VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

OPERATING SYSTEM
(CO2017)

Assignment

"Simple Operating System"

Instructor(s): Nguyễn Phương Duy, *CSE-HCMUT*
Hoàng Lê Hải Thanh, *CSE-HCMUT*

Students: Dương Gia Bảo - 2310207 *(Team L08_SysCallers)*
Nguyễn Đình Khôi - 2311681 *(Team L08_SysCallers)*
Bùi Nhật Quí - 2312864 *(Team L08_SysCallers)*
Trần Hoàng Uyên - 2313854 *(Team L08_SysCallers)*

HO CHI MINH CITY, MAY 2025

Mục lục

Danh mục hình vẽ

Danh mục bảng biểu

Danh sách thành viên và nhiệm vụ

STT	Họ và tên	MSSV	Nhiệm vụ	Hoàn thành
1	Dương Gia Bảo	2310207	- Hiện thực Scheduler - Trả lời câu hỏi	100%
2	Nguyễn Đình Khôi	2311681	- Hiện thực System Call - Trả lời câu hỏi	100%
3	Bùi Nhật Quí	2312864	- Hiện thực Memory Management - Trả lời câu hỏi	100%
4	Trần Hoàng Uyên	2313854	- Hiện thực Memory Management - Trả lời câu hỏi	100%

Bảng 1: Danh sách thành viên và nhiệm vụ

1 Mở đầu

1.1 Tổng quan bài tập lớn

Bài tập lớn "Simple Operating System" được thiết kế nhằm mục tiêu giúp sinh viên nắm bắt các khái niệm nền tảng về vận hành của một hệ điều hành thực thụ. Thông qua đề tài này, sinh viên được tiếp cận và thực hành với ba thành phần chính: bộ lập lịch tiến trình (scheduler), quản lý đồng bộ (synchronization) và quản lý bộ nhớ (memory management). Ngoài ra, đề tài còn yêu cầu xây dựng giao diện hệ điều hành thông qua cơ chế lời gọi hệ thống (system call).

Cụ thể, hệ điều hành mô phỏng sẽ quản lý hai loại tài nguyên ảo cơ bản là CPU và RAM:

- Bộ lập lịch (Scheduler): Áp dụng thuật toán Multi-Level Queue Scheduling (MLQ), trong đó các tiến trình được phân vào các hàng đợi theo mức ưu tiên. CPU sẽ chọn tiến trình từ hàng đợi có mức ưu tiên cao nhất để thực thi, với chính sách chia sẻ CPU dựa trên vòng lặp thời gian (time slice).
- Bộ nhớ ảo (Virtual Memory Management): Mỗi tiến trình có không gian địa chỉ riêng, tách biệt với tiến trình khác. Việc ánh xạ từ địa chỉ ảo sang địa chỉ vật lý được thực hiện theo cơ chế phân trang (paging), kết hợp với việc hoán đổi bộ nhớ (swapping) khi cần thiết.
- Giao diện lời gọi hệ thống (System Call Interface): Các tiến trình giao tiếp với hệ điều hành thông qua các lời gọi hệ thống như `listsyscall`, `memmap`, `killall`.

Bài tập lớn yêu cầu sinh viên xây dựng và tích hợp đầy đủ các module nêu trên để hình thành một hệ điều hành mô phỏng hoàn chỉnh, có khả năng thực thi nhiều tiến trình, quản lý bộ nhớ ảo và cung cấp các dịch vụ hệ thống cơ bản.

1.2 Cách chạy mô phỏng

Để thực hiện mô phỏng hệ điều hành, các bước tiến hành được trình bày như sau:

- Biên dịch hệ thống bằng lệnh `make all`, sau đó chạy mô phỏng bằng lệnh `./os [configure_file]`.
- Đối với một số testcase như `sched` và `os_1_singleCPU_mlq`, cần bật cờ `#define MM_FIXED_MEMSZ` trong file `include/os-cfg.h` để đảm bảo chương trình chạy đúng.
- Ngoài ra, có thể bật hoặc tắt dòng in ra thời gian trong file `timer.c` để dễ dàng theo dõi tiến trình thực thi trong output.

2 Scheduler

2.1 Giới thiệu về Scheduler

Trong bài tập lớn này, Scheduler là thành phần chịu trách nhiệm quản lý tiến trình và phân phối CPU. Mô hình sử dụng thuật toán "Multi-Level Queue" (MLQ), một phiên bản đơn giản hóa của hệ điều hành Linux. Mỗi tiến trình được gán một mức ưu tiên, và hệ thống duy trì một hàng đợi sẵn sàng (ready queue) cho mỗi mức ưu tiên từ 0 đến MAX_PRIO - 1.

Hệ thống chạy theo nguyên tắc round-robin trong mỗi hàng đợi và cấp CPU cho tiến trình theo mức ưu tiên từ cao đến thấp (ưu tiên cao có giá trị nhỏ hơn). Mỗi hàng đợi chỉ có một số lượng slot cố định để sử dụng CPU, sau đó nhường quyền lại cho hàng đợi khác. Điều này đảm bảo sự công bằng nhưng dẫn đến tình trạng "starvation" của các tiến trình ưu tiên thấp.

2.2 Hiện thực code

Việc hiện thực Scheduler trong bài toán được chia thành hai phần chính, trải dài qua các file `queue.c` và `sched.c`.

2.2.1 Hiện thực code trong file queue.c

Thao tác trên hàng đợi với hàm `enqueue()` và `dequeue()`. Hai hàm này dùng để thêm tiến trình vào và lấy tiến trình ra khỏi hàng đợi. Trong `MLQ`, tiến trình trong hàng đợi được sắp xếp dựa trên độ ưu tiên.

```
void enqueue(struct queue_t *q, struct pcb_t *proc)
{
    if (q == NULL || proc == NULL)
        return;
    if (q->size >= MAX_QUEUE_SIZE)
        return;
    q->proc[q->size] = proc;
    q->size++;
```

```
}

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

2.2.2 Hiện thực code trong file sched.c

Cấp phát tiến trình từ hàng đợi bằng hàm `get_mlq_proc()`. Hàm lấy tiến trình ra từ hàng đợi đúng theo quy định của `MLQ`, tức là đi từ mức ưu tiên cao đến thấp, mỗi hàng đợi có `slot` cố định.

```
struct pcb_t *get_mlq_proc(void)
{
    struct pcb_t *proc = NULL;
    pthread_mutex_lock(&queue_lock);
    int all_zero = 1;
    for (int i = 0; i < MAX_PRIO; i++)
```

```
    {
        if (slot[i] > 0)
        {
            all_zero = 0;
            break;
        }
    }

    if (all_zero)
    {
        for (int i = 0; i < MAX_PRIO; i++)
        {
            slot[i] = MAX_PRIO - i;
        }
    }

    for (int prio = 0; prio < MAX_PRIO; prio++)
    {
        if (!empty(&mlq_ready_queue[prio]))
        {
            if (slot[prio] > 0)
            {
                proc = dequeue(&mlq_ready_queue[prio]);
                slot[prio]--;
                break;
            }
        }
    }
    pthread_mutex_unlock(&queue_lock);
    return proc;
}
```

Thêm và đẩy tiến trình vào hệ thống: `add_proc()` được gọi khi một tiến trình mới được nạp từ đĩa vào hệ thống (từ `ld_routine()`). Trong khi đó, `put_proc()` được dùng khi tiến trình đã chạy xong một `time slice` và cần được đưa lại vào hàng đợi để chờ lần cấp CPU tiếp theo (gọi từ `cpu_routine()`).

```c
void put_proc(struct pcb_t *proc)
{
    proc->ready_queue = &ready_queue;
    proc->mlq_ready_queue = mlq_ready_queue;
    proc->running_list = &running_list;

    enqueue(&running_list, proc);

    return put_mlq_proc(proc);
}

void add_proc(struct pcb_t *proc)
{
    proc->ready_queue = &ready_queue;
    proc->mlq_ready_queue = mlq_ready_queue;
    proc->running_list = &running_list;

    enqueue(&running_list, proc);

    return add_mlq_proc(proc);
}
```

Sự khác biệt:

- `add_proc()` được dùng một lần duy nhất cho mỗi tiến trình, khi tiến trình đó mới được `load` vào hệ thống.
- `put_proc()` có thể được gọi nhiều lần sau mỗi `time slice`, đưa tiến trình quay lại hàng đợi nếu chưa hoàn thành.

2.3 Phân tích output

2.3.1 Trường hợp input sched

Cấu hình đầu vào:

- Comment out `#define MM_FIXED_MEMSZ` trong file `os-cfg.h` để chạy input:

```
4 2 3
```

```
0 p1s 1
1 p2s 0
2 p3s 0
```

- Các file tiến trình:

p1s	p2s	p3s
1 10	20 12	7 11
calc	calc	calc
calc	calc	calc
calc	calc	calc
calc	calc	calc
calc	calc	calc
calc	calc	calc
calc	calc	calc
calc	calc	calc
calc	calc	calc
calc	calc	calc
	calc	calc
	calc	

Bảng 2: Các tiến trình của sched

Output thực thi:

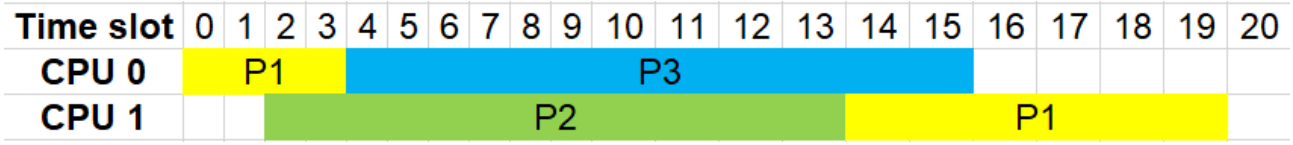
```
Time slot   0
ld_routine
  Loaded a process at input/proc/p1s, PID: 1 PRIO: 1
  CPU 0: Dispatched process  1
Time slot   1
  Loaded a process at input/proc/p2s, PID: 2 PRIO: 0
Time slot   2
  CPU 1: Dispatched process  2
  Loaded a process at input/proc/p3s, PID: 3 PRIO: 0
```

```
Time slot   3
Time slot   4
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  3
Time slot   5
Time slot   6
  CPU 1: Put process  2 to run queue
  CPU 1: Dispatched process  2
Time slot   7
Time slot   8
  CPU 0: Put process  3 to run queue
  CPU 0: Dispatched process  3
Time slot   9
Time slot  10
  CPU 1: Put process  2 to run queue
  CPU 1: Dispatched process  2
Time slot  11
Time slot  12
  CPU 0: Put process  3 to run queue
  CPU 0: Dispatched process  3
Time slot  13
Time slot  14
  CPU 1: Processed  2 has finished
  CPU 1: Dispatched process  1
Time slot  15
  CPU 0: Processed  3 has finished
  CPU 0 stopped
Time slot  16
Time slot  17
Time slot  18
  CPU 1: Put process  1 to run queue
  CPU 1: Dispatched process  1
Time slot  19
Time slot  20
  CPU 1: Processed  1 has finished
  CPU 1 stopped
```

Phân tích:

- `p1s` là tiến trình duy nhất tại thời điểm 0 nên được CPU 0 cấp phát ngay.
- `p2s` và `p3s` có ưu tiên cao hơn (prio = 0) nên được cấp CPU trước `p1s` trong những vòng sau.
- Cơ chế cấp phát đúng theo chính sách `MLQ`: các tiến trình cùng mức ưu tiên được chia CPU theo kiểu `round-robin`, mỗi hàng đợi sử dụng số `slot` nhất định rồi nhường lại.
- Các tiến trình được chạy xen kẽ, đảm bảo sự công bằng và hiệu quả theo thiết kế.
- Mỗi tiến trình sẽ kết thúc (`finish`) khi thực thi xong toàn bộ các dòng lệnh được khai báo trong file tương ứng tại thư mục `input/proc/` - ví dụ `input/proc/p1s`. Khi `pc == code->size`, tiến trình được giải phóng.

Sơ đồ Gantt:



Hình 1: Sơ đồ Gantt khi chạy input sched

Trong đó:

- Các ô chứa `P1`, `P2`, `P3` đại diện cho tiến trình được cấp CPU tại thời điểm tương ứng.
- Các ô trống thể hiện CPU đang chờ hoặc không có tiến trình sẵn sàng.

2.3.2 Trường hợp input sched_0

Cấu hình đầu vào:

- Comment out `#define MM_FIXED_MEMSZ` trong file `os-cfg.h` để chạy input:

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

```
4 s1 0
```

- Các file tiến trình:

s0	s1
12 15	20 7
calc	calc
calc	calc
calc	calc
calc	calc
calc	calc
calc	calc
calc	calc
calc	
calc	
calc	
calc	
calc	
calc	
calc	
calc	

Bảng 3: Các tiến trình của sched_0

Output thực thi:

