới queue có priority = 3
- Thời điểm 19->25: CPU xử lý process 2 ở ready queue với priority = 2 do tại thời điểm này đây là queue có độ ưu tiên cao nhất có process, bởi vì time slice của ready queue với priority = 2 là 6 đơn vị nên sau khi chạy xong 6 khoảng thời gian, process 2 sẽ chuyển tới queue có priority = 3
- Thời điểm 25->28: thời điểm này, 2 process đều ở ready queue với priority = 3, tuy nhiên CPU sẽ ưu tiên xử lý process enqueue trước (process 1) nên CPU sẽ lấy process 1 để xử lý
- Thời điểm 28->31: process 1 hoàn thành công việc và được xoá khỏi queue, lúc này trong ready queue với priority = 3 chỉ còn process 2 nên CPU sẽ lấy process 2 để xử lý. Process 2 hoàn thành công việc tại thời điểm 31 và được xoá khỏi queue

4.3.2 Sched

Input:

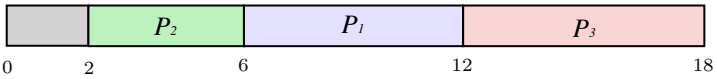
```
4 2 3
0 0 0 0 0
0 p1s 1
1 p1s 0
2 p1s 0
```

Time	Process	Path	Priority (for MLQ)	Burst time
0	1	p1s	1	10
1	2	p1s	0	10
2	3	p1s	0	10

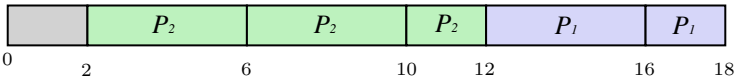
Time slice: 4
Number of CPU: 2
Number of Process: 3

Biểu đồ Gantt khi chạy Multilevel Feedback Queue:
CPU 0:

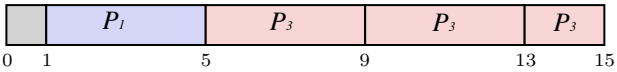
CPU 1:



Biểu đồ Gantt khi chạy Multilevel Queue:
CPU 0:



CPU 1:



So sánh khi chạy MLQ và MLFQ:

	MLFQ	MLQ
Average Waiting Time	4	4
Average Turnaround Time	14	14

Output:

```
ld_routine
Time slot   0
        Loaded a process at input/proc/p1s, PID: 1
```

```
Time slot   1
        CPU 0: Dispatched process  1 from queue with prio 0
        Loaded a process at input/proc/p1s, PID: 2
Time slot   2
        CPU 1: Dispatched process  2 from queue with prio 0
        Loaded a process at input/proc/p1s, PID: 3
Time slot   3
Time slot   4
Time slot   5
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  3 from queue with prio 0
Time slot   6
        CPU 1: Put process  2 to run queue
        CPU 1: Dispatched process  1 from queue with prio 1
Time slot   7
Time slot   8
Time slot   9
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  2 from queue with prio 1
Time slot  10
Time slot  11
Time slot  12
        CPU 1: Processed  1 has finished
        CPU 1: Dispatched process  3 from queue with prio 1
Time slot  13
Time slot  14
Time slot  15
        CPU 0: Processed  2 has finished
        CPU 0 stopped
Time slot  16
Time slot  17
Time slot  18
        CPU 1: Processed  3 has finished
        CPU 1 stopped
```

Giải thích:

- Thời điểm 0: hệ điều hành load process 1 vào ready queue với priority = 0 để chuẩn bị nạp vào CPU xử lí
- Thời điểm 1: CPU 0 lấy process 1 từ ready queue với priority = 0 và bắt đầu xử lí. Đồng thời lúc này, hệ điều hành cũng nạp process 2 vào ready queue với priority = 0.
- Thời điểm 2: CPU 1 lấy process 2 từ ready queue với priority = 0 và bắt đầu xử lí. Đồng thời lúc này, hệ điều hành cũng nạp process 3 vào ready queue với priority = 0.
- Thời điểm 1 -> 5: process 1 được CPU 0 xử lí. Time slice trong trường hợp này là 4, vậy nên đến thời điểm 5, process 1 sẽ kết thúc phiên xử lí và được đẩy xuống ready queue với priority = 1.
- Thời điểm 2 -> 6: ở thời điểm 2, process 1 đang được CPU 0 xử lí, process 3 đang được nạp vào ready queue, CPU 1 sẽ nạp process 2 vào để xử lí. Đến thời điểm 6, process 2 kết thúc phiên xử lí và được đẩy xuống ready queue với priority là 1.

- Thời điểm 5 -> 9: tại thời điểm 5, process 1 hoàn thành phiên xử lí ở CPU 0, process 2 đang được xử lí bởi CPU 1. CPU 0 sẽ nạp process 3 từ ready queue có priority 0 để xử lí cho đến thời điểm 9.
- Thời điểm 6 -> 12: tại thời điểm 6, CPU 1 kết thúc phiên làm việc với process 2, process 2 được đẩy xuống vào ready queue với priority 1. Process 3 đang được CPU 0 xử lí, process 1 enqueue vào ready queue có priority 1 trước nên được CPU 1 xử lý. Tại thời điểm 12, process 1 hoàn thành công việc và được xoá khỏi queue.
- Thời điểm 9 -> 15: tại thời điểm 9, process 1 đang được xử lý bởi CPU 1, process 2 đang enqueue vào ready queue có priority 1 trước nên sẽ được CPU 0 ưu tiên xử lý. Tại thời điểm 15, process 2 hoàn thành công việc và bị xoá khỏi queue.
- Thời điểm 12 -> 18: tại thời điểm 12, process 1 đã thành thành công việc, process 2 đang được CPU 0 xử lý, process 3 trong ready queue nên CPU 1 lấy process 3 từ ready queue với priority 1 để xử lý. Tại thời điểm 18, process 3 hoàn thành công việc và bị xoá khỏi queue.

5 Memory Management

5.1 Trả lời câu hỏi

5.1.1 Question: 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?

- Tính linh hoạt: vì được phân thành nhiều memory segment, thế nên hệ điều hành có thể phân bổ bộ nhớ theo cách linh hoạt hơn dựa vào nhu cầu của các process, có thể cấp phát cho nhiều mục đích như phần về code, stack, heap. Điều này cho phép hệ điều hành tối ưu hơn về việc phân bổ bộ nhớ cho từng process.
- Sự chia sẻ: việc chia thành nhiều memory segment cho phép sự chia sẻ bộ nhớ giữa nhiều process hơn, điều này đặc biệt hữu ích cho việc chia sẽ dữ liệu giữa các process hoặc để triển khai các cơ chế giao tiếp giữa các process, với sự chia sẻ này thì hệ điều hành có thể giảm mức độ sử dụng bộ nhớ và tăng hiệu suất của hệ thống lên.
- Bộ nhớ ảo: việc chia thành nhiều các memory segment có thể được sử dụng để triển khai bộ nhớ ảo, cho phép hệ điều hành quản lý bộ nhớ vật lý hiệu quả hơn, đặc biệt với cơ chế bộ nhớ ảo thì cho phép các process có thể truy cập nhiều bộ nhớ hơn so với bộ nhớ khả dụng về mặt vật lí được cấp phát.
- Giảm phân mảnh: bằng cách chia bộ nhớ thành nhiều phần, điều này giúp cho sự phân mảnh nội giảm , mặc nhiên sự phân mảnh nội vẫn có thể xảy ra nhưng với mức độ cực kì bé vì ta đã chia nhỏ bộ nhớ ra thành các nhỏ. Từ đó giúp cải thiện việc sử dụng bộ nhớ.

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

Trong quản lí bộ nhớ phân trang, việc ta chia địa chỉ thành nhiều hơn 2 cấp độ sẽ làm tăng số lần tra cứu bảng trang cần thiết để chuyển từ virtual address sang physical address, điều này sẽ làm tăng thời gian và chí phí truy cập bộ nhớ.

Nếu trong hệ thống phân trang 2 cấp, phần page number được chia thành 2 phần là p1 và p2 , page table 1 sẽ chưa con trỏ tới page table 2, page table 2 sẽ trỏ tới bộ nhớ vật lí chữa dữ liệu thực tế mà ta cần truy cập.

Việc ta chia địa chỉ thành nhiều hơn 2 cấp, ta sẽ cần thêm các page table để bổ sung vào quá trình truy cập, ví dụ hệ thống phân trang 3 cấp sẽ cần 3 page table p1 p2 p3 . Diều này sẽ làm tăng chi phí hoạt động của quá trình chuyển đổi từ địa chỉ ảo sang địa chỉ thực tế và làm tăng thời gian truy cập bộ nhớ.

Tuy nhiên trong một vài trường hợp thì hệ thống phân trang cấp cao lại có ích, ví dụ trong một số hệ thống lớn thì nếu thực hiện bảng phân trang 1 cấp thì kích thước của bảng phân trang sẽ rất lớn . khi bảng lớn thì việc truy xuất sẽ mất rất nhiều thời gian và quản lí rất khó khăn, việc thực hiện hệ thống phân trang nhiều cấp sẽ giúp chung ta chia nhỏ bảng phân trang lớn ra thành nhiều bảng bé hơn, điều này có thể giúp ta giảm thời gian cần thiết để truy cập bảng phân trang, hơn nữa việc chia nhỏ ra như vậy cung cấp khả năng kiểm soát chi tiết hơn đối với việc cấp phát bộ nhớ.