```
Time slot   0
ld_routine
  Loaded a process at input/proc/s0, PID: 1 PRIO: 4
Time slot   1
  CPU 0: Dispatched process  1
Time slot   2
Time slot   3
  CPU 0: Put process  1 to run queue
```

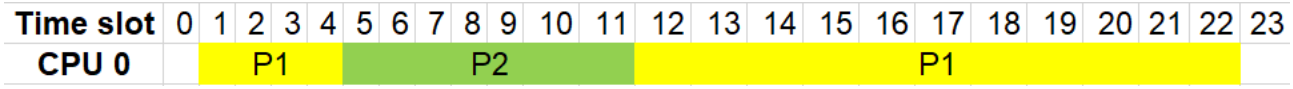
```
  CPU 0: Dispatched process  1
Time slot   4
  Loaded a process at input/proc/s1, PID: 2 PRIO: 0
Time slot   5
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  2
Time slot   6
Time slot   7
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  2
Time slot   8
Time slot   9
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  2
Time slot  10
Time slot  11
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  2
Time slot  12
  CPU 0: Processed  2 has finished
  CPU 0: Dispatched process  1
Time slot  13
Time slot  14
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
Time slot  15
Time slot  16
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
Time slot  17
Time slot  18
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
Time slot  19
Time slot  20
  CPU 0: Put process  1 to run queue
```

```
45   CPU 0: Dispatched process  1
46 Time slot  21
47 Time slot  22
48   CPU 0: Put process  1 to run queue
49   CPU 0: Dispatched process  1
50 Time slot  23
51   CPU 0: Processed  1 has finished
52   CPU 0 stopped
```

Phân tích:

- Tại time slot 0, tiến trình `s0` (PID 1, prio = 4) được nạp. Đây là tiến trình duy nhất nên được CPU 0 chạy đầu tiên.
- Tại slot 4, tiến trình `s1` (PID 2, prio = 0) được đưa vào hệ thống. Với ưu tiên cao hơn, `s1` lập tức giành quyền CPU từ `s0`.
- `P2 (s1)` được chạy liên tục từ slot 5 đến slot 12, trải qua nhiều lần enqueue và dispatch theo từng time slice. Cuối cùng, nó hoàn tất tại slot 12.
- Sau khi `P2` kết thúc, CPU 0 quay lại chạy `P1 (s0)` - vốn có ưu tiên thấp nên phải đợi đến khi hàng đợi `prio = 0` rỗng.
- Từ slot 12 đến slot 23, tiến trình `s0` tiếp tục được cấp CPU đều đặn theo từng time slic.
- Mỗi tiến trình kết thúc khi đã thực thi toàn bộ lệnh được định nghĩa trong file `input/proc/s0` và `input/proc/s1`.

Sơ đồ Gantt:



Hình 2: Sơ đồ Gantt khi chạy input sched_0

Trong đó:

- Các ô chứa `P1`, `P2` đại diện cho tiến trình được cấp CPU tại thời điểm tương ứng.
- Các ô trống thể hiện CPU đang chờ hoặc không có tiến trình sẵn sàng.

2.3.3 Trường hợp input sched_1

Cấu hình đầu vào:

- Comment out `#define MM_FIXED_MEMSZ` trong file `os-cfg.h` để chạy input:

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

- Các file tiến trình:

s0	s1	s2	s3
12 15	20 7	20 12	7 11
calc	calc	calc	calc
calc	calc	calc	calc
calc	calc	calc	calc
calc	calc	calc	calc
calc	calc	calc	calc
calc	calc	calc	calc
calc	calc	calc	calc
calc		calc	calc
calc		calc	calc
calc		calc	calc
calc		calc	calc
calc		calc	
calc			
calc			
calc			

Bảng 4: Các tiến trình của sched_1

Output thực thi:

```
Time slot   0
ld_routine
  Loaded a process at input/proc/s0, PID: 1 PRIO: 4
Time slot   1
  CPU 0: Dispatched process  1
Time slot   2
Time slot   3
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
Time slot   4
  Loaded a process at input/proc/s1, PID: 2 PRIO: 0
Time slot   5
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  2
Time slot   6
  Loaded a process at input/proc/s2, PID: 3 PRIO: 0
Time slot   7
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  3
  Loaded a process at input/proc/s3, PID: 4 PRIO: 0
Time slot   8
Time slot   9
  CPU 0: Put process  3 to run queue
  CPU 0: Dispatched process  4
Time slot  10
Time slot  11
  CPU 0: Put process  4 to run queue
  CPU 0: Dispatched process  4
Time slot  12
Time slot  13
  CPU 0: Put process  4 to run queue
  CPU 0: Dispatched process  4
Time slot  14
Time slot  15
  CPU 0: Put process  4 to run queue
```

```
  CPU 0: Dispatched process  4
Time slot  16
Time slot  17
  CPU 0: Put process  4 to run queue
  CPU 0: Dispatched process  4
Time slot  18
Time slot  19
  CPU 0: Put process  4 to run queue
  CPU 0: Dispatched process  4
Time slot  20
  CPU 0: Processed  4 has finished
  CPU 0: Dispatched process  2
Time slot  21
Time slot  22
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  3
Time slot  23
Time slot  24
  CPU 0: Put process  3 to run queue
  CPU 0: Dispatched process  2
Time slot  25
Time slot  26
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  3
Time slot  27
Time slot  28
  CPU 0: Put process  3 to run queue
  CPU 0: Dispatched process  2
Time slot  29
  CPU 0: Processed  2 has finished
  CPU 0: Dispatched process  3
Time slot  30
Time slot  31
  CPU 0: Put process  3 to run queue
  CPU 0: Dispatched process  3
Time slot  32
```

```
Time slot   33
  CPU 0: Put process  3 to run queue
  CPU 0: Dispatched process  3
Time slot   34
Time slot   35
  CPU 0: Processed  3 has finished
  CPU 0: Dispatched process  1
Time slot   36
Time slot   37
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
Time slot   38
Time slot   39
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
Time slot   40
Time slot   41
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
Time slot   42
Time slot   43
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
Time slot   44
Time slot   45
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
Time slot   46
  CPU 0: Processed  1 has finished
  CPU 0 stopped
```

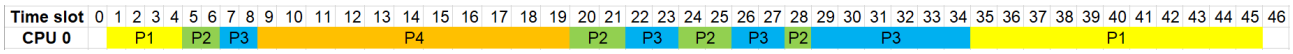
Phân tích:

- Từ slot 0 đến slot 4: chỉ có `s0` tồn tại trong hệ thống nên được cấp CPU liên tục.
- Tại slot 4, tiến trình `s1` (prio = 0) được nạp, có ưu tiên cao hơn - CPU lập

tức chuyển sang chạy `s1` ở slot 5.

- Sau đó, các tiến trình `s2`, `s3` tiếp tục được nạp vào slot 6–7, cũng thuộc nhóm ưu tiên cao nhất (0).
- Từ slot 7 đến 24: hệ thống bắt đầu xoay vòng (round-robin) giữa các tiến trình cùng mức ưu tiên 0 — cụ thể là các tiến trình `s1`, `s2`, `s3` lần lượt được cấp CPU theo thứ tự.
- Tiến trình `s1` (PID 2) kết thúc tại slot 29. Các tiến trình còn lại tiếp tục chạy xoay vòng như trước.
- Tiến trình `s3` (PID 4) và `s2` (PID 3) lần lượt kết thúc tại slot 20 và 35.
- Cuối cùng, khi hàng đợi ưu tiên 0 rỗng, tiến trình `s0` (PID 1, prio = 4) mới tiếp tục được cấp CPU.
- Từ slot 35 đến 46: `s0` được cấp CPU liên tục theo từng time slice và kết thúc tại slot 46.

Sơ đồ Gantt:



Hình 3: Sơ đồ Gantt khi chạy input sched_1

Trong đó:

- Các ô chứa `P1`, `P2`, `P3` và `P4` đại diện cho tiến trình được cấp CPU tại thời điểm tương ứng.
- Các ô trống thể hiện CPU đang chờ hoặc không có tiến trình sẵn sàng.

2.4 Trả lời câu hỏi

Question: What are the advantages of using the scheduling algorithm described in this assignment compared to other scheduling algorithms you have learned?

Trả lời

Ưu điểm của thuật toán MLQ so với các thuật toán khác:

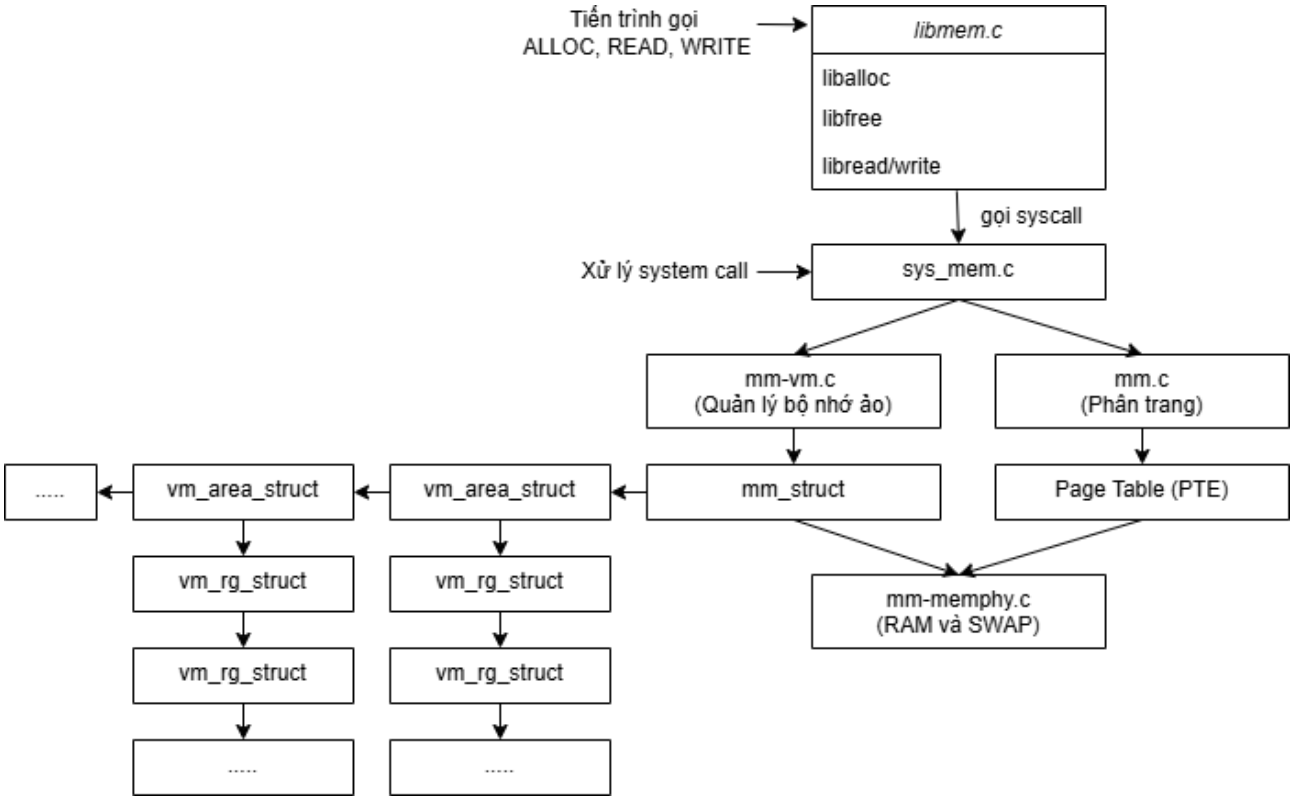
- Mỗi mức ưu tiên được gán cho một hàng đợi riêng biệt, giúp dễ dàng quản lý và chọn tiến trình có độ ưu tiên cao hơn để thực thi trước.
- Hệ thống có thể vận hành trên nhiều CPU một cách hiệu quả, mỗi CPU có thể chọn tiến trình từ hàng đợi theo nguyên tắc vòng tròn (round-robin), tối ưu hiệu suất xử lý.
- Các tiến trình trong cùng một hàng đợi được xử lý theo vòng tròn, giúp đảm bảo tính công bằng và tránh tình trạng "starvation" trong một mức ưu tiên.

Tuy nhiên, do không có cơ chế phản hồi (feedback) như Multilevel Feedback Queue, thuật toán trong bài không cho phép tiến trình thay đổi mức ưu tiên trong suốt thời gian thực thi, dễ dẫn đến "starvation" ở tiến trình có mức ưu tiên thấp.

3 Memory Management

3.1 Giới thiệu về Memory Management

Trong một hệ điều hành, việc quản lý bộ nhớ là một trong những nhiệm vụ cốt lõi và phức tạp nhất. Hệ thống mô phỏng này triển khai một cơ chế quản lý bộ nhớ gồm nhiều phân lớp, mô phỏng lại cách thức hiện đại mà các hệ điều hành sử dụng để bảo vệ tiến trình, tối ưu tài nguyên, và cho phép hoán trang khi cần thiết. Dưới đây là sơ đồ tổng quan về tổ chức các lớp của bộ nhớ trong hệ thống này:



Hình 4: Sơ đồ tổ chức các lớp bộ nhớ

Các phân lớp quản lý bộ nhớ từ trên xuống có vai trò như sau:

- Thư viện người dùng (`libmem.c`): Cấp phát, ghi/đọc (alloc, write/read) bộ nhớ cho tiến trình. Gọi syscall để thực thi.
- Xử lý System Call (`sys_mem.c`): Nhận và xử lý syscall từ tiến trình, gọi vào các phân lớp `mm-vm` và `mm` để thực thi quản lý bộ nhớ.
- Bộ nhớ ảo (`mm-vm.c`): Quản lý vùng nhớ của tiến trình.

- Phân trang (`mm.c`): Thực hiện ánh xạ page – frame qua bảng trang, cấp phát khung RAM, hoán trang.
- Bộ nhớ vật lý (`mm-memphy.c`): thực hiện thao tác thô trên bộ nhớ vật lý, mô phỏng RAM, SWAP, cấp phát frame, đọc/ghi các ô nhớ thực.

3.2 Hiện thực code

3.2.1 Hiện thực code trong file libmem.c

Thư viện quản lý bộ nhớ cho người dùng là thư viện cung cấp các hàm quản lý bộ nhớ cho chương trình, chịu trách nhiệm tiếp nhận các yêu cầu cấp phát, giải phóng, đọc và ghi dữ liệu trong bộ nhớ, sau đó chuyển chúng thành các thao tác tương ứng ở kernel mode. Khi cần mở rộng không gian bộ nhớ hoặc truy xuất dữ liệu ở mức vật lý, nó sẽ gọi system call do chương trình không thể trực tiếp truy cập tài nguyên phần cứng.

Trong chế độ user mode, chương trình người dùng thực hiện các thao tác cấp phát, đọc hoặc ghi bộ nhớ thông qua các hàm do thư viện libmem cung cấp. Thư viện này đóng vai trò trung gian giữa không gian người dùng và kernel: khi cần mở rộng vùng nhớ hoặc truy cập dữ liệu trong RAM, libmem sẽ thực hiện các system call để chuyển quyền điều khiển xuống kernel mode.

Trong kernel mode, hệ điều hành tiếp nhận các yêu cầu từ libmem thông qua syscall handler, sau đó phân phối đến các module quản lý bộ nhớ ảo và phân trang để thực hiện các tác vụ cụ thể như cấp phát khung trang, hoán đổi trang, hoặc truy cập bộ nhớ vật lý.

Mặc dù libmem thuộc về user mode, nó vẫn có thể truy cập một số cấu trúc được kernel cấp phát riêng cho tiến trình, như bảng trang (page table) hay thông tin vùng nhớ ảo (VM area), nhằm phục vụ xác định trạng thái bộ nhớ trước khi thực hiện system call. Tuy nhiên, mọi thao tác can thiệp đến tài nguyên phần cứng (frame, swap, RAM) vẫn được ủy thác hoàn toàn cho kernel mode thông qua cơ chế syscall.

Hàm `__alloc` có nhiệm vụ cấp phát một vùng nhớ ảo cho một tiến trình (process) trong một vùng quản lý bộ nhớ ảo cụ thể (VM Area) bằng cách gọi `get_free_vmrg_area` để tìm vùng trống đủ lớn trong VM Area, và lưu thông tin vùng nhớ đó vào bảng ký

hiệu (symrgtbl) của tiến trình.

```
int __alloc(struct pcb_t *caller, int vmaid, int rgid, int size, int *
    alloc_addr)
{
    pthread_mutex_lock(&mmvm_lock);
    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;
    }
    else
    {
        struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm,
    vmaid);

        int inc_sz = PAGING_PAGE_ALIGNSZ(size);

        int old_sbrk = cur_vma->sbrk;

        struct sc_regs regs;
        regs.a1 = SYSMEM_INC_OP;
        regs.a2 = vmaid;
        regs.a3 = inc_sz;

        int ret = syscall(caller, 17, &regs);
        if (ret != 0)
        {
            pthread_mutex_unlock(&mmvm_lock);
            return ret;
        }
        cur_vma->sbrk = old_sbrk + inc_sz;
        *alloc_addr = old_sbrk;
        if (get_free_vmrg_area(caller, vmaid, size, &rgnode) == 0)
```

```
        {
            caller->mm->symrgtbl[rgid].rg_start = old_sbrk;
            caller->mm->symrgtbl[rgid].rg_end = old_sbrk + inc_sz;
        }
    }
    pthread_mutex_unlock(&mmvm_lock);
    printf("===== PHYSICAL MEMORY AFTER ALLOCATION =====\n");
#ifdef IODUMP
    printf("PID=%d - Region=%d - Address=%08lx - Size=%d byte\n",
           caller->pid, rgid, (unsigned long)*alloc_addr, size);
#ifdef PAGETBL_DUMP
    print_pgtbl(caller, 0, -1);
#endif
#endif
    return 0;
}
```

Hàm `__free` đánh dấu region memory (tương ứng với rgid) trong area memory (tương ứng với vmaid) là trống và đưa vào freerg_list, không thu hồi frame ngay.

```
int __free(struct pcb_t *caller, int vmaid, int rgid)
{
    if (rgid < 0 || rgid > PAGING_MAX_SYMTBL_SZ)
        return -1;

    struct vm_rg_struct *freerg = (struct vm_rg_struct *)malloc(sizeof
   (struct vm_rg_struct));
    if (freerg == NULL)
        return -1;

    pthread_mutex_lock(&mmvm_lock);
    freerg->rg_start = caller->mm->symrgtbl[rgid].rg_start;
    freerg->rg_end = caller->mm->symrgtbl[rgid].rg_end;
    freerg->rg_next = NULL;

    enlist_vm_freerg_list(caller->mm, freerg);

```

```
    caller->mm->symrgtbl[rgid].rg_start = 0;
    caller->mm->symrgtbl[rgid].rg_end = 0;
    pthread_mutex_unlock(&mmvm_lock);
    printf("===== PHYSICAL MEMORY AFTER DEALLOCATION =====\n");
#ifdef IODUMP
    printf("PID=%d - Region=%d\n",
           caller->pid, rgid);
#ifdef PAGETBL_DUMP
    print_pgtbl(caller, 0, -1); // print max TBL
#endif
#endif
    return 0;
}
```

Hàm pg_getpage đảm bảo rằng một trang (page) có số hiệu pgn của tiến trình được yêu cầu đang có trong RAM. Nếu trang đang nằm trên bộ nhớ phụ (swap), gọi find_victim_page để chọn một trang trong RAM cần đẩy ra, swap-out trang nạn nhân ra vùng swap, swap-in trang yêu cầu vào RAM và cập nhật lại bảng trang (page table).

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

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

        find_victim_page(caller->mm, &vicpgn);
        vicpte = mm->pgd[vicpgn];
        vicfpn = PAGING_PTE_FPN(vicpte);

        MEMPHY_get_freefp(caller->active_mswp, &swpfpn);

        struct sc_regs regs_out;
```

```
        regs_out.a1 = SYSMEM_SWP_OP;
        regs_out.a2 = vicfpn;
        regs_out.a3 = swpfpn;
        int status = syscall(caller, 17, &regs_out);
        if (status != 0)
        {
            pthread_mutex_unlock(&mmvm_lock);
            return status;
        }

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

        tgtfpn = vicfpn;

        int tgt_swpoff = PAGING_PTE_SWP(pte);

        struct sc_regs regs_in;
        regs_in.a1 = SYSMEM_SWP_OP;
        regs_in.a2 = tgt_swpoff;
        regs_in.a3 = tgtfpn;
        status = syscall(caller, 17, &regs_in);
        if (status != 0)
        {
            pthread_mutex_unlock(&mmvm_lock);
            return status;
        }

        pte_set_fpn(&mm->pgd[pgn], tgtfpn);
        PAGING_PTE_SET_PRESENT(mm->pgd[pgn]);

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

    *fpn = PAGING_FPN(mm->pgd[pgn]);
    pthread_mutex_unlock(&mmvm_lock);
```

```
    return 0;
}
```

Hàm pg_getval đọc dữ liệu tại địa chỉ ảo addr. Gọi pg_getpage để đảm bảo trang có trong RAM, sau đó dùng syscall để đọc từ địa chỉ vật lý.

```
int pg_getval(struct mm_struct *mm, int addr, BYTE *data, struct pcb_t
    *caller)
{
    int pgn = PAGING_PGN(addr);
    int off = PAGING_OFFST(addr);
    int fpn;

    /* Get the page to MEMRAM, swap from MEMSWAP if needed */
    if (pg_getpage(mm, pgn, &fpn, caller) != 0)
        return -1; /* invalid page access */

    /* TODO
     *  MEMPHY_read(caller->mram, phyaddr, data);
     *  MEMPHY READ
     *  SYSCALL 17 sys_memmap with SYSMEM_IO_READ
     */
    int phyaddr = fpn * PAGING_PAGESZ + off;
    return MEMPHY_read(caller->mram, phyaddr, data);
}
```

Hàm pg_setval ghi dữ liệu vào địa chỉ ảo addr. Đảm bảo trang đã tồn tại rồi gọi syscall ghi dữ liệu vào địa chỉ vật lý.

```
int pg_setval(struct mm_struct *mm, int addr, BYTE value, struct pcb_t
    *caller)
{
    int pgn = PAGING_PGN(addr);
    int off = PAGING_OFFST(addr);
    int fpn;

    /* Get the page to MEMRAM, swap from MEMSWAP if needed */
    if (pg_getpage(mm, pgn, &fpn, caller) != 0)
```

```
        return -1; /* invalid page access */

    /* TODO
     *  MEMPHY_write(caller->mram, phyaddr, value);
     *  MEMPHY WRITE
     *  SYSCALL 17 sys_memmap with SYSMEM_IO_WRITE
     */
    int phyaddr = fpn * PAGING_PAGESZ + off;
    return MEMPHY_write(caller->mram, phyaddr, value);
}
```

Giao diện đọc cho chương trình, dùng `__read` để lấy giá trị từ vùng nhớ.

```
int libread(
    struct pcb_t *proc, // Process executing the instruction
    uint32_t source,    // Index of source register
    uint32_t offset,    // Source address = [source] + [offset]
    uint32_t *destination)
{
    BYTE data;
    int val = __read(proc, 0, source, offset, &data);

    /* TODO update result of reading action*/
    // destination
    *destination = data;
#ifdef IODUMP
    printf("===== PHYSICAL MEMORY AFTER READING =====\n");
    printf("read region=%d offset=%d value=%d\n", source, offset, data
   );
#ifdef PAGETBL_DUMP
    print_pgtbl(proc, 0, -1); // print max TBL
#endif
    MEMPHY_dump(proc->mram);
#endif

    return val;
}
```

Hàm `find_victim_page` tìm một trang nạn nhân (victim page) để thay thế khi cần cấp phát khung trang mới (frame) trong bộ nhớ RAM, theo chiến lược FIFO (First-In-First-Out). Loại bỏ trang đó khỏi hàng đợi FIFO và trả về chỉ số trang (pgn) thông qua `*retpgn`. Nếu không tìm được trang nào trả về -1.

```
int find_victim_page(struct mm_struct *mm, int *retpgn)
{
    if (mm->fifo_pgn != NULL)
    {
        struct pgn_t *pPage = NULL;
        struct pgn_t *lPage = mm->fifo_pgn;
        while (lPage->pg_next != NULL)
        {
            pPage = lPage;
            lPage = lPage->pg_next;
        }
        *retpgn = lPage->pgn;
        if (pPage == NULL)
        {
            mm->fifo_pgn = lPage->pg_next;
        }
        else
        {
            pPage->pg_next = lPage->pg_next;
        }

        free(lPage);

        return 0;
    }
    *retpgn = -1;
    return -1;
}
```

Hàm `get_free_vmrg_area` duyệt danh sách vùng trống để tìm vùng đủ lớn để cấp phát. Nếu có thì cập nhật lại freerg_list.

```
int get_free_vmrg_area(struct pcb_t *caller, int vmaid, int size,
   struct vm_rg_struct *newrg)
{
    struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid)
   ;
    if (cur_vma == NULL)
        return -1;

    struct vm_rg_struct *rgit = cur_vma->vm_freerg_list;
    if (rgit == NULL)
        return -1;

    /* Probe unintialized newrg */
    newrg->rg_start = newrg->rg_end = -1;

    /* TODO Traverse on list of free vm region to find a fit space */
    while (rgit != NULL)
    {
        if (rgit->rg_start + size <= rgit->rg_end)
        {
            newrg->rg_start = rgit->rg_start;
            newrg->rg_end = rgit->rg_start + size;

            if (rgit->rg_start + size < rgit->rg_end)
            {
                rgit->rg_start = rgit->rg_start + size;
            }
            else
            {

                struct vm_rg_struct *nextrg = rgit->rg_next;

                if (nextrg != NULL)
                {
                    rgit->rg_start = nextrg->rg_start;
                    rgit->rg_end = nextrg->rg_end;
```

```
35                      rgit->rg_next = nextrg->rg_next;
36                      free(nextrg);
37                  }
38                  else
39                  {
40                      rgit->rg_start = rgit->rg_end;
41                      rgit->rg_next = NULL;
42                  }
43              }
44              break;
45          }
46          else
47          {
48              rgit = rgit->rg_next;
49          }
50      }
51
52      if (newrg->rg_start == -1)
53      {
54          return -1;
55      }
56
57      return 0;
58  }
```

3.2.2 Xử lý lời gọi hệ thống Syscall

3.2.2.1 Vai trò

Tiếp nhận các yêu cầu quản lý bộ nhớ từ libmem và thực thi chúng với đặc quyền ở kernel mode. Nó định nghĩa các mã lệnh syscall liên quan đến bộ nhớ và chứa hàm xử lý tương ứng. Nói cách khác, syscall đóng vai trò bộ phân phối: nhận trap từ user, phân loại loại thao tác bộ nhớ, rồi gọi các hàm thích hợp ở các phân lớp dưới (bộ quản lý bộ nhớ ảo, vật lý hoặc phân trang) để thực hiện, đảm bảo quá trình chuyển đổi giữa user và kernel diễn ra an toàn và đúng chức năng.