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

Segmentation kết hợp với paging là một kỹ thuật quản lí bộ nhớ kết hợp ưu điểm của 2 kĩ thuật quản lí bộ nhớ, phân đoạn và phân trang. Kỹ thuật này chia process thành nhiều segment, sau đó mỗi segment sẽ có một page table riêng, và mỗi segment sẽ được chia thành các page, mỗi page sau đó sẽ được map tới physical memory dựa vào page table của mỗi segment, và mỗi segment cũng sẽ có segment table riêng để map segment number tới địa chỉ của page table.

Ưu điểm:

- Tính linh hoạt: Segmentation với paging cung cấp kỹ thuật quản lí bộ nhớ linh hoạt hơn so với từng kỹ thuật riêng lẻ, với segmentation, mỗi segment khác nhau có thể được phân bổ với kích thước khác nhau dựa trên nhu cầu sử dụng, kết hợp với paging cho phép ta sử dụng bộ nhớ một cách hiệu quả.
- Tính bảo vệ: Segmentation với paging cung cấp cho ta sự riêng tư, giữa các segment, không cho các segment truy cập và thay đổi dữ liệu lẫn nhau, điều này giúp tăng cường bảo mật và ổn định, ngăn chặn các hành vi ngoài ý muốn có thể gây ra lỗi.
- Tính chia sẻ: Segmentation với paging hỗ trợ bộ nhớ dùng chung giữa các process, điều này hữu ích cho giao tiếp giữa các process và giúp việc sử dụng bộ nhớ hiệu quả hơn.
- Bộ nhớ ảo: segmentation với paging hỗ trợ việc sử dụng bộ nhớ ảo, giúp cải thiện performance và đạt được các lợi ích của bộ nhớ ảo mang lại cho hệ thống.
- Sự phân mảnh: việc kết hợp paging sẽ giúp giảm đáng kể tình trạng phân mãnh ngoại và phân mãnh nội sẽ được kiểm soát ở mức thấp nhất.

Nhược điểm:

- Sự phức tạp: việc kết hợp segmentation với paging phức tạp hơn so với paging và nó yêu cầu cả sự hỗ trợ từ phần cứng, điều này gây ra sự khó khăn khi hiện thực và bảo trì.
- Phân mảnh: phân mảnh nội vẫn xuất hiện ở segmentation với paging và sự phân mảnh ngoại vẫn có thể xảy ra nếu segment bị chia quá bé.
- Tốn chi phí: Việc hiện thực segmentation với paging yêu cầu chi phí để chứa và duy trì bảng phân trang và phân đoạn, điều này làm tăng mức sử dụng bộ nhớ.

5.1.4 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.

Trong một hệ điều hành, cơ chế đồng bộ hóa liên quan tới việc thực thi đa luồng, đa tiến trình. Khi các luồng hay tiến trình cùng được thực thi và truy cập vô vùng dữ liệu chung thì sẽ dẫn tới các vấn đề như race condition, deadlocks và các vấn đề về đồng bộ dữ liệu dẫn tới sự sai sót về mặt dữ liệu.

Trong hệ điều hành đơn giản nhóm chúng em thực thi, ta có thể thấy sự xử lý đồng bộ được thực hiện thông qua các biến mutex (event_lock, timer_lock) và các biến condition (event_cond, timer_cond), hai biến này thường được kết hợp sử dụng để tạo ra condition variables để đảm bảo sự đồng bộ về bộ đếm thời gian chung. Trong cùng một time slot, tất cả các luồng xử lý đều phải hoàn thành công việc theo đúng time slot, tránh tình trạng luồng thứ nhất thực hiện công việc của time slot 2 trong khi luồng thứ 2 vẫn đang thực hiện công việc của time slot 1, điều này

gây ra sự thiếu thực tế và chính xác của hệ thống.

Một ví dụ điển hình ta có thể nêu ra là nếu ta thực hiện 2 luồng, 1 luồng load process và một luồng khác thực thi, nếu như không có tính đồng bộ về time slot, luồng load thực thi lệnh load của process đến vào thời điểm thứ 7, nếu luồng load nhanh hơn nên khi luồng thực thi mới tới time slot thứ 3 thì trong queue đã có process đến vào thời điểm thứ 7 và cpu sẽ thực thi process đó. Điều này là hoàn toàn sai với lý thuyết thực tế.

Trong hàm get-MLQ-proc hay các hàm chia sẽ vùng dữ liệu chung sử dụng các biến mutex-lock để đảm bảo rằng chỉ có 1 luồng duy nhất thực thi hàm đó tại 1 thời điểm, điều này đảm bảo các vùng dữ liệu chung sẽ được cập nhật một cách chuẩn xác và loại bỏ khả năng bị race condition.

5.2 Hiện thực

5.2.1 Memory management - Physical memory (mm-memphy.c)

5.2.1.a Hàm MEMPHY_dump

Hàm MEMPHY_dump sẽ in ra nội dung của bộ nhớ ra màn hình, do kích thước bộ nhớ có thể rất lớn, việc in tất cả nội dung của bộ nhớ là tương đối dài, do đó ta sẽ chỉ in những địa chỉ ô nhớ có nội dung (khác 0), và để tránh việc nội dung ô nhớ không được in ra khi giá trị được ghi vào là 0, ta sẽ luôn ghi vào bộ nhớ những giá trị khác 0 trong Bài tập lớn này.

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

5.2.2 Memory management (mm.c)

5.2.2.a Hàm alloc_pages_range

Hàm alloc_pages_range có nhiệm vụ tìm ra một danh sách với số lượng cho trước các frame trống trên RAM để ánh xạ các vùng địa chỉ ảo mới được mở rộng ra.

Đầu tiên kiểm tra xem số lượng frame cần thiết có lớn hơn số lượng frame tối đa của RAM hay không, nếu có thì Thrashing đã xảy ra do hệ điều hành sẽ danh phần lớn thời gian cho việc thay trang mà không thực thi các instruction.

Tiếp theo sử dụng hàm MEMPHY_get_freefp() để tìm kiếm frame trống có trên RAM, nếu có thì ta sẽ thêm frame page number của frame đó vào list *frm_lst. Nếu không còn frame trống trên RAM ta sẽ phải thực hiện swapping out một page để tạo khoảng trống trên RAM, ở đây ta sử dụng hàm find_victim_page() để tìm victim_page sẽ được swap out ra khỏi RAM, hàm này sẽ chỉ tìm kiếm những page thuộc chính process đang thực thi (Local Page Replacement). Như đã đề cập ở trên, nếu process chưa có bất kì page nào được ánh xạ vào RAM, Global Page

Replacement sẽ được sử dụng để tiếp tục tìm kiếm frame có thể thay thế. Hiện thực thêm hàm MEMPHY_get_usedfp() trong file mm-memphy.c hàm này đơn giản chỉ trả về thông tin của frame hiện đang có trên RAM được ánh xạ sớm nhất. Khi trường hợp này xảy ra, victim page hiện tại không phải là page của chính process đang thực thi mà là của process khác. Tuy nhiên, sau khi tìm được frame khả thi và có các thông tin của process sở hữu nó, trong cả hai trường hợp Local và Global, ta đều chỉ cần tìm kiếm một frame trống trong bộ nhớ SWAP để chuyển nội dung từ victim page xuống sau đó cập nhật các thuộc tính của mm_struct.

```
int alloc_pages_range(struct pcb_t *caller, int req_pgnum, struct framephy_struct**
    frm_lst)
{
  int pgit, fpn;
  //Thrashing
  if (req_pgnum > (caller->mram->maxsz / PAGING_PAGESZ)) {
    perror("Thrasing!\n");
    exit(EXIT_FAILURE);
  }
  struct framephy_struct *newfp_str;
  for(pgit = 0; pgit < req_pgnum; pgit++)
  {
    if(MEMPHY_get_freefp(caller->mram, &fpn) == 0)
   {
      if (pgit == 0) {
        newfp_str = malloc(sizeof(struct framephy_struct));
        newfp_str->owner = caller->mm;
        newfp_str->fpn = fpn;
        newfp_str->fp_next = NULL;
        *frm_lst = newfp_str;
      }
      else {
        newfp_str->fp_next = malloc(sizeof(struct framephy_struct));
        newfp_str->fp_next->owner = caller->mm;
        newfp_str->fp_next->fpn = fpn;
        newfp_str->fp_next->fp_next = NULL;
        newfp_str = newfp_str->fp_next;
      }
   } else {  // ERROR CODE of obtaining somes but not enough frames
      int vicpgn, swpfpn;
      int vicfpn;
      uint32_t vicpte;
      /* Find victim page */
      if (find_victim_page(caller->mm, &vicpgn) ==0) {
        vicpte = caller->mm->pgd[vicpgn];
        vicfpn = PAGING_FPN(vicpte);
        /* Remove frame from used_fp_list*/
        MEMPHY_remove_usedfp(caller->mram, vicfpn);
        /* Get free frame in MEMSWP */
        MEMPHY_get_freefp(caller->active_mswp, &swpfpn);
        /* Copy victim frame to swap */
        __swap_cp_page(caller->mram, vicfpn, caller->active_mswp, swpfpn);
        pte_set_swap(&caller->mm->pgd[vicpgn], 0, swpfpn);
      }
      else {
```

```
45        /*Get global frame*/
46        struct framephy_struct *fp = MEMPHY_get_usedfp(caller->mram);
47        find_victim_page(fp->owner, &vicpgn);
48        vicpte = fp->owner->pgd[vicpgn];
49        vicfpn = PAGING_FPN(vicpte);
50        /* Get free frame in MEMSWP */
51        MEMPHY_get_freefp(caller->active_mswp, &swpfpn);
52        /* Copy victim frame to swap */
53        __swap_cp_page(caller->mram, fp->fpn, caller->active_mswp, swpfpn);
54        pte_set_swap(&fp->owner->pgd[vicpgn], 0, swpfpn);
55
56      }
57      if (pgit == 0) {
58        newfp_str = malloc(sizeof(struct framephy_struct));
59        newfp_str->owner = caller->mm;
60        newfp_str->fpn = vicfpn;
61        newfp_str->fp_next = NULL;
62        *frm_lst = newfp_str;
63
64      }
65      else {
66        while (newfp_str->fp_next) {
67          newfp_str = newfp_str ->fp_next;
68        }
69        newfp_str->fp_next = malloc(sizeof(struct framephy_struct));
70        newfp_str->fp_next->owner = caller->mm;
71        newfp_str->fp_next->fpn = vicfpn;
72        newfp_str->fp_next->fp_next = NULL;
73        newfp_str = *frm_lst;
74
75      }
76    }
77  }
78  return 0;
79}
```