3.2.2.2 Chức năng chính

Hàm `__sys_memmap(struct pcb_t *caller, struct sc_regs* regs)` là trình xử lý chính cho system call opcode 17, chuyên thực hiện các thao tác quản lý bộ nhớ ở mức kernel khi libmem gửi yêu cầu xuống qua syscall. Trong đó, hai tham số đầu vào là caller đóng vai trò là PCB của tiến trình gọi syscall và regs là tập thanh ghi mô phỏng chứa thông tin syscall (opcode + tham số). regs->a1 sẽ chứa tao tác yêu cầu từ người dùng, cụ thể:

- SYSMEM_INC_OP: gọi hàm inc_vma_limit(caller, regs->a2, regs->a3) để tăng kích thước vùng nhớ ảo, regs->a2 là vm_id của vùng nhớ, regs->a3 là số byte cần tăng.
- SYSMEM_SWP_OP: gọi hàm __mm_swap_page(caller, regs->a2, regs->a3) để thực hiện sao chép dữ liệu giữa trang RAM và SWAP.
- SYSMEM_IO_READ: gọi hàm MEMPHY_read(caller->mram, regs->a2, &value) để đọc byte tại địa chỉ regs->a2 và lưu kết quả vào biến value.
- SYSMEM_IO_WRITE: gọi hàm MEMPHY_write(caller->mram, regs->a2, regs->a3) để ghi giá trị regs->a3 vào địa chỉ regs->a2.

Do hàm `syscall` luôn trả về 0, nên để tiện theo dõi các trường hợp như "OOM: vm_map_ram out of memory", ta sẽ điều chỉnh trong hàm `__sys_memmap` khi xảy ra trường hợp SYSMEM_INC_OP thì sẽ trả về giá trị của hàm `inc_vma_limit`.

```
case SYSMEM_INC_OP:
    int ret = inc_vma_limit(caller, regs->a2, regs->a3);
    return ret;
```

Phân lớp này nằm giữa user và kernel. Ở phía trên, nó nhận các lệnh từ libmem. Ở phía dưới, nó gọi các hàm của hệ thống quản lý bộ nhớ `(mm-vm, mm, mm-memphy)` để thực hiện tác vụ. Mỗi chức năng của syscall tương ứng với việc kích hoạt một chức năng ở tầng dưới, cụ thể là SYSMEM_INC_OP kích hoạt tăng vùng nhớ ảo, SYSMEM_SWP_OP kích hoạt cơ chế swap trang, SYSMEM_IO_READ và SYSMEM_IO_WRITE kích hoạt truy xuất bộ nhớ vật lý. Sau khi tầng dưới hoàn thành, syscall trả kết quả về cho libmem. Tầng này đảm bảo tính đóng gói và an toàn: chương

trình người dùng không trực tiếp tác động đến cấu trúc kernel mà phải thông qua syscall, kernel kiểm soát và chuyển tiếp phù hợp.

3.2.3 Hiện thực code trong file mm-vm.c

Quản lý bộ nhớ ảo ở cấp độ tiến trình, đặc biệt là các vùng địa chỉ ảo (virtual memory areas) trong không gian của tiến trình. Nó có chức năng tương tương tự các đoạn (segments) của tiến trình (code, data, heap, stack,...). Mỗi vùng được biểu diễn bởi một cấu trúc `vm_area_struct` chứa thông tin như số định danh của vùng `vm_id`, địa chỉ bắt đầu `vm_start`, địa chỉ kết thúc `vm_end`, và danh sách các khoảng trống trong vùng `vm_freerg_list` (danh sách các đoạn trống có thể cấp phát `vm_rg_struct`), `mm-vm` chịu trách nhiệm duy trì ranh giới vùng nhớ ảo của tiến trình và đảm bảo việc mở rộng hay thu hẹp vùng được thực hiện hợp lệ, không chồng lấn. Ngoài ra, `mm-vm` còn phối hợp xử lý page fault: quyết định trang nào swap out, trang nào swap in.

Hàm tạo một `vm_rg_struct` mới tại vị trí sbrk hiện tại của tiến trình, có độ dài alignedsz, cập nhật sbrk sau khi cấp phát, và trả về con trỏ đến vùng vừa tạo.

```
1  struct vm_rg_struct *get_vm_area_node_at_brk(struct pcb_t *caller, int
       vmaid, int size, int alignedsz)
2  {
3      struct vm_rg_struct *newrg;
4      /* TODO retrive current vma to obtain newrg, current comment out
      due to compiler redundant warning*/
5      struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid)
      ;
6      if (cur_vma == NULL)
7      {
8          return NULL;
9      }
10     newrg = malloc(sizeof(struct vm_rg_struct));
11     if (newrg == NULL)
12     {
13         return NULL;
14     }
15
```

```
    /* TODO: update the newrg boundary*/
    newrg->rg_start = cur_vma->sbrk;
    newrg->rg_end = newrg->rg_start + alignedsz;
    newrg->rg_next = NULL;
    cur_vma->sbrk = newrg->rg_end;
    return newrg;
}
```

Hàm `validate_overlap_vm_area` kiểm tra xem vùng địa chỉ ảo [vmastart, vmaend] có trùng với bất kỳ vùng đã được cấp phát nào trong vùng vmaid hay không.

```
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 != NULL)
    {
        if (vma->vm_id == vmaid)
        {
            vma = vma->vm_next;
            continue;
        }
        if (!(vma->vm_end <= vmastart || vma->vm_start >= vmaend))
        {
            return -1;
        }
        vma = vma->vm_next;
    }
    return 0;
}
```

Hàm `inc_vma_limit` dùng để tăng giới hạn vùng nhớ vmaid. Hàm này thực hiện việc tạo một vùng trống mới tại cuối vùng hiện tại, điều chỉnh vm_end và sbrk của vùng.

```
int inc_vma_limit(struct pcb_t *caller, int vmaid, int inc_sz)
```

```
{
    struct vm_rg_struct *newrg = malloc(sizeof(struct vm_rg_struct));
    int inc_amt = PAGING_PAGE_ALIGNSZ(inc_sz);
    int incnumpage = inc_amt / PAGING_PAGESZ;
    struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid)
   ;
    int old_sbrk = cur_vma->sbrk;

    struct vm_rg_struct *area = get_vm_area_node_at_brk(caller, vmaid,
    inc_sz, inc_amt);
    if (!area || !cur_vma)
    {
        cur_vma->sbrk = old_sbrk;
        return -1;
    }
    int old_end = cur_vma->vm_end;

    /*Validate overlap of obtained region */
    if (validate_overlap_vm_area(caller, vmaid, area->rg_start, area->
   rg_end) < 0)
    {
        cur_vma->sbrk = old_sbrk;
        free(area);
        return -1;
    }

    if (vm_map_ram(caller, area->rg_start, area->rg_end, old_end,
   incnumpage, newrg) < 0)
    {
        cur_vma->sbrk = old_sbrk;
        free(area);
        return -1;
    }
    cur_vma->vm_end = area->rg_end;
    enlist_vm_rg_node(&cur_vma->vm_freerg_list, newrg);
    return 0;
```

```
}
```

3.2.4 Hiện thực code trong file mm.c

Hiện thực cơ chế phân trang của hệ thống, nó quản lý bảng trang page table của mỗi tiến trình, gán các số khung trang vật lý cho các số trang, và thực thi các thuật toán thay thế trang khi cần. Tầng này biết rõ trang nào đang ở trong RAM, trang nào đã bị swap out vào SWAP, và chứa các hàm để cập nhật trạng thái này, ngoài ra cũng cung cấp các tiện ích để yêu cầu frame mới từ bộ nhớ vật lý và để copy dữ liệu của trang qua lại giữa RAM và SWAP.

Lớp này làm việc hoàn toàn trong kernel. Khi `mm-vm` yêu cầu ánh xạ trang mới, `mm` thực hiện cấp phát frame và cập nhật PTE. Khi `mm-vm` yêu cầu swap trang, `mm` thực hiện sao chép và điều chỉnh PTE. Ngoài ra, `mm` còn tương tác với `mm-memphy`: mọi thao tác lấy frame trống, trả frame, đọc ghi dữ liệu hay sao chép trang đều dựa trên chức năng của `mm-memphy`. Tầng mm chuyển các yêu cầu này thành các lệnh cụ thể trên `mm-memphy`. Ngoài ra, mm cũng cung cấp thông tin cho syscall để hoàn thành việc đọc/ghi: khi syscall SYSTEM_IO_READ/WRITE được gọi, `mm` (qua bảng trang) đã biết địa chỉ vật lý hợp lệ để `mm-memphy` thao tác.

Hàm `vmap_page_range` ánh xạ một dải các khung trang vật lý (physical frames) vào không gian địa chỉ ảo (virtual address space) của một tiến trình bắt đầu từ addr. Cập nhật vùng ánh xạ trả về qua ret_rg.

```
int vmap_page_range(struct pcb_t *caller,
                    int addr,
                    int pgnum,
                    struct framephy_struct *frames,
                    struct vm_rg_struct *ret_rg)
{
    int pgit = 0;
    int pgn = PAGING_PGN(addr);
    struct framephy_struct *fpit = frames;
    // TODO: update the rg_end and rg_start of ret_rg
    ret_rg->rg_start = addr;
```

```
    ret_rg->rg_end = addr + 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 = 0; pgit < pgnum; pgit++)
    {
        if (fpit == NULL)
        {
            return -1;
        }
        init_pte(&caller->mm->pgd[pgn + pgit], 1, fpit->fpn, 0, 0, 0,
   0);
        enlist_pgn_node(&caller->mm->fifo_pgn, pgn + pgit);
        fpit = fpit->fp_next;
    }
    return 0;
}
```

Hàm `alloc_pages_range` cấp phát một dải khung trang vật lý (frame) trong RAM cho tiến trình. Yêu cầu số lượng trang là req_pgnum và trả về danh sách các frame đã cấp phát qua frm_lst. Trả lỗi và thu hồi các frame vừa cấp nếu không đủ số frame cần thiết.

```
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;
    // TODO: allocate the page
    for (pgit = 0; pgit < req_pgnum; pgit++)
    {
        // TODO: allocate the page
        if (MEMPHY_get_freefp(caller->mram, &fpn) == 0)
        {
            struct framephy_struct *newfp = malloc(sizeof(struct
```

```
    framephy_struct));
12              newfp->fpn = fpn;
13              newfp->owner = caller->mm;
14              newfp->fp_next = newfp_str;
15              newfp_str = newfp;
16              *frm_lst = newfp_str;
17          }
18          else
19          {
20              // TODO: ERROR CODE of obtaining somes but not enough
    frames
21              struct framephy_struct *fpit = newfp_str;
22              while (fpit)
23              {
24                  MEMPHY_put_freefp(caller->mram, fpit->fpn);
25                  struct framephy_struct *next = fpit->fp_next;
26                  free(fpit);
27                  fpit = next;
28              }
29              return -3000;
30          }
31      }
32      return 0;
33  }
```

Hàm `init_mm` khởi tạo không gian bộ nhớ cho một tiến trình: bảng trang, danh sách vùng nhớ, danh sách trang đang dùng.

```
1  int init_mm(struct mm_struct *mm, struct pcb_t *caller)
2  {
3      struct vm_area_struct *vma0 = malloc(sizeof(struct vm_area_struct)
       );
4
5      mm->pgd = malloc(PAGING_MAX_PGN * sizeof(uint32_t));
6
7      /* By default the owner comes with at least one vma */
8      vma0->vm_id = 0;
```

```
    vma0->vm_start = 0;
    vma0->vm_end = vma0->vm_start;
    vma0->sbrk = vma0->vm_start;
    struct vm_rg_struct *first_rg = init_vm_rg(vma0->vm_start, vma0->
   vm_end);
    enlist_vm_rg_node(&vma0->vm_freerg_list, first_rg);

    /* TODO update VMA0 next */
    vma0->vm_next = NULL;

    /* Point vma owner backward */
    vma0->vm_mm = mm;

    /* TODO: update mmap */
    mm->mmap = vma0;

    return 0;
}
```

3.2.5 Hiện thực code trong file mm-memphy.c

Mô phỏng phần cứng bộ nhớ vật lý trong hệ thống, nó bao gồm cả bộ nhớ RAM và SWAP. Nhiệm vụ chính là quản lý các khung trang vật lý (physical frames), đánh dấu cái nào còn trống, cái nào đã được cấp và thực hiện đọc/ghi dữ liệu nhị phân tại địa chỉ vật lý cụ thể.

Phía trên, `mm-memphy` nhận lệnh từ `mm`, `mm-memphy` không biết về tiến trình hay trang (do các lớp trên xử lý), nó chỉ cung cấp các khung trống và khả năng lưu trữ dữ liệu. Trong hệ điều hành đơn giản này, mỗi tiến trình có thể có con trỏ đến một cấu trúc `mm-memphy` cho RAM chính (proc->mram) và một cấu trúc cho vùng SWAP đang dùng (proc->active_mswp). Điều quan trọng là lớp này phải đảm bảo tính toàn vẹn dữ liệu: khi `mm` ghi dữ liệu vào một địa chỉ, lần sau đọc lại phải đúng dữ liệu đó. Ngoài ra, nó còn hỗ trợ cơ chế hoán đổi: RAM và SWAP kết hợp với nhau để chuyển dữ liệu trang. Tóm lại, `mm-memphy` tương tác với `mm` như một bộ nhớ vật lý đơn giản: cung cấp frame khi được yêu cầu và đọc/ghi nội dung tại địa chỉ cho trước.

Hàm `MEMPHY_dump` in toàn bộ nội dung bộ nhớ vật lý ra màn hình.

```
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;
}
```

3.3 Phân tích output

Do đầu ra của các tập tin kiểm thử khá dài, để đảm bảo tính súc tích cho báo cáo, nhóm chỉ trình bày chi tiết output của ba testcase tiêu biểu: os_0_mlq_paging, os_1_mlq_paging_small_1K và os_1_singleCPU_mlq. Các output còn lại sẽ được nhóm chuẩn bị sẵn, sẵn sàng trình bày theo yêu cầu của giảng viên trong buổi thuyết trình hoặc có thể kiểm tra thông qua mã nguồn (thư mục `team_output`) nhóm đã nộp.

3.3.1 Trường hợp input os_0_mlq_paging

Cấu hình đầu vào:

- Configuration file `os_0_mlq_paging` để khởi động Simple Operating System:

```
6 2 4
```

```
1048576 16777216 0 0 0
0 p0s 0
2 p1s 15
4 p1s 0
6 p1s 0
```

- Các file tiến trình:

p0s	p1s
1 14	1 10
calc	calc
alloc 300 0	calc
alloc 300 4	calc
free 0	calc
alloc 100 1	calc
write 100 1 20	calc
read 1 20 20	calc
write 102 2 20	calc
read 2 20 20	calc
write 103 3 20	calc
read 3 20 20	
calc	
free 4	
calc	

Bảng 5: Các tiến trình của os_0_mlq_paging

Output thực thi:

```
ld_routine
  Loaded a process at input/proc/p0s, PID: 1 PRIO: 0
  CPU 0: Dispatched process  1
===== 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
================================================================
  Loaded a process at input/proc/p1s, PID: 2 PRIO: 15
  CPU 1: Dispatched process  2
===== 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
================================================================
===== PHYSICAL MEMORY AFTER DEALLOCATION =====
PID=1 - Region=0
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 AFTER ALLOCATION =====
PID=1 - Region=1 - Address=00000000 - 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
================================================================
  Loaded a process at input/proc/p1s, PID: 3 PRIO: 0
===== PHYSICAL MEMORY AFTER WRITING =====
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 00000114: 100
===== PHYSICAL MEMORY END-DUMP =====
================================================================
  Loaded a process at input/proc/p1s, PID: 4 PRIO: 0
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  3
  CPU 1: Put process  2 to run queue
  CPU 1: Dispatched process  4
  CPU 0: Put process  3 to run queue
  CPU 0: Dispatched process  1
===== PHYSICAL MEMORY AFTER READING =====
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 00000114: 100
===== PHYSICAL MEMORY END-DUMP =====
================================================================
===== PHYSICAL MEMORY AFTER WRITING =====
write region=2 offset=20 value=102
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 00000114: 102
===== PHYSICAL MEMORY END-DUMP =====
================================================================
  CPU 1: Put process  4 to run queue
  CPU 1: Dispatched process  3
===== PHYSICAL MEMORY AFTER READING =====
read region=2 offset=20 value=102
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 00000114: 102
===== PHYSICAL MEMORY END-DUMP =====
================================================================
===== PHYSICAL MEMORY AFTER WRITING =====
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 00000114: 103
===== PHYSICAL MEMORY END-DUMP =====
================================================================
===== PHYSICAL MEMORY AFTER READING =====
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 00000114: 103
===== PHYSICAL MEMORY END-DUMP =====
================================================================
  CPU 1: Processed  3 has finished
  CPU 1: Dispatched process  4
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
===== PHYSICAL MEMORY AFTER DEALLOCATION =====
PID=1 - Region=4
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
================================================================
  CPU 0: Processed  1 has finished
  CPU 0: Dispatched process  2
  CPU 1: Processed  4 has finished
  CPU 1 stopped
  CPU 0: Processed  2 has finished
  CPU 0 stopped
```

Phân tích:

- alloc 300 0, alloc 300 4: cấp phát 300 byte cho mỗi region 0 (từ 0x00000000) và 4 (từ 0x00000200), page table gồm 4 trang (0 - 1024): page 0 -> frame 1, page 1 -> frame 0, page 2 -> frame 3, page 3 -> frame 2.
- free 0: giải phóng vùng nhớ region 0, đưa vùng nhớ vào freerg_list nhưng page table giữ nguyên do không gỡ ánh xạ (unmap) từ page table và không giải phóng frame vật lý.
- alloc 100 1: cấp phát 100 byte cho region 1, bắt đầu từ địa chỉ 0, sử dụng lại vùng nhớ đã thu hồi, page table giữ nguyên.
- write 100 1 20: chạy lệnh write 100 1 20, ghi 100 vào offset 20 (0x14 theo hexadecimal) của region 1 (bắt đầu từ 0x00000000). Virtual address = 0x00000000 + 0x14 = 0x14 thuộc page 0 (offset trong page = 0x14 % 0x100 = 0x14), ánh xạ đến frame 1 (bắt đầu từ địa chỉ vật lý 1*256 = 256 = 0x100) → địa chỉ vật lý là 0x100 + 0x14 = 0x114 → BYTE 0x114 = 100.
- read 1 20 20: thực thi lệnh read 1 20 20, đọc được giá trị tại offset 20 của region 1 (0x114) là 100, trùng với giá trị đã write.
- write 102 2 20: region 2 chưa được cấp phát, mặc định địa chỉ là 0x00000000, tương tự với lệnh write ở time slot 6 → 0x114 = 102.
- read 2 20 20: đọc được giá trị 102 tại offset 20 của region 2 (0x114), trùng với giá trị đã ghi phía trước.
- Tương tự, sau khi thực thi lệnh write 103 3 20, giá trị tại 0x114 = 103; thực thi lệnh read 3 20 20, nhận được giá trị 103.
- free 4, đưa vùng nhớ của region 4 vào freerg_list, page table vẫn giữ nguyên do không gỡ ánh xạ (unmap) từ page table và không giải phóng frame vật lý

3.3.2 Trường hợp input os_1_mlq_paging_small_1K

Cấu hình đầu vào:

- Configuration file `os_1_mlq_paging_small_1K` để khởi động Simple Operating System:

```
2 4 8
2048 16777216 0 0 0
1 p0s  130
2 s3  39
4 m1s  15
6 s2  120
7 m0s  120
9 p1s  15
11 s0 38
16 s1 0
```

- Các file tiến trình:

p0s	s3	m1s	s2	m0s	p1s	s0	s1
1 14	7 11	1 6	20 12	1 6	1 10	12 15	20 7
calc	calc	alloc 300 0	calc	alloc 300 0	calc	calc	calc
alloc 300 0	calc	alloc 100 1	calc	alloc 100 1	calc	calc	calc
alloc 300 4	calc	free 0	calc	free 0	calc	calc	calc
free 0	calc	alloc 100 2	calc	alloc 100 2	calc	calc	calc
alloc 100 1	calc	free 2	calc	write 102 1 20	calc	calc	calc
write 100 1 20	calc	free 1	calc	write 1 2 1000	calc	calc	calc
read 1 20 20	calc		calc		calc	calc	calc
write 102 2 20	calc		calc		calc	calc	
read 2 20 20	calc		calc		calc	calc	
write 103 3 20	calc		calc		calc	calc	

read 3 20 20 calc free 4 calc	calc		calc calc			calc calc calc calc calc	

Bảng 6: Các tiến trình của os_1_mlq_paging_small_1K

Output thực thi:

```
    Time slot   0
ld_routine
Time slot   1
  Loaded a process at input/proc/p0s, PID: 1 PRIO: 130
  CPU 3: Dispatched process  1
Time slot   2
===== 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
================================================================
  Loaded a process at input/proc/s3, PID: 2 PRIO: 39
Time slot   3
  CPU 3: Put process  1 to run queue
  CPU 3: Dispatched process  2
  CPU 1: Dispatched process  1
===== 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
  Loaded a process at input/proc/m1s, PID: 3 PRIO: 15
  CPU 2: Dispatched process  3
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=3 - Region=0 - Address=00000000 - Size=300 byte
print_pgtbl: 0 - 512
00000000: 80000005
00000004: 80000004
Page Number: 0 -> Frame Number: 5
Page Number: 1 -> Frame Number: 4
================================================================
===== PHYSICAL MEMORY AFTER DEALLOCATION =====
PID=1 - Region=0
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   5
  CPU 3: Put process  2 to run queue
  CPU 3: Dispatched process  2
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=3 - Region=1 - Address=0000012c - Size=100 byte
  CPU 1: Put process  1 to run queue
  CPU 1: Dispatched process  1
print_pgtbl: 0 - 512
```

```
00000000: 80000005
00000004: 80000004
Page Number: 0 -> Frame Number: 5
Page Number: 1 -> Frame Number: 4
================================================================
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=1 - Region=1 - Address=00000000 - 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
  Loaded a process at input/proc/s2, PID: 4 PRIO: 120
===== PHYSICAL MEMORY AFTER WRITING =====
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 00000114: 100
===== PHYSICAL MEMORY END-DUMP =====
================================================================
  CPU 2: Put process  3 to run queue
```

```
  CPU 2: Dispatched process  3
===== PHYSICAL MEMORY AFTER DEALLOCATION =====
PID=3 - Region=0
print_pgtbl: 0 - 512
00000000: 80000005
00000004: 80000004
Page Number: 0 -> Frame Number: 5
Page Number: 1 -> Frame Number: 4
================================================================
  CPU 0: Dispatched process  4
Time slot   7
  Loaded a process at input/proc/m0s, PID: 5 PRIO: 120
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=3 - Region=2 - Address=00000000 - Size=100 byte
print_pgtbl: 0 - 512
00000000: 80000005
00000004: 80000004
Page Number: 0 -> Frame Number: 5
Page Number: 1 -> Frame Number: 4
================================================================
  CPU 3: Put process  2 to run queue
  CPU 3: Dispatched process  2
  CPU 1: Put process  1 to run queue
  CPU 1: Dispatched process  5
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=5 - Region=0 - Address=00000000 - Size=300 byte
print_pgtbl: 0 - 512
00000000: 80000007
00000004: 80000006
Page Number: 0 -> Frame Number: 7
Page Number: 1 -> Frame Number: 6
================================================================
Time slot   8
  CPU 2: Put process  3 to run queue
  CPU 2: Dispatched process  3
===== PHYSICAL MEMORY AFTER DEALLOCATION =====
```

```
PID=3 - Region=2
print_pgtbl: 0 - 512
00000000: 80000005
00000004: 80000004
Page Number: 0 -> Frame Number: 5
Page Number: 1 -> Frame Number: 4
================================================================
  CPU 0: Put process  4 to run queue
  CPU 0: Dispatched process  4
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=5 - Region=1 - Address=0000012c - Size=100 byte
print_pgtbl: 0 - 512
00000000: 80000007
00000004: 80000006
Page Number: 0 -> Frame Number: 7
Page Number: 1 -> Frame Number: 6
================================================================
Time slot   9
  Loaded a process at input/proc/p1s, PID: 6 PRIO: 15
  CPU 3: Put process  2 to run queue
  CPU 3: Dispatched process  6
  CPU 1: Put process  5 to run queue
  CPU 1: Dispatched process  2
===== PHYSICAL MEMORY AFTER DEALLOCATION =====
PID=3 - Region=1
print_pgtbl: 0 - 512
00000000: 80000005
00000004: 80000004
Page Number: 0 -> Frame Number: 5
Page Number: 1 -> Frame Number: 4
================================================================
Time slot  10
  CPU 2: Processed  3 has finished
  CPU 2: Dispatched process  5
===== PHYSICAL MEMORY AFTER DEALLOCATION =====
PID=5 - Region=0
```

```
print_pgtbl: 0 - 512
00000000: 80000007
00000004: 80000006
Page Number: 0 -> Frame Number: 7
Page Number: 1 -> Frame Number: 6
================================================================
  CPU 0: Put process  4 to run queue
  CPU 0: Dispatched process  4
Time slot  11
  CPU 3: Put process  6 to run queue
  CPU 3: Dispatched process  6
  CPU 1: Put process  2 to run queue
  CPU 1: Dispatched process  2
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=5 - Region=2 - Address=00000000 - Size=100 byte
print_pgtbl: 0 - 512
00000000: 80000007
00000004: 80000006
Page Number: 0 -> Frame Number: 7
Page Number: 1 -> Frame Number: 6
================================================================
  Loaded a process at input/proc/s0, PID: 7 PRIO: 38
Time slot  12
  CPU 2: Put process  5 to run queue
  CPU 2: Dispatched process  7
  CPU 0: Put process  4 to run queue
  CPU 0: Dispatched process  5
===== PHYSICAL MEMORY AFTER WRITING =====
write region=1 offset=20 value=102
print_pgtbl: 0 - 512
00000000: 80000007
00000004: 80000006
Page Number: 0 -> Frame Number: 7
Page Number: 1 -> Frame Number: 6
================================================================
===== PHYSICAL MEMORY DUMP =====
```

```
BYTE 00000114: 100
BYTE 00000640: 102
===== PHYSICAL MEMORY END-DUMP =====
================================================================
Time slot  13
  CPU 3: Put process  6 to run queue
  CPU 3: Dispatched process  6
  CPU 1: Put process  2 to run queue
  CPU 1: Dispatched process  2
===== PHYSICAL MEMORY AFTER WRITING =====
write region=2 offset=1000 value=1
print_pgtbl: 0 - 512
00000000: c0000000
00000004: 80000006
Page Number: 0 -> Frame Number: 0
Page Number: 1 -> Frame Number: 6
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000114: 100
BYTE 00000640: 102
BYTE 000007e8: 1
===== PHYSICAL MEMORY END-DUMP =====
================================================================
Time slot  14
  CPU 1: Processed  2 has finished
  CPU 1: Dispatched process  4
  CPU 0: Processed  5 has finished
  CPU 0: Dispatched process  1
===== PHYSICAL MEMORY AFTER READING =====
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 00000114: 100
BYTE 00000640: 102
BYTE 000007e8: 1
===== PHYSICAL MEMORY END-DUMP =====
================================================================
  CPU 2: Put process  7 to run queue
  CPU 2: Dispatched process  7
Time slot  15
  CPU 3: Put process  6 to run queue
  CPU 3: Dispatched process  6
===== PHYSICAL MEMORY AFTER WRITING =====
write region=2 offset=20 value=102
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 00000114: 102
BYTE 00000640: 102
BYTE 000007e8: 1
===== PHYSICAL MEMORY END-DUMP =====
================================================================
Time slot  16
  Loaded a process at input/proc/s1, PID: 8 PRIO: 0
  CPU 2: Put process  7 to run queue
```

```
  CPU 2: Dispatched process  8
  CPU 1: Put process  4 to run queue
  CPU 1: Dispatched process  7
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  4
Time slot  17
  CPU 3: Put process  6 to run queue
  CPU 3: Dispatched process  6
Time slot  18
  CPU 1: Put process  7 to run queue
  CPU 1: Dispatched process  7
  CPU 0: Put process  4 to run queue
  CPU 0: Dispatched process  4
  CPU 2: Put process  8 to run queue
  CPU 2: Dispatched process  8
Time slot  19
  CPU 3: Processed  6 has finished
  CPU 3: Dispatched process  1
===== PHYSICAL MEMORY AFTER READING =====
read region=2 offset=20 value=102
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 00000114: 102
BYTE 00000640: 102
BYTE 000007e8: 1
===== PHYSICAL MEMORY END-DUMP =====
================================================================
```

```
Time slot  20
  CPU 1: Put process  7 to run queue
  CPU 1: Dispatched process  7
===== PHYSICAL MEMORY AFTER WRITING =====
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 00000114: 103
BYTE 00000640: 102
BYTE 000007e8: 1
===== PHYSICAL MEMORY END-DUMP =====
================================================================
  CPU 0: Processed  4 has finished
  CPU 0 stopped
  CPU 2: Put process  8 to run queue
  CPU 2: Dispatched process  8
Time slot  21
  CPU 3: Put process  1 to run queue
  CPU 3: Dispatched process  1
===== PHYSICAL MEMORY AFTER READING =====
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 00000114: 103
BYTE 00000640: 102
BYTE 000007e8: 1
===== PHYSICAL MEMORY END-DUMP =====
================================================================
Time slot  22
  CPU 1: Put process  7 to run queue
  CPU 1: Dispatched process  7
  CPU 2: Put process  8 to run queue
  CPU 2: Dispatched process  8
Time slot  23
  CPU 3: Put process  1 to run queue
  CPU 3: Dispatched process  1
===== PHYSICAL MEMORY AFTER DEALLOCATION =====
PID=1 - Region=4
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
================================================================
  CPU 2: Processed  8 has finished
  CPU 2 stopped
Time slot  24
  CPU 1: Put process  7 to run queue
  CPU 1: Dispatched process  7
Time slot  25
```

```
  CPU 3: Processed  1 has finished
  CPU 3 stopped
Time slot  26
  CPU 1: Put process  7 to run queue
  CPU 1: Dispatched process  7
Time slot  27
  CPU 1: Processed  7 has finished
  CPU 1 stopped