5.2.2.b Hàm vmap_page_range

Hàm vmap_page_range được gọi sau khi ta đã có một danh sách các frame page number khả thi có thể được ánh xạ, nhiệm vụ của hàm này là thực hiện các bước ánh xạ như cập nhật PTE, thêm thông tin của frame vào usedfp_list của RAM, thêm thông tin của page vào fifo_pgn của mm_struct.

```
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   int pgit = 0;
8   int pgn = PAGING_PGN(addr);
9   /* TODO map range of frame to address space
```

```
10   *      [addr to addr + pgnum*PAGING_PAGESZ
11   *      in page table caller->mm->pgd[]
12   */
13  for (; pgit < pgnum; pgit++){
14    pte_set_fpn(&caller->mm->pgd[pgn + pgit], frames->fpn);
15    MEMPHY_put_usedfp(caller->mram, frames->fpn, caller->mm);
16    frames = frames->fp_next;
17    enlist_pgn_node(&caller->mm->fifo_pgn, pgn+pgit);
18  }
19   /* Tracking for later page replacement activities (if needed)
20    * Enqueue new usage page */
21  return 0;
22 }
```

5.2.3 Memory management - virtual memory (mm-vm.c)

5.2.3.a Hàm __alloc

Hàm alloc được gọi khi có yêu cầu cấp phát vùng nhớ, khi gọi hàm này đầu tiên hàm get_free_vmrg_area() sẽ được gọi nhằm tìm kiếm xem có memory region nào trong virtual memory area có đủ kích thước cần thiết. Nếu có thì ta chỉ cần thêm thông tin của vùng nhớ mới được cấp phát vào symbol table để quản lý.

Tuy nhiên, nếu không có vùng nhớ nào đã được ánh xạ đủ kích thước, ta sẽ phải mở rộng virtual memory area bằng hàm inc_vma_limit() với kích thước bằng một số nguyên lần kích thước trang và lớn hơn hoặc bằng kích thước cần thiết. Khi gọi hàm này vùng địa chỉ ảo được mở rộng sẽ được ánh xạ vào bộ nhớ vật lý thông qua hàm các hàm được hiện thực trong file mm.c

Cuối cùng, đơn giản ta cũng chỉ cần thêm thông tin của vùng nhớ mới được cấp phát vào symbol table để quản lý. Tuy nhiên bên cạnh đó là thêm vùng địa chỉ ảo chưa sử dụng tới (do mỗi lần mở rộng ra luôn mở rộng với kích thước bằng một số nguyên lần của trang) vào vm_freerg_list là một danh sách quản lý các vùng địa chỉ ảo chưa được cấp phát nhưng thuộc những trang đã được ánh xạ vào bộ nhớ vật lý.

```
1 int __alloc(struct pcb_t *caller, int vmaid, int rgid, int size, int *alloc_addr)
2 {
3   /*Allocate at the toproof */
4   struct vm_rg_struct rgnode;
5   if (get_free_vmrg_area(caller, vmaid, size, &rgnode) == 0)
6   {
7     caller->mm->symrgtbl[rgid].rg_start = rgnode.rg_start;
8     caller->mm->symrgtbl[rgid].rg_end = rgnode.rg_end;
9
10    *alloc_addr = rgnode.rg_start;
11    //printf status
12    printf("Allocation for Process %d - size needed %d\n", caller->pid, size);
13    print_pgtbl(caller, 0, -1);
14    print_list_vma(caller->mm->mmap);
15    print_list_rg(caller->mm->mmap->vm_freerg_list);
16    return 0;
17  }
```

```
18
19  /* TODO get_free_vmrg_area FAILED handle the region management (Fig.6)*/
20
21  /*Attempt to increate limit to get space */
22  struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);
23  int inc_sz = PAGING_PAGE_ALIGNSZ(size);
24  //int inc_limit_ret
25  int old_sbrk ;
26  old_sbrk = cur_vma->sbrk;
27
28  /* TODO INCREASE THE LIMIT */
29  inc_vma_limit(caller, vmaid, inc_sz);
30
31  /*Successful increase limit */
32  caller->mm->symrgtbl[rgid].rg_start = old_sbrk;
33  caller->mm->symrgtbl[rgid].rg_end = old_sbrk + size;
34  if (old_sbrk + size < cur_vma->sbrk){
35    struct vm_rg_struct* left_rg = malloc(sizeof(struct vm_rg_struct));
36    left_rg->rg_start = old_sbrk + size;
37    left_rg->rg_end = cur_vma->sbrk;
38    enlist_vm_freerg_list(caller->mm, left_rg);
39  }
40  *alloc_addr = old_sbrk;
41  //printf status
42  printf("Allocation for Process %d - size needed %d\n", caller->pid, size);
43  print_pgtbl(caller, 0, -1);
44  print_list_vma(caller->mm->mmap);
45  print_list_rg(caller->mm->mmap->vm_freerg_list);
46  return 0;
47}
```

5.2.3.b Hàm __free

Hàm __free được gọi khi một vùng nhớ được giải phóng, tuy nhiên, thay vì giải phóng cả vùng nhớ vật lý được ánh xạ vào từ vùng nhớ này, ta chỉ kiểm tra các ngoại lệ có thể xảy ra (region id không hợp lệ, virtual memory area không hợp lệ,...) sau đó nếu không có ngoại lệ gì thì cập nhật symbol table đồng thời thêm vùng địa chỉ ảo do sự giải phóng vừa rồi để lại vào vm_freerg_list để quản lý

```
1 int __free(struct pcb_t *caller, int vmaid, int rgid)
2 {
3   struct vm_rg_struct *rgnode=malloc(sizeof(struct vm_rg_struct));
4   if(rgid < 0 || rgid > PAGING_MAX_SYMTBL_SZ)
5     return -1;
6
7   /* TODO: Manage the collect freed region to freerg_list */
8   struct vm_rg_struct *currg = get_symrg_byid(caller->mm, rgid);
9   struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);
10  if (currg == NULL || cur_vma == NULL) /* Invalid memory identify */
11    return -1;
12  // assign values for rgnode so that it can enlist to freeList
13  rgnode->rg_start = caller->mm->symrgtbl[rgid].rg_start;
```

```
14  rgnode->rg_end = caller->mm->symrgtbl[rgid].rg_end;
15  // free the freed region
16  caller->mm->symrgtbl[rgid].rg_start = 0;
17  caller->mm->symrgtbl[rgid].rg_end = 0;
18  caller->mm->symrgtbl[rgid].rg_next = NULL;
19
20  /*enlist the obsoleted memory region */
21  enlist_vm_freerg_list(caller->mm, rgnode);
22  printf("Free for Process %d: free range [%ld -%ld]\n", caller->pid, rgnode->rg_start,
      rgnode->rg_end);
23  /* Print Status */
24  print_pgtbl(caller, 0, -1);
25  print_list_vma(caller->mm->mmap);
26  print_list_rg(caller->mm->mmap->vm_freerg_list);
27  return 0;
28 }
```