```

Phân tích:

- PID 1 (p0s):
  - Time slot 2, 3: cấp phát 300 byte cho mỗi region 0 (từ 0x00000000) và 4 (từ 0x00000200), page table gồm 4 trang (0 - 1024): page 0 -> frame 1, page 1 -> frame 0, page 2 -> frame 3, page 3 -> frame 2.
  - Time slot 4: giải phóng vùng nhớ region 0, đưa vùng nhớ vào freerg_list nhưng page table giữ nguyên do không gỡ ánh xạ (unmap) từ page table và không giải phóng frame vật lý.
  - Time slot 5: cấp phát 100 byte cho region 1, bắt đầu từ địa chỉ 0, sử dụng lại vùng nhớ đã thu hồi, page table giữ nguyên.
  - Time slot 6: chạy lệnh write 100 1 20, ghi 100 vào offset 20 (0x14 theo hexadecimal) của region 1 (bắt đầu từ 0x00000000). Virtual address = 0x00000000 + 0x14 = 0x14 thuộc page 0 (offset trong page = 0x14 % 0x100 = 0x14), ánh xạ đến frame 1 (bắt đầu từ địa chỉ vật lý 1*256 = 256 = 0x100) → địa chỉ vật lý là 0x100 + 0x14 = 0x114 → BYTE 0x114 = 100.
  - Time slot 14: thực thi lệnh read 1 20 20, đọc được giá trị tại offset 20 của region 1 (0x114) là 100, trùng với giá trị đã write.
  - Time slot 15: thực thi lệnh write 102 2 20, region 2 chưa được cấp phát, mặc định địa chỉ là 0x00000000, tương tự với lệnh write ở time slot 6 → 0x114 = 102.

  - Time slot 19: thực thi lệnh read 2 20 20, đọc được giá trị 102 tại offset 20 của region 2 (0x114), trùng với giá trị đã ghi phía trước.
  - Tương tự với Time slot 20 và 21, sau khi thực thi lệnh write 103 3 20, giá trị tại 0x114 = 103; sau khi thực thi lệnh read 3 20 20, nhận được giá trị 103.
  - Time slot 23: thực thi lệnh free 4, đưa vùng nhớ của region 4 vào freerg_list.

- PID 3 (m1s):
  - Time slot 4: cấp phát 300 byte cho region 0, page table gồm 2 trang (0 - 512): page 1 -> frame 5, page 1 -> frame 4
  - Time slot 5: cấp phát 100 byte cho region 1, từ địa chỉ 300. Page table giữ nguyên.
  - Time slot 6: free region 0, đưa vùng region 0 - 300 vào freerg_list.
  - Time slot 7: cấp phát 100 byte cho region 2, bắt đầu từ địa chỉ 0, page table giữ nguyên.
  - Time slot 8: free region 2, đưa vùng nhớ 0 - 100 vào freerg_list, page table giữ nguyên.
  - Time slot 9: free region 1, đưa vùng nhớ 300 - 400 vào freerg_list.

- PID 5 (m0s):
  - Time slot 7: cấp phát 300 byte cho region 0, bắt đầu từ địa chỉ 0. Page table gồm 2 trang (0 - 512): page 0 -> frame 7, page 1 -> frame 6.
  - Time slot 8: cấp phát 100 byte cho region 1, bắt đầu từ địa chỉ 0x12c (300), page table giữ nguyên.
  - Time slot 10: free region 0, đưa vùng nhớ từ 0 - 300 vào freerg_list
  - Time slot 11: cấp phát 100 byte cho region 2, bắt đầu từ địa chỉ 0.
  - Time slot 12: thực thi lệnh write 102 1 20, ghi 102 vào offset 20 (0x14 theo hexadecimal) của region 1 (bắt đầu từ 0x12c). Virtual address =

0x0000012c + 0x14 = 0x140 thuộc page 1 (offset trong page = 0x140 % 0x100 = 0x40), ánh xạ đến frame 6 (bắt đầu từ địa chỉ vật lý 6*256 = 1536 = 0x600) → địa chỉ vật lý là 0x600 + 0x40 = 0x640 → BYTE 0x640 = 102

– Time slot 13: thực thi lệnh write 1 2 1000. Viết giá trị 1 vào offset 1000 (0x3e8) của region 2 (bắt đầu từ địa chỉ 0x00000000). Khi đó, địa chỉ ảo được tính là 0x00000000 + 0x3e8 = 0x3e8, tương ứng với page 3 với offset là 0xE8 trong trang. Tuy nhiên, region 2 trước đó chỉ được cấp phát 100 byte, tức là chưa đầy 1 trang (256 byte). Việc truy cập vào page 3, vượt ngoài phạm vi được cấp phát, gây ra page fault. Hệ thống lại nhận diện truy cập này là hợp lệ về logic, nhưng không còn đủ frame trống trong RAM nên đã swap out page 0 để giải phóng frame, rồi ánh xạ page 3 vào đó. Điều này thể hiện rõ qua giá trị Page Table Entry của page 0 chuyển từ 0x800000XX (bit 31 = 1: valid, bit 30 = 0: present in RAM) sang 0xC0000000 (bit 31 = 1: valid, bit 30 = 1: not present), tức là trang hợp lệ nhưng không còn trong RAM mà đã bị swap out ra SWAP.

3.3.3 Trường hợp input os_1_singleCPU_mlq

Cấu hình đầu vào:

- Configuration file `os_1_singleCPU_mlq` với `#define MM_FIXED_MEMSZ` để khởi động Simple Operating System:

```
2 1  8
1 s4   4
2 s3   3
4 m1s  2
6 s2   3
7 m0s  3
9 p1s  2
```

```
11 s0  1
16 s1  0
```

- Các file tiến trình:

s4	s3	m1s	s2	m0s	p1s	s0	s1
20 7	7 11	1 6	20 12	1 6	1 10	12 15	20 7
calc	calc	alloc 300 0	calc	alloc 300 0	calc	calc	calc
calc	calc	alloc 100 1	calc	alloc 100 1	calc	calc	calc
calc	calc	free 0	calc	free 0	calc	calc	calc
calc	calc	alloc 100 2	calc	alloc 100 2	calc	calc	calc
calc	calc	free 2	calc	write 102 1 20	calc	calc	calc
calc	calc	free 1	calc	write 1 2 1000	calc	calc	calc
calc	calc		calc		calc	calc	calc
	calc		calc		calc	calc	
	calc		calc		calc	calc	
	calc		calc		calc	calc	
	calc		calc			calc	
			calc			calc	
						calc	
						calc	
						calc	

Bảng 7: Các tiến trình của os_1_singleCPU_mlq

Output thực thi:

```
    Time slot   0
ld_routine
Time slot   1
  Loaded a process at input/proc/s4, PID: 1 PRIO: 4
Time slot   2
  Loaded a process at input/proc/s3, PID: 2 PRIO: 3
```

```
  CPU 0: Dispatched process  2
Time slot   3
Time slot   4
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  2
  Loaded a process at input/proc/m1s, PID: 3 PRIO: 2
Time slot   5
Time slot   6
  Loaded a process at input/proc/s2, PID: 4 PRIO: 3
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  3
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=3 - 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   7
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=3 - Region=1 - Address=0000012c - Size=100 byte
print_pgtbl: 0 - 512
00000000: 80000001
00000004: 80000000
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
================================================================
  Loaded a process at input/proc/m0s, PID: 5 PRIO: 3
Time slot   8
  CPU 0: Put process  3 to run queue
  CPU 0: Dispatched process  3
===== PHYSICAL MEMORY AFTER DEALLOCATION =====
PID=3 - Region=0
print_pgtbl: 0 - 512
00000000: 80000001
```

```
00000004: 80000000
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
================================================================
Time slot   9
  Loaded a process at input/proc/p1s, PID: 6 PRIO: 2
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=3 - Region=2 - Address=00000000 - Size=100 byte
print_pgtbl: 0 - 512
00000000: 80000001
00000004: 80000000
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
================================================================
Time slot  10
  CPU 0: Put process  3 to run queue
  CPU 0: Dispatched process  6
Time slot  11
  Loaded a process at input/proc/s0, PID: 7 PRIO: 1
Time slot  12
  CPU 0: Put process  6 to run queue
  CPU 0: Dispatched process  7
Time slot  13
Time slot  14
  CPU 0: Put process  7 to run queue
  CPU 0: Dispatched process  7
Time slot  15
Time slot  16
  Loaded a process at input/proc/s1, PID: 8 PRIO: 0
  CPU 0: Put process  7 to run queue
  CPU 0: Dispatched process  8
Time slot  17
Time slot  18
  CPU 0: Put process  8 to run queue
  CPU 0: Dispatched process  8
Time slot  19
```

```
Time slot  20
  CPU 0: Put process  8 to run queue
  CPU 0: Dispatched process  8
Time slot  21
Time slot  22
  CPU 0: Put process  8 to run queue
  CPU 0: Dispatched process  8
Time slot  23
  CPU 0: Processed  8 has finished
  CPU 0: Dispatched process  7
Time slot  24
Time slot  25
  CPU 0: Put process  7 to run queue
  CPU 0: Dispatched process  7
Time slot  26
Time slot  27
  CPU 0: Put process  7 to run queue
  CPU 0: Dispatched process  7
Time slot  28
Time slot  29
  CPU 0: Put process  7 to run queue
  CPU 0: Dispatched process  7
Time slot  30
Time slot  31
  CPU 0: Put process  7 to run queue
  CPU 0: Dispatched process  7
Time slot  32
Time slot  33
  CPU 0: Put process  7 to run queue
  CPU 0: Dispatched process  7
Time slot  34
  CPU 0: Processed  7 has finished
  CPU 0: Dispatched process  3
===== PHYSICAL MEMORY AFTER DEALLOCATION =====
PID=3 - Region=2
print_pgtbl: 0 - 512
```

```
00000000: 80000001
00000004: 80000000
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
================================================================
Time slot  35
===== PHYSICAL MEMORY AFTER DEALLOCATION =====
PID=3 - Region=1
print_pgtbl: 0 - 512
00000000: 80000001
00000004: 80000000
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
================================================================
Time slot  36
  CPU 0: Processed  3 has finished
  CPU 0: Dispatched process  6
Time slot  37
Time slot  38
  CPU 0: Put process  6 to run queue
  CPU 0: Dispatched process  6
Time slot  39
Time slot  40
  CPU 0: Put process  6 to run queue
  CPU 0: Dispatched process  6
Time slot  41
Time slot  42
  CPU 0: Put process  6 to run queue
  CPU 0: Dispatched process  6
Time slot  43
Time slot  44
  CPU 0: Processed  6 has finished
  CPU 0: Dispatched process  5
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=5 - Region=0 - Address=00000000 - Size=300 byte
print_pgtbl: 0 - 512
```

```
00000000: 80000003
00000004: 80000002
Page Number: 0 -> Frame Number: 3
Page Number: 1 -> Frame Number: 2
================================================================
Time slot  45
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=5 - Region=1 - Address=0000012c - Size=100 byte
print_pgtbl: 0 - 512
00000000: 80000003
00000004: 80000002
Page Number: 0 -> Frame Number: 3
Page Number: 1 -> Frame Number: 2
================================================================
Time slot  46
  CPU 0: Put process  5 to run queue
  CPU 0: Dispatched process  5
===== PHYSICAL MEMORY AFTER DEALLOCATION =====
PID=5 - Region=0
print_pgtbl: 0 - 512
00000000: 80000003
00000004: 80000002
Page Number: 0 -> Frame Number: 3
Page Number: 1 -> Frame Number: 2
================================================================
Time slot  47
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=5 - Region=2 - Address=00000000 - Size=100 byte
print_pgtbl: 0 - 512
00000000: 80000003
00000004: 80000002
Page Number: 0 -> Frame Number: 3
Page Number: 1 -> Frame Number: 2
================================================================
Time slot  48
  CPU 0: Put process  5 to run queue
```

```
  CPU 0: Dispatched process  5
===== PHYSICAL MEMORY AFTER WRITING =====
write region=1 offset=20 value=102
print_pgtbl: 0 - 512
00000000: 80000003
00000004: 80000002
Page Number: 0 -> Frame Number: 3
Page Number: 1 -> Frame Number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000240: 102
===== PHYSICAL MEMORY END-DUMP =====
================================================================
Time slot  49
===== PHYSICAL MEMORY AFTER WRITING =====
write region=2 offset=1000 value=1
print_pgtbl: 0 - 512
00000000: c0000000
00000004: 80000002
Page Number: 0 -> Frame Number: 0
Page Number: 1 -> Frame Number: 2
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000240: 102
BYTE 000003e8: 1
===== PHYSICAL MEMORY END-DUMP =====
================================================================
Time slot  50
  CPU 0: Processed  5 has finished
  CPU 0: Dispatched process  2
Time slot  51
Time slot  52
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  2
Time slot  53
Time slot  54
```

```
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  2
Time slot  55
Time slot  56
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  2
Time slot  57
  CPU 0: Processed  2 has finished
  CPU 0: Dispatched process  4
Time slot  58
Time slot  59
  CPU 0: Put process  4 to run queue
  CPU 0: Dispatched process  4
Time slot  60
Time slot  61
  CPU 0: Put process  4 to run queue
  CPU 0: Dispatched process  4
Time slot  62
Time slot  63
  CPU 0: Put process  4 to run queue
  CPU 0: Dispatched process  4
Time slot  64
Time slot  65
  CPU 0: Put process  4 to run queue
  CPU 0: Dispatched process  4
Time slot  66
Time slot  67
  CPU 0: Put process  4 to run queue
  CPU 0: Dispatched process  4
Time slot  68
Time slot  69
  CPU 0: Processed  4 has finished
  CPU 0: Dispatched process  1
Time slot  70
Time slot  71
  CPU 0: Put process  1 to run queue
```

```
  CPU 0: Dispatched process  1
Time slot  72
Time slot  73
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
Time slot  74
Time slot  75
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
Time slot  76
  CPU 0: Processed  1 has finished
  CPU 0 stopped
```

Phân tích:

- PID 3 (m1s):
  - Time slot 6: thực thi lệnh alloc 300 0, cấp phát 300 byte cho region 0 bắt đầu từ địa chỉ 0, page table gồm 2 trang (0 - 512): page 0 -> frame 1, page 1 -> frame 0
  - Time slot 7: thực thi lệnh alloc 100 1, cấp phát 100 byte cho region 1, bắt đầu từ địa chỉ 0x0000012c (300). Page table giữ nguyên.
  - Time slot 8: thực thi lệnh free 0, giải phóng region 0, đưa vùng region 0 (0-300) vào freerg_list.
  - Time slot 9: thực thi lệnh alloc 100 2, cấp phát 100 byte cho region 2, bắt đầu từ địa chỉ 0. Page table giữ nguyên.
  - Time slot 34: thực thi lệnh free 2, giải phóng region 2, đưa vùng region 2 (300-400) vào freerg_list. Page table giữ nguyên.
  - Time slot 35: thực thi lệnh free 1, giải phóng region 1, đưa vùng region 1 (0-100) vào freerg_list. Page table giữ nguyên.
- PID 5 (m0s):
  - Time slot 44: thực thi lệnh alloc 300 0, cấp phát 300 byte cho region 0 bắt đầu từ địa chỉ 0, page table gồm 2 trang (0 - 512): page 0 -> frame

3, page 1 -> frame 2.

- Time slot 45: thực thi lệnh alloc 100 1, cấp phát 100 byte cho region 1, bắt đầu từ địa chỉ 0x0000012c (300). Page table giữ nguyên.
- Time slot 46: thực thi lệnh free 0, giải phóng region 0, đưa vùng region 0 (0-300) vào freerg_list.
- Time slot 47: thực thi lệnh alloc 100 2, cấp phát 100 byte cho region 2, bắt đầu từ địa chỉ 0. Page table giữ nguyên.
- Time slot 48: thực thi lệnh write 102 1 20, ghi 102 vào offset 20 (0x14 theo hexadecimal) của region 1 (bắt đầu từ 0x12c). Virtual address = 0x0000012c + 0x14 = 0x140 thuộc page 1 (offset trong page = 0x140 % 0x100 = 0x40), ánh xạ đến frame 2 (bắt đầu từ địa chỉ vật lý 2*256 = 512 = 0x200) → địa chỉ vật lý là 0x200 + 0x40 = 0x240 → BYTE 0x240 = 102
- Time slot 49: thực thi lệnh write 1 2 1000. Ghi giá trị 1 vào offset 1000 (0x3E8) của region 2, bắt đầu từ địa chỉ 0x00000000. Khi đó, địa chỉ ảo = 0x00000000 + 0x3E8 = 0x3E8, tức nằm trong page 3 (offset 0xE8 trong trang). Tuy nhiên, trước đó region 2 chỉ được cấp phát 100 byte, chưa đủ 1 trang. Việc truy cập vượt quá phạm vi được cấp phát gây ra page fault, buộc hệ thống phải xử lý bằng cách cấp phát thêm một trang mới. Do không còn frame trống, hệ thống swap out page 0 để thu hồi frame, rồi ánh xạ page 3 vào frame này. Điều này thể hiện qua giá trị Page Table Entry của page 0 chuyển từ 0x800000XX (bit 31 = 1: valid, bit 30 = 0: present in RAM) sang 0xC0000000 (bit 31 = 1: valid, bit 30 = 1: not present), tức là trang hợp lệ nhưng không còn trong RAM mà đã bị swap out ra SWAP.

3.4 Trả lời câu hỏi

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?

Trả lời

Trong thiết kế bộ nhớ phân đoạn của hệ điều hành đơn giản này, việc chia không gian nhớ thành nhiều phân đoạn riêng biệt theo chức năng – như vùng mã lệnh (code), vùng dữ liệu (data) và ngăn xếp (stack) – giúp phản ánh đúng cấu trúc logic của chương trình, tạo điều kiện thuận lợi cho lập trình viên quản lý và hệ điều hành kiểm soát tài nguyên. Đặc biệt trong ngữ cảnh lập trình hướng đối tượng, mỗi lớp hoặc đối tượng có thể gán vào một phân đoạn riêng, từ đó nâng cao khả năng mở rộng, tái sử dụng và bảo trì mã nguồn. Khi truy xuất dữ liệu, chương trình chỉ cần xác định số phân đoạn và độ lệch (offset), dẫn đến quá trình truy cập bộ nhớ trở nên trực tiếp và hiệu quả hơn.

Trong môi trường đa người dùng, mỗi tiến trình hoặc người dùng được cấp phát không gian phân đoạn riêng, đảm bảo tính cô lập và an toàn giữa các tiến trình, đồng thời duy trì tính ổn định chung của hệ thống. Về mặt phần cứng, cơ chế phân đoạn như trong chế độ real mode của kiến trúc x86 cho phép kết hợp segment và offset để mở rộng không gian địa chỉ, nhờ đó chỉ dùng thanh ghi 16-bit vẫn có thể truy cập đến 1MB bộ nhớ, giảm yêu cầu về độ rộng thanh ghi so với việc phải dùng thanh ghi 20-bit. Khi một phân đoạn như vùng code hoặc data vượt quá 64KB, hệ điều hành dễ dàng cấp phát thêm phân đoạn mới mà không ảnh hưởng đến các phân đoạn hiện có, rất phù hợp với các ứng dụng lớn hoặc có nhu cầu bộ nhớ cao.

So với cơ chế phân trang, phân đoạn không gây phân mảnh trong vì bộ nhớ được cấp phát vừa khớp với kích thước yêu cầu, đồng thời các phân đoạn thường có kích thước lớn hơn trang, giúp tối ưu hóa hiệu suất. Cấu trúc bảng phân đoạn đơn giản, gọn nhẹ hơn bảng trang, làm giảm overhead quản lý và tiết kiệm tài nguyên hệ thống. Thêm vào đó, việc di chuyển một phân đoạn độc lập trong không gian nhớ trở nên dễ dàng hơn–hệ điều hành có thể tái định vị từng phân đoạn mà không cần di chuyển toàn bộ vùng địa chỉ của chương trình. Nhờ những ưu điểm này, thiết kế bộ nhớ phân đoạn mang lại cả tính linh hoạt, hiệu quả và tối ưu về cả phần mềm lẫn phần cứng.

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

Trả lời

Nếu chúng ta chia địa chỉ thành nhiều hơn 2 cấp trong hệ thống quản lý bộ nhớ phân trang (paging), điều này sẽ có một số ảnh hưởng đến RAM và bộ nhớ SWAP (bộ nhớ phụ):

- Ưu điểm:
  - **Giảm chi phí bộ nhớ:** Khi dùng phân trang nhiều cấp, mỗi bảng trang sẽ nhỏ hơn và chỉ được tạo ra khi cần thiết, nên tổng bộ nhớ cần để lưu các bảng trang cũng giảm. Điều này đặc biệt có lợi trong các hệ thống có không gian địa chỉ ảo lớn.
  - **Cấp phát bảng trang linh hoạt:** Khi chia thành nhiều cấp, việc tạo và quản lý các bảng trang trở nên dễ dàng hơn. Hệ thống chỉ tạo ra bảng trang khi thật sự cần, tùy theo chương trình đang chạy, nhờ đó tránh lãng phí bộ nhớ.
  - **Sử dụng hiệu quả bộ nhớ phân mảnh:** Phân trang nhiều cấp giúp tận dụng bộ nhớ tốt hơn. Thay vì cần một bảng trang lớn chiếm một vùng bộ nhớ liên tục, thì có thể chia nhỏ ra thành nhiều bảng trang nhỏ hơn và đặt chúng rải rác trong bộ nhớ. Nhờ đó, có thể sử dụng các phần bộ nhớ trống một cách hiệu quả hơn.
  - **Truy cập bảng trang nhanh hơn:** Trong hệ thống phân trang nhiều cấp, bảng trang được chia thành nhiều bảng nhỏ hơn. Nhờ đó, mỗi bảng sẽ có kích thước nhỏ, nên việc tìm kiếm và truy cập thông tin trong bảng diễn ra nhanh hơn.
- Nhược điểm:
  - **Tăng thời gian dịch địa chỉ:** Mỗi cấp bảng trang cần một lần truy cập bộ nhớ để tra cứu. Càng nhiều cấp thì số lần truy cập bộ nhớ tăng, khiến thời gian xử lý lâu hơn.