5.2.3.c Hàm pg_getpage

Có tính chất tương tự như hàm alloc_page_range(), hàm pg_getpage() được gọi khi READ-/WRITE được gọi, page number sẽ được truyền vào để tìm ra frame page number tương ứng đã được ánh xạ vào. Tuy nhiên, do thao tác này đòi hỏi việc truy cập vào vùng nhớ vật lý. Nếu frame được ánh xạ tới có trên RAM thì điều đơn giản cần làm là lưu frame page number tương ứng vào *fpn. Ngược lại, nếu frame cần tìm không có trên RAM mà nằm ở bộ nhớ SWAP, ta cần đưa nó lên RAM để có thể được truy cập từ CPU, sự kết hợp giữa Local và Global Page Replacement sẽ được áp dụng, qua đồng thời cập nhật các thông tin cần thiết như tương tự trong hàm alloc_pages_range()

```
1 int pg_getpage(struct mm_struct *mm, int pgn, int *fpn, struct pcb_t *caller)
2 {
3   uint32_t pte = mm->pgd[pgn];
4   if (!PAGING_PAGE_PRESENT(pte)) {
5     return -1;
6   }
7   if (pte&PAGING_PTE_SWAPPED_MASK)
8   { /* Page is not online, make it actively living */
9     int vicpgn, swpfpn;
10    int vicfpn;
11    uint32_t vicpte;
12
13    int tgtfpn = PAGING_SWP(pte);//the target frame storing our variable
14
15    /* TODO: Play with your paging theory here */
16    /* Find victim page */
17    if (find_victim_page(caller->mm, &vicpgn) == 0) {
18      vicpte = caller->mm->pgd[vicpgn];
19      vicfpn = PAGING_FPN(vicpte);
20      /* Remove frame from used_fp_list*/
21      MEMPHY_remove_usedfp(caller->mram, vicfpn);
22      /* Get free frame in MEMSWP */
23      MEMPHY_get_freefp(caller->active_mswp, &swpfpn);
24
```

```
25      /* Do swap frame from MEMRAM to MEMSWP and vice versa*/
26      /* Copy victim frame to swap */
27      __swap_cp_page(caller->mram, vicfpn, caller->active_mswp, swpfpn);
28      /* Copy target frame from swap to mem */
29      __swap_cp_page(caller->active_mswp, tgtfpn, caller->mram, vicfpn);
30
31      /* Update page table */
32      pte_set_swap(&caller->mm->pgd[vicpgn], 0, swpfpn);
33
34      /* Update its online status of the target page */
35      pte_set_fpn(&caller->mm->pgd[pgn], vicfpn);
36      enlist_pgn_node(&caller->mm->fifo_pgn,pgn);
37      MEMPHY_put_usedfp(caller->mram, vicfpn, caller->mm);
38      *fpn = vicfpn;
39    }
40    else {
41      /*Get global frame*/
42      struct framephy_struct *fp = MEMPHY_get_usedfp(caller->mram);
43      find_victim_page(fp->owner, &vicpgn);
44      vicpte = fp->owner->pgd[vicpgn];
45      vicfpn = PAGING_FPN(vicpte);
46      /* Get free frame in MEMSWP */
47      MEMPHY_get_freefp(caller->active_mswp, &swpfpn);
48      /* Copy victim frame to swap */
49      __swap_cp_page(caller->mram, fp->fpn, caller->active_mswp, swpfpn);
50      /* Copy target frame from swap to mem */
51      __swap_cp_page(caller->active_mswp, tgtfpn, caller->mram, vicfpn);
52
53      /* Update page table */
54      pte_set_swap(&fp->owner->pgd[vicpgn], 0, swpfpn);
55
56      /* Update its online status of the target page */
57      pte_set_fpn(&caller->mm->pgd[pgn], vicfpn);
58      enlist_pgn_node(&caller->mm->fifo_pgn,pgn);
59      MEMPHY_put_usedfp(caller->mram, vicfpn, caller->mm);
60      *fpn = vicfpn;
61    }
62  }
63  else *fpn = PAGING_FPN(pte);
64  return 0;
65}
```

5.2.3.d Hàm find_victim_page

Hàm này tìm ra victim page để chuyển đổi nội dung và ánh xạ xuông bộ nhớ SWAP nhằm tạo frame trống trên RAM.

Trong mm_struct - struct quản lý bộ nhớ của mỗi process, fifo_pgn là một danh sách các page number của process đang được ánh xạ vào bộ nhớ vật lý. Danh sách này được cập nhật mỗi khi một page của process được ánh xạ vào một frame trên RAM hoặc một frame đang thuộc quản lý của process bị swapping out xuống bộ nhớ SWAP thì page number tương ứng sẽ được loại ra khỏi danh sách.

Đầu tiên ta kiểm tra nếu danh sách đó rỗng thì trả về -1. Nếu danh sách chỉ có một phần tử thì giải phòng vùng nhớ tương ứng và set danh sách về NULL. Trong những trường hợp còn lại, do các phần tử trong danh sách được thêm vào đầu nên khi lấy phần tử ra ta phải lấy phần từ cuối cùng của danh sách để thỏa mãn FIFO. Page number của victim page được chọn sẽ được lưu trong *retpgn

```
int find_victim_page(struct mm_struct *mm, int *retpgn)
{
  struct pgn_t *pg = mm->fifo_pgn;
  /* TODO: Implement the theorical mechanism to find the victim page */
  if (!pg) {
    return -1;
  }
  if (!pg->pg_next) {
    *retpgn = pg->pgn;
    free(pg);
    mm->fifo_pgn = NULL;
  }
  else {
    struct pgn_t *pre = mm->fifo_pgn;
    while (pre->pg_next->pg_next) {
      pre = pre->pg_next;
      pg = pg->pg_next;
      }
    pg = pg->pg_next;
    pre->pg_next = NULL;
    *retpgn = pg->pgn;
    free(pg);
  }
  return 0;
}
```

5.2.3.e Hàm validate_overlap_vm_area

Hàm validate_overlap_vm_area() có tác dụng kiểm tra xem các vùng địa chỉ ảo (virtual memory area) có bị chồng lên nhau hay không khi các vùng này được mở rộng. Tuy nhiên, ở Bài tập lớn lần này, ta không hiện thực **Segmentation for paging** khi mà một process được chia thành nhiều segment khác nhau, mỗi segment có một VMA riêng và có segment table riêng mà chỉ hiện thực 1 segment duy nhất của toàn bộ process. Chính vì vậy, sẽ không có trường hợp các VMA bị chồng lên nhau trong Bài tập lớn lần này.

```
int validate_overlap_vm_area(struct pcb_t *caller, int vmaid, int vmastart, int vmaend)
{
  struct vm_area_struct *vma = caller->mm->mmap;

  /* TODO validate the planned memory area is not overlapped */
  while (vma) {
    if (vma->vm_id != vmaid && vmaend > vma->vm_start && vmastart < vma->vm_end)
      return -1;
    else vma = vma->vm_next;
  }
```

```
11  return 0;
12 }
```

5.3 Nhắc lại về cách thức hệ điều hành hoạt động khi thực hiện các instruction

5.3.1 ALLOC

Lệnh ALLOC nhận vào hai tham số: kích thước vùng nhớ cần cấp phát và ID của vùng nhớ đó (ID này chính là index trong Symbol Table của process để quản lý vùng nhớ được cấp phát). Khi một process thực thi lệnh ALLOC, hệ điều hành sẽ kiểm tra xem trong vm_freerg_list - một danh sách quản lý các vùng địa chỉ ảo trống có thể được cấp phát cho process, các vùng địa chỉ này đều có điểm chung là chưa được sử dụng bởi process tuy nhiên đều đã được ánh xạ vào trong bộ nhớ vật lý. Nếu trong vm_freerg_list không có vùng địa chỉ ảo nào có có khả năng đáp ứng kích thước cần cấp phát. Vùng địa chỉ ảo của process sẽ được mở rộng.

Mở rộng vùng địa chỉ ảo: Khi vùng địa chỉ ảo được mở rộng, kích thước mở rộng sẽ luôn luôn là một số nguyên lần kích thước mỗi trang, do đó sẽ có khả năng sẽ có vùng địa chỉ ảo dư ra. Khi đó vùng địa chỉ ảo này sẽ được thêm vào vm_freerg_list để quản lý cho việc cấp phát thêm nếu cần thiết sau này.

Ánh xạ vào bộ nhớ vật lý: sau khi mở rộng vùng địa chỉ ảo bằng một số nguyên lần kích thước trang, các trang này sẽ cần phải tìm kiếm những frame còn đang chưa được ánh xạ trên bộ nhớ chính (RAM) để ánh xạ vào. Các frame trống được quản lý bằng danh sách free_fp_list của bộ nhớ vật lý. Nếu tìm thấy được đủ số lượng frame trống cần thiết, các trang địa chỉ ảo mới sẽ được ánh xạ vào bộ nhớ chính bằng việc cập nhật page table của process. Nếu không tìm thấy được frame trống trên bộ nhớ chính (do bộ nhớ đầy), hệ điều hành sẽ thực hiện cơ chế **swapping** (hay còn thường sử dụng là **page in** và **page out**). Cơ chế này sẽ giúp tạo ra các frame trống trên bộ nhớ chính bằng việc ánh xạ lại các trang đang được ánh xạ trên bộ nhớ chính vào bộ nhớ thứ cấp qua đó tạo ra các frame trống trên bộ nhớ chính.

Thay trang cục bộ (Local Page Replacement): việc lựa chọn trang để thay thế (lựa chọn frame trống trên bộ nhớ chính để page out xuống bộ nhớ thứ cấp) được thực hiện chỉ trên những trang của chính process đó, do bản thân mỗi process cũng không biết được sự tồn tại của nhau và mỗi process cũng không thể truy cập được vào vùng quản lý ánh xạ của process khác. Do đó sẽ tồn tại một vấn đề như sau. Khi một process cần cấp phát bộ nhớ khi bộ nhớ đã đầy, bản thân process đó sẽ phải tìm kiếm những trang của chính nó đã được ánh xạ vào bộ nhớ để thay thế, điều này sẽ không thể thực hiện được khi process chưa từng được cấp phát vùng nhớ trước đó.

Thay trang toàn cục (Global Page Replacement): Do đó process đó phải tìm kiếm một frame thuộc process khác để swapping out xuống bộ nhớ SWAP và tạo frame trống trên RAM. Việc hiện thực cơ chế này được hỗ trợ bởi một danh sách các frame đã được sử dụng trong RAM, danh sách này sẽ được lấy ra theo FIFO khi cần thiết, qua đó giúp cải thiện multiprogramming. Tuy nhiên việc này sẽ làm tăng tính phụ thuộc lẫn nhau của các process, một điều không hề mong muốn.

Chính vì vậy, ở Bài tập lớn lần này, nhóm sẽ kết hợp cả hai cơ chế thay trang, khi RAM không còn frame trống, trước hết process sẽ tìm kiếm các page của chính process đó làm victim_page, nếu còn page nào có thể được thay thế, process đó sẽ có thể được sử dụng bất kì frame nào đang có trên RAM (lựa chọn theo FIFO) để sử dụng cho việc ánh xạ. Sự kết hợp này sẽ góp phần cải thiện tính multiprogamming của hệ thống đồng thời hạn chế sự phụ thuộc của các process với nhau

5.3.2 FREE

Khi giải phóng một vùng nhớ, hệ điều hành đơn giản sẽ chuyển vùng địa chỉ ảo vừa được giải phóng vào trong vm_freerg_list cho việc sử cấp phát sau này mà không thu hồi vùng nhớ vật lý được ánh xạ để tránh tạo ra nhiều lỗ trống trong bộ nhớ vật lý.

5.3.3 READ/WRITE

Khi thực hiện việc đọc/ghi một ô nhớ, cần đi đến địa chỉ ảo tương ứng với các tham số truyền vào sau đó ánh xạ vào trong bộ nhớ vật lý, việc đọc/ghi đều đã được hiện thực bằng các hàm có sẵn của bộ nhớ vật lý.

Khi việc ánh xạ nhận được kết quả là vùng nhớ nằm trong bộ nhớ thứ cấp, trước khi thực hiện thao tác đọc/ghi ta cần thực hiện cơ chế swapping để chuyển nội dung của vùng nhớ thứ cấp cần đọc/ghi lên bộ nhớ chính tương tự như khi thực thi ALLOC.

5.4 Quản lý bộ nhớ

5.4.1 os_1_mlq_paging_small_4K

Input:

```
2 4 8
4096 16777216 0 0 0
1 p0s
2 s3
4 m1s
6 s2
7 m0s
9 p1s
11 s0
16 s1
```

Time	Process	Path	Burst time
1	1	p0s	14
2	2	s3	17
4	3	m1s	6
6	4	s2	13
7	5	m0s	6
9	6	p1s	11
11	7	s0	15
16	8	s1	7

Time slice: 2
Number of CPU: 4
Number of Process: 8
Memory RAM size: 4096
Memory SWAP size: 16777216

Output:

```
Time slot   0
ld_routine
Time slot   1
   Loaded a process at input/proc/p0s, PID: 1
Time slot   2
   CPU 3: Dispatched process  1 from queue with prio 0
   Loaded a process at input/proc/s3, PID: 2
Time slot   3
   CPU 2: Dispatched process  2 from queue with prio 0
Allocation for Process 1 - size needed 300
----------------PAGE TABLE AND FREE_RG LIST CONTENT--------------
print_pgtbl: 0 - 512
00000000: 80000000
00000004: 80000001
print_list_vma:
va[0->512]

print_list_rg:
rg[300->512]
rg[0->0]
```

- **Process 1**
- Thực thi lệnh ALLOC cho Process 1, vùng địa chỉ ảo hiện tại của process vẫn có kích thước là 0, với kích thước cần cấp phát là 300 vậy nên cần mở rộng vùng địa chỉ ảo thêm 2 trang, khi đó 2 trang này có địa chỉ bắt đầu lần lượt là 00000000 và 00000004, lần lượt có page table entry (pte) là 80000000 và 80000001.
- PTE bắt đầu bằng 8 do (1000) tương ứng bit PRESENT và SWAPPED lần lượt là 1 và 0, frame được ánh xạ tồn tại trên bộ nhớ vật lý và nằm ở bộ nhớ chính.
- Frame number (FPN) lần lượt là 0 và 1, hai frame trên bộ nhớ chính được ánh xạ lần lượt có ID là 0 và 1
- Vùng địa chỉ ảo có kích thước 2 trang va[0->512]
- Vùng địa chỉ ảo khởi tạo rg[0->0] và vùng địa chỉ ảo chưa sử dụng tới rg[300->512] (do vùng nhớ cần cấp phát chỉ là 300)

```
Time slot   4
   CPU 3: Put process  1 to run queue
   CPU 3: Dispatched process  1 from queue with prio 1
Allocation for Process 1 - size needed 300
----------------PAGE TABLE AND FREE_RG LIST CONTENT--------------
print_pgtbl: 0 - 1024
00000000: 80000000
00000004: 80000001
00000008: 80000002
00000012: 80000003
print_list_vma:
va[0->1024]

```

```
print_list_rg:
rg[812->1024]
rg[300->512]
rg[0->0]
```

- **Process 1**
- Kích thước vùng địa chỉ cần thiết là 300, tuy nhiên trong vm_freerg_list chỉ có rg[300-512] có kích thường là 212, do đó cần mở rộng vùng địa chỉ ảo thêm 2 trang. Khi này vùng địa chỉa ảo sẽ là va[0->1024] tương đương 4 trang
- Hai vùng nhớ vật lý mới vừa được ánh xạ vào cũng nằm trên bộ nhớ chính (bắt đầu là 8)
- Frame number (FPN) lần lượt là 2 và 3, hai frame trên bộ nhớ chính được ánh xạ lần lượt có ID là 2 và 3
- Vùng địa chỉ ảo có kích thước 2 trang va[0->512]
- Vùng địa chỉa ảo chưa sử dụng tới [300->512] và có thêm rg[812->1024]

```
   Loaded a process at input/proc/m1s, PID: 3
Time slot   5
   CPU 1: Dispatched process  3 from queue with prio 0
Allocation for Process 3 - size needed 300
----------------PAGE TABLE AND FREE_RG LIST CONTENT--------------
print_pgtbl: 0 - 512
00000000: 80000004
00000004: 80000005
print_list_vma:
va[0->512]

print_list_rg:
rg[300->512]
rg[0->0]
```

- **Process 3**
- Process 3 tương tự cần cấp phát vùng nhớ có kích thước 300, tương tự như với process 1, ta cũng có vùng địa chỉ ảo là [0->512] và các vùng địa chỉ ảo trống như cũ.
- Hai vùng nhớ vật lý mới vừa được ánh xạ vào cũng nằm trên bộ nhớ chính (bắt đầu là 8)
- Frame number (FPN) lần lượt là 4 và 5, hai frame trên bộ nhớ chính được ánh xạ lần lượt có ID là 4 và 5. Do các frame có ID từ 0 đến 3 đang thuộc Process 1

```
   CPU 2: Put process  2 to run queue
   CPU 2: Dispatched process  2 from queue with prio 1
Free for Process 1: free range [0 -300]
----------------PAGE TABLE AND FREE_RG LIST CONTENT--------------
print_pgtbl: 0 - 1024
00000000: 80000000
```

```
00000004: 80000001
00000008: 80000002
00000012: 80000003
print_list_vma:
va[0->1024]

print_list_rg:
rg[0->300]
rg[812->1024]
rg[300->512]
rg[0->0]
```

- **Process 1**
- Vùng nhớ [0->300] được giải phóng và được thêm vào vm_freerg_list
- Các PTE không có sự thay đổi do ta không thu hồi vùng nhớ vật lý

```
Time slot   6
Allocation for Process 1 - size needed 100
----------------PAGE TABLE AND FREE_RG LIST CONTENT--------------
print_pgtbl: 0 - 1024
00000000: 80000000
00000004: 80000001
00000008: 80000002
00000012: 80000003
print_list_vma:
va[0->1024]

print_list_rg:
rg[100->300]
rg[812->1024]
rg[300->512]
rg[0->0]
```

- **Process 1**
- Kích thước vùng địa chỉ cần thiết là 100, trong vm_freerg_list hiện tại có rg[0-300] (do đã được giải phóng) có kích thường là 300, do đó có thể sử dụng vùng địa chỉ ảo này do có kích thước đáp ứng được. Do đó vm_freerg_list còn lại vùng rg[100->300] do rg[0-100] đã được sử dụng, bên cạnh đó vẫn còn các vùng địa chỉ ảo trống là rg[812-1024] và rg[300-512]
- Các PTE không có sự thay đổi do ta không thu hồi vùng nhớ vật lý

```
Allocation for Process 3 - size needed 100
----------------PAGE TABLE AND FREE_RG LIST CONTENT--------------
print_pgtbl: 0 - 512
00000000: 80000004
00000004: 80000005
print_list_vma:
va[0->512]
```

```
8
9print_list_rg:
10rg[400->512]
11rg[0->0]
```

- **Process 3**
- Kích thước vùng địa chỉ cần thiết là 100, trong vm_freerg_list chỉ có rg[300-512] có kích thường là 212, do đó có thể sử dụng vùng địa chỉ ảo này do có kích thước đáp ứng được. Do đó vm_freerg_list còn lại vùng rg[400->512] do rg[300-400] đã được sử dụng
- Còn lại, vùng địa chỉ ảo, PTE,... không có sự thay đổi

```
1   Loaded a process at input/proc/s2, PID: 4
2
3Time slot   7
4   CPU 0: Dispatched process  4 from queue with prio 0
5write region=1 offset=20 value=100
6print_pgtbl: 0 - 1024
700000000: 80000000
800000004: 80000001
900000008: 80000002
1000000012: 80000003
11----------------MEMORY CONTENT--------------
12Address: Content
130x00000014: 00000064
```

- **Process 1**
- Giá trị 100 (0x00000064) được ghi vào ô nhớ trong bộ nhớ vật lý được ánh xạ từ địa chỉ ảo với offset 20 của memory region có ID là 1
- Địa chỉ ô nhớ tương ứng trong bộ nhớ vật lý là 0x00000014 với nội dung được ghi vào tương ứng.

```
1   CPU 1: Put process  3 to run queue
2   CPU 1: Dispatched process  3 from queue with prio 1
3Free for Process 3: free range [0 -300]
4----------------PAGE TABLE AND FREE_RG LIST CONTENT--------------
5print_pgtbl: 0 - 512
600000000: 80000004
700000004: 80000005
8print_list_vma:
9va[0->512]
10
11print_list_rg:
12rg[0->300]
13rg[400->512]
14rg[0->0]
15
```

- **Process 3**
- Vùng nhớ [0->300] được giải phóng và được thêm vào vm_freerg_list
- Các PTE không có sự thay đổi do ta không thu hồi vùng nhớ vật lý

```
   Loaded a process at input/proc/m0s, PID: 5
Time slot   8
Allocation for Process 3 - size needed 100
----------------PAGE TABLE AND FREE_RG LIST CONTENT--------------
print_pgtbl: 0 - 512
00000000: 80000004
00000004: 80000005
print_list_vma:
va[0->512]

print_list_rg:
rg[100->300]
rg[400->512]
rg[0->0]
```

- **Process 3**
- Tương tự đối với process 3 khi cần vùng địa chỉ có kích thước 100, vùng rg[0-300] (do đã được giải phóng) có kích thường là 300, do đó có thể sử dụng vùng địa chỉ ảo này do có kích thước đáp ứng được. Do đó vm_freerg_list còn lại vùng rg[100->300] do rg[0-100] đã được sử dụng, bên cạnh đó vẫn còn các vùng địa chỉ ảo trống là rg[400-512]
- Các PTE không có sự thay đổi do ta không thu hồi vùng nhớ vật lý

```
   CPU 3: Put process  1 to run queue
   CPU 3: Dispatched process  5 from queue with prio 0
Allocation for Process 5 - size needed 300
----------------PAGE TABLE AND FREE_RG LIST CONTENT--------------
print_pgtbl: 0 - 512
00000000: 80000006
00000004: 80000007
print_list_vma:
va[0->512]

print_list_rg:
rg[300->512]
rg[0->0]
```

- **Process 5**
- Process 5 tương tự như process 1 và 3 cần cấp phát vùng nhớ có kích thước 300, tương tự như với process 1, ta cũng có vùng địa chỉ ảo là [0->512] và các vùng địa chỉ ảo trống như cũ.
- Hai vùng nhớ vật lý mới vừa được ánh xạ vào cũng nằm trên bộ nhớ chính (bắt đầu là 8)

- Frame number (FPN) lần lượt là 4 và 5, hai frame trên bộ nhớ chính được ánh xạ lần lượt có ID là 6 và 7. Do các frame có ID từ 0 đến 3 đang thuộc Process 1, frame có ID 4 và 5 thuộc Process 3

```
Time slot   9
Free for Process 3: free range [0 -100]
----------------PAGE TABLE AND FREE_RG LIST CONTENT--------------
print_pgtbl: 0 - 512
00000000: 80000004
00000004: 80000005
print_list_vma:
va[0->512]

print_list_rg:
rg[0->100]
rg[100->300]
rg[400->512]
rg[0->0]
```

- **Process 3**
- Vùng nhớ [0->100] được giải phóng và được thêm vào vm_freerg_list
- Các PTE không có sự thay đổi do ta không thu hồi vùng nhớ vật lý

```
   CPU 2: Put process  2 to run queue
Allocation for Process 5 - size needed 100
----------------PAGE TABLE AND FREE_RG LIST CONTENT--------------
print_pgtbl: 0 - 512
00000000: 80000006
00000004: 80000007
print_list_vma:
va[0->512]

print_list_rg:
rg[400->512]
rg[0->0]
```

- **Process 5**
- Tương tự đối với Process 5 khi cần vùng địa chỉ có kích thước 100, vùng địa chỉ rg[300-400] sẽ được lấy để sử dụng và còn lại vùng rg[400-512]

```
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  4 from queue with prio 1
   CPU 2: Dispatched process  1 from queue with prio 2
read region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000000
00000004: 80000001
```

```
00000008: 80000002
00000012: 80000003
----------------MEMORY CONTENT--------------
Address: Content
0x00000014: 00000064
```

- **Process 1**
- Giá trị đọc được từ ô nhớ trong bộ nhớ vật lý có địa chỉ ảo ánh xạ vào từ memory region ID = 1 và offset = 20 vào.
- Giá trị này chính là giá trị ta ghi vào với các tham số tương ứng

```
   Loaded a process at input/proc/p1s, PID: 6
Time slot  10
write region=2 offset=20 value=102
print_pgtbl: 0 - 1024
00000000: 80000000
00000004: 80000001
00000008: 80000002
00000012: 80000003
Segmentation Fault
```

- **Process 1**
- Ghi giá trị 102 vào ô nhớ trong bộ nhớ vật lý có địa chỉ ảo ánh xạ vào từ memory region ID = 2 và offset = 20.
- Tuy nhiên chưa có vùng địa chỉ ảo nào được cấp phát có ID là 2, vậy nên việc truy cập này là không hợp lệ.

```
Free for Process 3: free range [300 -400]
----------------PAGE TABLE AND FREE_RG LIST CONTENT--------------
print_pgtbl: 0 - 512
00000000: 80000004
00000004: 80000005
print_list_vma:
va[0->512]

print_list_rg:
rg[300->400]
rg[0->100]
rg[100->300]
rg[400->512]
rg[0->0]
```

- **Process 3**
- Vùng nhớ [0->100] và [300->400] được giải phóng và được thêm vào vm_freerg_list

- Các PTE không có sự thay đổi do ta không thu hồi vùng nhớ vật lý

```
   CPU 3: Put process  5 to run queue
   CPU 3: Dispatched process  6 from queue with prio 0
Time slot  11
   Loaded a process at input/proc/s0, PID: 7
Segmentation Fault
read region=2 offset=20 value=0
print_pgtbl: 0 - 1024
00000000: 80000000
00000004: 80000001
00000008: 80000002
00000012: 80000003
----------------MEMORY CONTENT--------------
Address: Content
0x00000014: 00000064
```

- **Process 1**
- Đọc giá trị từ ô nhớ trong bộ nhớ vật lý có địa chỉ ảo ánh xạ vào từ memory region ID = 2 và offset = 20.
- Tuy nhiên chưa có vùng địa chỉ ảo nào được cấp phát có ID là 2 như đã đề cập ở trước, vậy nên việc truy cập này là không hợp lệ, giá trị đọc được sẽ là 0.

```
   CPU 1: Processed  3 has finished
   CPU 1: Dispatched process  7 from queue with prio 0
Time slot  12
write region=3 offset=20 value=103
print_pgtbl: 0 - 1024
00000000: 80000000
00000004: 80000001
00000008: 80000002
00000012: 80000003
Segmentation Fault
```

- **Process 1**
- Tương tự, ghi giá trị 103 vào ô nhớ trong bộ nhớ vật lý có địa chỉ ảo ánh xạ vào từ memory region ID = 3 và offset = 20.
- Tuy nhiên cũng chưa có vùng địa chỉ ảo nào được cấp phát có ID là 3, vậy nên việc truy cập này cũng không hợp lệ.

```
   CPU 3: Put process  6 to run queue
   CPU 3: Dispatched process  5 from queue with prio 1
Free for Process 5: free range [0 -300]
----------------PAGE TABLE AND FREE_RG LIST CONTENT--------------
print_pgtbl: 0 - 512
00000000: 80000006
```

```
00000004: 80000007
print_list_vma:
va[0->512]

print_list_rg:
rg[0->300]
rg[400->512]
rg[0->0]
```

- **Process 5**
- Vùng nhớ [0->300] được giải phóng và được thêm vào vm_freerg_list
- Các PTE không có sự thay đổi do ta không thu hồi vùng nhớ vật lý

```
Time slot  13
   CPU 1: Put process  7 to run queue
   CPU 1: Dispatched process  6 from queue with prio 1
   CPU 0: Put process  4 to run queue
   CPU 0: Dispatched process  7 from queue with prio 1
Segmentation Fault
read region=3 offset=20 value=0
print_pgtbl: 0 - 1024
00000000: 80000000
00000004: 80000001
00000008: 80000002
00000012: 80000003
----------------MEMORY CONTENT--------------
Address: Content
0x00000014: 00000064
```

- **Process 1**
- Tương tự, tiếp tục đọc giá trị từ ô nhớ trong bộ nhớ vật lý có địa chỉ ảo ánh xạ vào từ memory region ID = 3 và offset = 20.
- Tuy nhiên cũng vẫn chưa có vùng địa chỉ ảo nào được cấp phát có ID là 3 như đã đề cập ở trước, vậy nên việc truy cập này là không hợp lệ, giá trị đọc được sẽ là 0.

```
Allocation for Process 5 - size needed 100
----------------PAGE TABLE AND FREE_RG LIST CONTENT--------------
print_pgtbl: 0 - 512
00000000: 80000006
00000004: 80000007
print_list_vma:
va[0->512]

print_list_rg:
rg[100->300]
rg[400->512]
rg[0->0]
```

- **Process 5**
- Tiếp tục yêu cầu vùng địa chỉ có kích thước là 100, ta có thể sử dụng ngay vùng [0->100] trong vùng địa chỉ trống [0->300], do đó vm_freerg_list còn lại các vùng rg[100->300] và rg[400->512]

```
Time slot  14
write region=1 offset=20 value=102
print_pgtbl: 0 - 512
00000000: 80000006
00000004: 80000007
----------------MEMORY CONTENT--------------
Address: Content
0x00000014: 00000064
0x00000740: 00000066
```

- **Process 5**
- Giá trị 102 (0x00000066) được ghi vào ô nhớ trong bộ nhớ vật lý được ánh xạ từ địa chỉ ảo với offset 20 của memory region có ID là 1
- Địa chỉ ô nhớ tương ứng trong bộ nhớ vật lý là 0x00000740 với nội dung được ghi vào tương ứng.

```
Time slot  15
   CPU 2: Put process  1 to run queue
        CPU 2: Dispatched process  2 from queue with prio 2
write region=2 offset=1000 value=1
print_pgtbl: 0 - 512
00000000: 80000006
00000004: 80000007
Offset out of range
----------------MEMORY CONTENT--------------
Address: Content
0x00000014: 00000064
0x00000740: 00000066
```

- **Process 5**
- Giá trị 1 (0x0000001) được ghi vào ô nhớ trong bộ nhớ vật lý được ánh xạ từ địa chỉ ảo với offset 1000 của memory region có ID là 2
- Do địa chỉ ảo tương ứng đã vượt quá vùng địa chỉ ảo hiện tại (VMA) do đó việc truy cập là không hợp lệ (Offset out of range).

```
Time slot  16
   CPU 3: Processed  5 has finished
   Loaded a process at input/proc/s1, PID: 8
   CPU 3: Dispatched process  8 from queue with prio 0
Time slot  17
```

```
   CPU 1: Put process  6 to run queue
   CPU 1: Dispatched process  4 from queue with prio 2
   CPU 0: Put process  7 to run queue
   CPU 0: Dispatched process  6 from queue with prio 2
Time slot  18
   CPU 3: Put process  8 to run queue
   CPU 3: Dispatched process  8 from queue with prio 1
Time slot  19
Time slot  20
Time slot  21
   CPU 2: Put process  2 to run queue
   CPU 2: Dispatched process  7 from queue with prio 2
   CPU 3: Put process  8 to run queue
   CPU 3: Dispatched process  8 from queue with prio 2
Time slot  22
   CPU 0: Processed  6 has finished
   CPU 0: Dispatched process  1 from queue with prio 3
Free for Process 1: free range [512 -812]
----------------PAGE TABLE AND FREE_RG LIST CONTENT--------------
print_pgtbl: 0 - 1024
00000000: 80000000
00000004: 80000001
00000008: 80000002
00000012: 80000003
print_list_vma:
va[0->1024]

print_list_rg:
rg[512->812]
rg[100->300]
rg[812->1024]
rg[300->512]
rg[0->0]
```

- **Process 1**
- Vùng nhớ [512->812] được giải phóng và được thêm vào vm_freerg_list
- Các PTE không có sự thay đổi do ta không thu hồi vùng nhớ vật lý

```
   CPU 3: Processed  8 has finished
Time slot  23
   CPU 1: Put process  4 to run queue
   CPU 1: Dispatched process  2 from queue with prio 3
   CPU 3: Dispatched process  4 from queue with prio 3
   CPU 3: Processed  4 has finished
Time slot  24
   CPU 0: Processed  1 has finished
   CPU 3 stopped
   CPU 0 stopped
Time slot  25
Time slot  26
```

```
Time slot  27
   CPU 2: Put process  7 to run queue
   CPU 2: Dispatched process  7 from queue with prio 3
Time slot  28
   CPU 1: Processed  2 has finished
   CPU 1 stopped
Time slot  29
Time slot  30
   CPU 2: Processed  7 has finished
   CPU 2 stopped
```

- Các lệnh CALC sẽ được thực thi mà không làm theo đổi gì đến quá trình quản lý bộ nhớ.
- Các process sẽ thực hiện nốt các lệnh và sau đó finish, khi hoàn tất việc thực thi, các process sẽ thu hồi vùng nhớ vật lý được ánh xạ để nhường chỗ cho các process đến sau.
- Có thể thấy, output trả về kết quả tương đối phù hợp với cơ chế hoạt động quản lý bộ nhớ của hệ điều hành, các frame trống sẽ được ánh xạ theo thứ tự, khi process kết thúc thì bộ nhớ vật lý sẽ thu hồi các frame để sử dụng cho các process đến sau.
- Do chưa xuất hiện cơ chế swapping nên thực thi READ/WRITE đều chưa làm theo đổi việc ánh xạ giữa địa chỉ ảo đến bộ nhớ vật lý.
- Việc quản lý vùng địa chỉ ảo cũng hoạt động đúng với mong muốn khi việc cấp phát vùng địa chỉ ảo và bộ nhớ diễn ra.
- Ở ví dụ này, cơ chế SWAP chưa xuất hiện do RAM vẫn đủ khoảng trống cho tất cả các process. Ví dụ tiếp theo sẽ làm rõ hơn cơ chế này.

5.4.2 mytc

Ví dụ này sẽ tập trung làm rõ cơ chế swapping được hiện thực, những cơ chế đã được hiện thực cho ra kết quả tương đối như mong muốn ở ví dụ trước sẽ được lược qua.

Input:

```
2 1 2
1024 16777216 0 0 0
0 swap1
4 swap2
```

Process swap1

```
1 10
calc
alloc 300 0
alloc 300 4
free 0
alloc 300 1
write 100 0 20
read 0 20 20
calc
calc
calc
```

Process swap2

```
1 6
alloc 300 0
alloc 300 1
free 0
alloc 400 2
write 102 1 20
calc
```

Time	Process	Path	Burst time
0	1	swap1	10
4	2	swap2	6

Time slice: 2
Number of CPU: 1
Number of Process: 2
Memory RAM size: 1024
Memory SWAP size: 16777216

Ở ví dụ này, kích thước của RAM khởi tạo rất nhỏ (1024B = 4 frames) nhằm tăng khả năng xảy ra swap.

```
Time slot   0
ld_routine
   Loaded a process at input/proc/swap1, PID: 1
Time slot   1
   CPU 0: Dispatched process  1 from queue with prio 0
Time slot   2
Allocation for Process 1 - size needed 300
----------------PAGE TABLE AND FREE_RG LIST CONTENT--------------
print_pgtbl: 0 - 512
00000000: 80000000
00000004: 80000001
print_list_vma:
va[0->512]

print_list_rg:
rg[300->512]
rg[0->0]

Time slot   3
   CPU 0: Put process  1 to run queue
   CPU 0: Dispatched process  1 from queue with prio 1
Allocation for Process 1 - size needed 300
----------------PAGE TABLE AND FREE_RG LIST CONTENT--------------
print_pgtbl: 0 - 1024
00000000: 80000000
00000004: 80000001
00000008: 80000002
00000012: 80000003
print_list_vma:
```

```
va[0->1024]

print_list_rg:
rg[812->1024]
rg[300->512]
rg[0->0]
```

- **Process 1**
- Sau 2 lần ALLOC, process 1 có vùng địa chỉ ảo với kích thước là 4 page, và cả 4 page này đều được ánh xạ vào RAM, RAM lúc này đã đầy do kích thước của RAM cũng chỉ là 4 frame.

```
Time slot   4
Free for Process 1: free range [0 -300]
----------------PAGE TABLE AND FREE_RG LIST CONTENT--------------
print_pgtbl: 0 - 1024
00000000: 80000000
00000004: 80000001
00000008: 80000002
00000012: 80000003
print_list_vma:
va[0->1024]

print_list_rg:
rg[0->300]
rg[812->1024]
rg[300->512]
rg[0->0]

   Loaded a process at input/proc/swap2, PID: 2
Time slot   5
Allocation for Process 1 - size needed 300
----------------PAGE TABLE AND FREE_RG LIST CONTENT--------------
print_pgtbl: 0 - 1024
00000000: 80000000
00000004: 80000001
00000008: 80000002
00000012: 80000003
print_list_vma:
va[0->1024]

print_list_rg:
rg[812->1024]
rg[300->512]
rg[0->0]
```

- **Process 1**
- Sau khi được FREE và ALLOC lại cùng một vùng nhớ có kích thước 300, trạng thái của bộ nhớ và ánh xạ không có gì thay đổi.

```
Time slot   6
write region=0 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000000
00000004: 80000001
00000008: 80000002
00000012: 80000003
Segmentation Fault
```

- **Process 1**
- Segmentation Fault do ban đầu memory region có ID là 0 được ALLOC nhưng sau được được FREE và ALLOC lại với ID là 1, do đó việc region ID ở đây là 0 không hợp lệ.

```
Time slot   7
   CPU 0: Put process  1 to run queue
   CPU 0: Dispatched process  2 from queue with prio 0
Allocation for Process 2 - size needed 300
----------------PAGE TABLE AND FREE_RG LIST CONTENT--------------
print_pgtbl: 0 - 512
00000000: 80000000
00000004: 80000001
print_list_vma:
va[0->512]

print_list_rg:
rg[300->512]
rg[0->0]
```

- **Process 2**
- Do process 1 đã sử dụng hết tất cả các frame trên RAM, process 2 buộc phải tìm kiếm victim page từ chính các page của mình. Tuy nhiên do process 2 chưa có bất kì page nào được ánh xạ vào RAM do đó process 2 phải lấy các frame do process 1 đang quản lý và chuyển ánh xạ của các page thuộc process 1 xuống bộ nhớ SWAP.
- Do 2 page đầu tiên được ánh xạ của process 1 được ánh xạ vào frame có ID là 0 và 1 nên 2 frame này theo thứ tự FIFO sẽ được nhường cho process 2

```
Time slot   8
Allocation for Process 2 - size needed 300
----------------PAGE TABLE AND FREE_RG LIST CONTENT--------------
print_pgtbl: 0 - 1024
00000000: c0000040
00000004: c0000060
00000008: 80000000
00000012: 80000001
print_list_vma:
va[0->1024]

```

```
print_list_rg:
rg[812->1024]
rg[300->512]
rg[0->0]
```

- **Process 2**
- Do trên RAM cũng không còn frame nào trống nên process 2 phải tự tìm kiếm victim page của chính mình, khi này process 2 đã có 2 page vừa được map ở time slot 7 đã có sẵn trên RAM, do đó 2 page này sẽ được swapping out ra khỏi RAM xuống bộ nhớ SWAP để nhường chỗ cho 2 page mới ánh xạ.
- Do đó 2 page mới sẽ được ánh xạ vào các frame có ID là 0 và 1. Hai victim page ban đầu sẽ được ánh xạ xuống bộ nhớ SWAP (có PTE bắt đầu là 'c' tương đương 4 bit lớn nhất là 1100, bit swap được set lên 1).
- Các frame number ở bộ nhớ SWAP vừa được chọn là 2 và 3 (c0000040 tương đương frame number = 2 và c0000060 tương đương frame number =3) do các frame có ID là 0 và 1 đã được sử dụng bởi process 1 khi nhường frame trên RAM cho process 2 ở time slot 7

```
Time slot   9
   CPU 0: Put process  2 to run queue
   CPU 0: Dispatched process  2 from queue with prio 1
Free for Process 2: free range [0 -300]
----------------PAGE TABLE AND FREE_RG LIST CONTENT--------------
print_pgtbl: 0 - 1024
00000000: c0000040
00000004: c0000060
00000008: 80000000
00000012: 80000001
print_list_vma:
va[0->1024]

print_list_rg:
rg[0->300]
rg[812->1024]
rg[300->512]
rg[0->0]

Time slot  10
Allocation for Process 2 - size needed 400
----------------PAGE TABLE AND FREE_RG LIST CONTENT--------------
print_pgtbl: 0 - 1536
00000000: c0000040
00000004: c0000060
00000008: c0000080
00000012: c00000a0
00000016: 80000000
00000020: 80000001
print_list_vma:
va[0->1536]

```

```
print_list_rg:
rg[1424->1536]
rg[0->300]
rg[812->1024]
rg[300->512]
rg[0->0]
```

- **Process 2**
- Do trên RAM cũng không còn frame nào trống nên process 2 phải tự tìm kiếm victim page của chính mình, khi này process 2 đã có 2 page trên bộ nhớ SWAP và 2 page vừa được ánh xạ ở time slot 8 đã có sẵn trên RAM, do đó 2 page này sẽ được swapping out ra khỏi RAM xuống bộ nhớ SWAP để nhường chỗ cho 2 page mới ánh xạ.
- Do đó 2 page mới sẽ được ánh xạ vào các frame có ID là 0 và 1. Hai victim page ban đầu sẽ được ánh xạ xuống bộ nhớ SWAP
- Các frame number ở bộ nhớ SWAP được chọn tiếp theo là 4 và 5

```
Time slot  11
write region=1 offset=20 value=102
print_pgtbl: 0 - 1536
00000000: c0000040
00000004: c0000060
00000008: c0000080
00000012: c00000a0
00000016: 80000000
00000020: 80000001
----------------MEMORY CONTENT--------------
Address: Content
0x00000014: 00000066
Time slot  12
Time slot  13
   CPU 0: Processed  2 has finished
   CPU 0: Dispatched process  1 from queue with prio 2
Segmentation Fault
read region=0 offset=20 value=0
print_pgtbl: 0 - 1024
00000000: c0000000
00000004: c0000020
00000008: 80000002
00000012: 80000003
----------------MEMORY CONTENT--------------
Address: Content
0x00000014: 00000066
```

- **Process 1**
- Trạng thái của process 1 sau khi nhường 2 frame có ID là 0 và 1 trên RAM cho process 2, 2 page ban đầu được ánh xạ và các frame được nhường đã được swapping out xuống bộ nhớ SWAP và có frame number là 0 và 1 (tương ứng với c0000000 và c0000020)

```
Time slot  14
Time slot  15
Time slot  16
Time slot  17
   CPU 0: Processed  1 has finished
   CPU 0 stopped
```

- Cuối cùng, các process thực thi nốt cách lệnh còn lại và hoàn tất,
- Có thể thấy cơ chế swapping với sự kết hợp của Local và Global Page Replacement đã được hiện thực cho ra kết quả như mong đợi, giúp tận dụng tối đa kích thước của RAM.

6 Kết luận

Qua bài tập lớn lần này, nhóm đã hiện thực cơ chế hoạt động của hệ điều hành với 2 cơ chế chính: Định thời (Scheduling) và Quản lý bộ nhớ (Memory Management)

- Định thời: giải thuật định thời dựa trên Multilevel-queue được hiện thực với những ưu và khuyết điểm đã được trả lời ở những câu hỏi ở trên, nhóm nhận thấy việc cải thiện thêm một chút thành giải thuật định thời Multilevel Feedback Queue giúp tăng tính linh hoạt trong định thời CPU. Tuy nhiên, để hiện thực được những cơ chế giúp MLFQ trở nên hiệu quả hơn hẳn là không hề đơn giản, và còn phụ thuộc nhiều vào những yếu tố khác nhau.
- Quản lý bộ nhớ: mặc dù kết quả nhận được tương đối giống với lý thuyết đặt ra, tuy nhiên nhóm nhận thấy cơ chế quản lý bộ nhớ có thể được mở rộng hơn nhằm tăng thêm hiệu suất của bộ nhớ cũng như xử lý được những ngoại lệ có thể gặp phải. Quản lý bộ nhớ là một cơ chế rất phức tạp đòi hỏi hệ thống phải có khả năng xử lý được tất cả những ngoại lệ xảy ra.
- Trong quá trình thực hiện bài tập lớn, nhóm nhận thấy trong thực tế để xây dựng được hệ điều hành hoạt động theo đúng lý thuyết đặt ra là tương đối khó khăn, do cơ chế hoạt động của hệ điều hành là sự kết hợp có tính hệ thống của nhiều những thành phần khác nhau, mỗi thành phần lại có những cơ chế hiện thực và quản lý khác nhau. Chính vì vậy, nhóm đi nhận thấy việc hiện thực các cơ chế hoạt động của hệ điều hành cho ra kết quả mang tính tương đối và có thể linh hoạt thay đổi tùy theo nhu cầu và mục đích khác nhau.