- **Cài đặt và quản lý phức tạp hơn:** Khi có nhiều cấp của bảng phân trang, hệ thống sẽ cần một cơ chế phức tạp hơn để quản lý các bảng trang và xử lý các lỗi (như lỗi khi không tìm thấy bảng trang). Điều này làm tăng độ phức tạp trong thiết kế và bảo trì hệ thống phân trang.

Question: What are the advantages and disadvantages of segmentation with paging?

Trả lời

Phân đoạn kết hợp với phân trang là một kỹ thuật quản lý bộ nhớ kết hợp điểm mạnh của cả hai phương pháp. Dưới đây là các ưu điểm và nhược điểm của nó:

- Ưu điểm:
  - **Linh hoạt trong quản lý bộ nhớ:** Phân đoạn cho phép chia không gian địa chỉ thành các đoạn logic riêng biệt (ví dụ: đoạn code, đoạn dữ liệu, đoạn stack...), mỗi đoạn có thể được cấp phát bộ nhớ theo đúng kích thước thực tế, không bị giới hạn bởi kích thước cố định như trong phân trang đơn thuần. Điều này giúp quản lý bộ nhớ hợp lý hơn thay vì chỉ dùng một khối lớn.
  - **Bảo vệ và chia sẻ tốt hơn:** Mỗi đoạn có thể có quyền truy cập riêng, giúp tăng khả năng bảo vệ bộ nhớ (ngăn chặn các lỗi ghi đè dữ liệu hoặc thực thi sai phần bộ nhớ). Ngoài ra, các đoạn như code có thể được chia sẻ giữa nhiều tiến trình giúp tiết kiệm tài nguyên.
  - **Mở rộng không gian địa chỉ:** Nhờ chia nhỏ đoạn thành các trang, hệ điều hành có thể ánh xạ các trang cần thiết vào RAM, còn lại lưu trên ổ cứng. Cho phép chương trình có thể hoạt động mặc dù tổng kích thước lớn hơn bộ nhớ vật lý hiện tại.
  - **Hỗ trợ bộ nhớ ảo:** Nhờ kết hợp phân đoạn và phân trang, hệ thống có thể cung cấp bộ nhớ ảo, giúp các tiến trình "thấy" như đang có nhiều bộ nhớ hơn thực tế. Khi cần, các trang sẽ được hoán đổi giữa RAM và đĩa cứng, giúp chương trình hoạt động trơn tru dù thiếu RAM.

- Nhược điểm:
  - **Phức tạp hơn trong quản lý:** Việc phải quản lý cả bảng phân đoạn và bảng trang khiến hệ thống phức tạp hơn và cần xử lý nhiều bước khi truy cập bộ nhớ.
  - **Tốn tài nguyên hơn:** Cần lưu trữ thêm bảng phân đoạn và bảng trang cho mỗi chương trình, do đó chiếm thêm không gian trong bộ nhớ.
  - **Gây phân mảnh bộ nhớ:**
    - Phân mảnh ngoài: các đoạn có kích thước khác nhau, nên có thể có những khoảng trống không dùng được.
    - Phân mảnh trong: các trang có kích thước cố định, có thể không sử dụng hết không gian trong mỗi trang.
  - **Ảnh hưởng đến hiệu suất:** Việc phải qua nhiều bước (tra bảng đoạn sau đó mới tra bảng trang) để tìm địa chỉ thật có thể làm chậm tốc độ truy cập bộ nhớ.

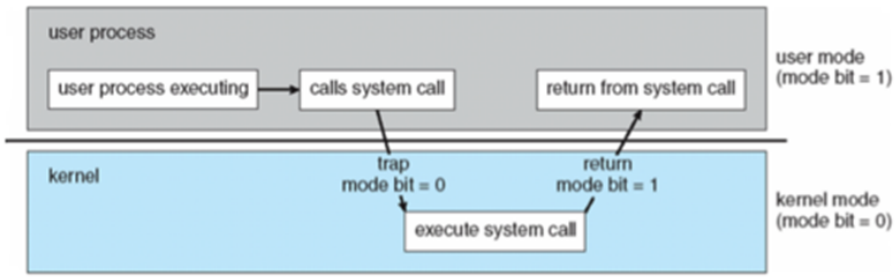
4 System Call

4.1 Giới thiệu về System Call

System Call (lời gọi hệ thống) là một phương thức để các ứng dụng sử dụng các dịch vụ của hệ điều hành như truy cập và quản lý các tập tin (file management) hay tạo và quản lý các tiến trình hệ thống (process management). System Call rất cần thiết để hệ điều hành hoạt động ổn định vì chúng chuẩn hóa cách thức truy cập vào tài nguyên hệ thống của các ứng dụng máy tính.

4.2 Nguyên lý hoạt động của System Call

Khi System Call được gọi đến, nó tạm thời dừng việc thực thi của chương trình đã gọi nó và chuyển quyền thực thi đó cho nhân của hệ điều hành (kernel). Sau đó, hệ điều hành sẽ thực hiện các thao tác của System Call được gọi đến và kích hoạt lại việc thực thi chương trình sau khi hoàn tất System Call.



Hình 5: Mô tả đơn giản về quá trình thực thi System Call

4.3 Hiện thực code

4.3.1 Hiện thực code trong file sys_killall.c

Hàm xử lý chính là hàm `__sys_killall` nhận các tham số từ tiến trình gọi đến nó và thực hiện việc tìm kiếm, sau đó gọi `free_process_memory` để giải phóng bộ nhớ trước khi loại bỏ chúng.

```
1 static pthread_mutex_t syscall_lock = PTHREAD_MUTEX_INITIALIZER;
2 void free_process_memory(struct pcb_t *proc)
```

```
{
    for (int i = 0; i < PAGING_MAX_SYMTBL_SZ; i++)
    {
        if (proc->mm->symrgtbl[i].rg_start != 0 || proc->mm->symrgtbl[
   i].rg_end != 0)
        {
            libfree(proc, i);
        }
    }
}

int __sys_killall(struct pcb_t *caller, struct sc_regs *regs)
{
    char proc_name[100];
    uint32_t raw;
    BYTE ch;
    uint32_t memrg = regs->a1;

    memset(proc_name, 0, sizeof(proc_name));
    int idx;
    for (idx = 0; idx < (int)sizeof(proc_name) - 1; idx++)
    {
        if (libread(caller, memrg, idx, &raw) < 0)
        {
            return -1;
        }
        ch = (BYTE)(raw & 0xFF);
        if (ch == '\0')
        {
            break;
        }
        if (!isprint(ch))
        {
            break;
        }
        proc_name[idx] = (char)ch;
```

```
38      }
39      proc_name[idx] = '\0';
40
41      if (proc_name[0] == '\0')
42      {
43          return -1;
44      }
45
46      printf("The procname retrieved from memregionid %u is \"%s\"\n",
    memrg, proc_name);
47
48      pthread_mutex_lock(&syscall_lock);
49      if (caller->running_list)
50      {
51          for (int i = 0; i < caller->running_list->size; i++)
52          {
53              struct pcb_t *proc = caller->running_list->proc[i];
54              if (proc && strstr(proc->path, proc_name))
55              {
56                  printf("  -> Kill running: %s (PID %d)\n", proc->path,
    proc->pid);
57                  free_process_memory(proc);
58                  caller->running_list->proc[i] = NULL;
59              }
60          }
61      }
62      pthread_mutex_unlock(&syscall_lock);
63
64      pthread_mutex_lock(&syscall_lock);
65      if (caller->mlq_ready_queue)
66      {
67          for (int prio = 0; prio < MAX_PRIO; prio++)
68          {
69              struct queue_t *q = &caller->mlq_ready_queue[prio];
70              for (int j = 0; j < q->size;)
71              {
```

```
72                  struct pcb_t *p = q->proc[j];
73                  if (p && strstr(p->path, proc_name))
74                  {
75                      printf("  -> Kill mlq ready queue: %s (PID %d)\n",
    p->path, p->pid);
76                      free_process_memory(p);
77                      for (int k = j; k + 1 < q->size; ++k)
78                          q->proc[k] = q->proc[k + 1];
79                      --q->size;
80                  }
81                  else
82                  {
83                      ++j;
84                  }
85              }
86          }
87      }
88      pthread_mutex_unlock(&syscall_lock);
89
90      return 0;
91  }
```

4.3.2 Tương tác giữa các module trong System Call

Module quản lý bộ nhớ: System Call `killall` sử dụng hàm `libread` để đọc tên các tiến trình từ vùng nhớ được chỉ định bằng tham số `region_id`.

Module điều khiển tiến trình của OS: System Call `killall` duyệt qua các tiến trình trong hệ thống và so sánh tên của các tiến trình đó. Nếu tìm thấy các tiến trình khớp tên thì System Call `killall` sẽ kết thúc tiến trình và giải phóng bộ nhớ.

Module quản lý hàng đợi: System Call `killall` sẽ duyệt qua 2 hàng đợi `running_list` và `mlq_running_list` để giải phóng các tiến trình khớp tên đã cung cấp.

4.4 Phân tích output

4.4.1 Trường hợp input os_syscall

Cấu hình đầu vào:

- Configuration file `os_syscall` để khởi động Simple Operating System:

```
2 1 1
2048 16777216 0 0 0
9 sc2  15
```

- File tiến trình `sc2`:

```
20 5
alloc 100 1
write 80 1 0
write 48 1 1
write -1 1 2
syscall 101 1
```

- Syscall 101 được định nghĩa là `sys_killall` và trong file `sc2` sẽ thực hiện 1 lần gọi System Call.

Output thực thi:

```
ld_routine
  Loaded a process at input/proc/sc2, PID: 1 PRIO: 15
  CPU 0: Dispatched process  1
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=1 - Region=1 - Address=00000000 - Size=100 byte
print_pgtbl: 0 - 256
00000000: 80000000
Page Number: 0 -> Frame Number: 0
================================================================
===== PHYSICAL MEMORY AFTER WRITING =====
```

```
write region=1 offset=0 value=80
print_pgtbl: 0 - 256
00000000: 80000000
Page Number: 0 -> Frame Number: 0
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 80
===== PHYSICAL MEMORY END-DUMP =====
================================================================
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
===== PHYSICAL MEMORY AFTER WRITING =====
write region=1 offset=1 value=48
print_pgtbl: 0 - 256
00000000: 80000000
Page Number: 0 -> Frame Number: 0
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 80
BYTE 00000001: 48
===== PHYSICAL MEMORY END-DUMP =====
================================================================
===== PHYSICAL MEMORY AFTER WRITING =====
write region=1 offset=2 value=-1
print_pgtbl: 0 - 256
00000000: 80000000
Page Number: 0 -> Frame Number: 0
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 80
BYTE 00000001: 48
BYTE 00000002: -1
===== PHYSICAL MEMORY END-DUMP =====
================================================================
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
```

```
===== PHYSICAL MEMORY AFTER READING =====
read region=1 offset=0 value=80
print_pgtbl: 0 - 256
00000000: 80000000
Page Number: 0 -> Frame Number: 0
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 80
BYTE 00000001: 48
BYTE 00000002: -1
===== PHYSICAL MEMORY END-DUMP =====
================================================================
===== PHYSICAL MEMORY AFTER READING =====
read region=1 offset=1 value=48
print_pgtbl: 0 - 256
00000000: 80000000
Page Number: 0 -> Frame Number: 0
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 80
BYTE 00000001: 48
BYTE 00000002: -1
===== PHYSICAL MEMORY END-DUMP =====
================================================================
===== PHYSICAL MEMORY AFTER READING =====
read region=1 offset=2 value=-1
print_pgtbl: 0 - 256
00000000: 80000000
Page Number: 0 -> Frame Number: 0
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000000: 80
BYTE 00000001: 48
BYTE 00000002: -1
===== PHYSICAL MEMORY END-DUMP =====
================================================================
```

```
The procname retrieved from memregionid 1 is "P0"
  CPU 0: Processed  1 has finished
  CPU 0 stopped
```

Phân tích:

- Từ output trên, quá trình thực hiện System Call diễn ra bằng việc đọc dữ liệu từ file `sc2` để nạp các System Call theo yêu cầu với PID = 1 và mức độ ưu tiên (PRIO) là 15. CPU 0 sẽ đảm nhiệm việc thực thi trên.
- Với yêu cầu "`alloc 100 1`", tiến trình sẽ cấp phát 100 bytes vùng nhớ tại REGION ID = 1 với địa chỉ là 00000000. Page Table này sẽ ánh xạ từ Page 0 vào Frame 0.
- Với các yêu cầu "`write 80 1 0`", "`write 48 1 1`", và "`write -1 1 2`", tiến trình sẽ ghi dữ liệu vào vùng nhớ. Các giá trị được ghi lần lược là 80, 48, và -1 với các các giá trị offset tương ứng là 0, 1, và 2. Mỗi lần ghi dữ liệu sẽ kèm theo quá trình dump dữ liệu.
- Sau các quá trình trên, tiến trình sẽ thực hiện "`syscall 101 1`" để đọc dữ liệu từ vùng nhớ được cấp phát và kết thúc tiến trình theo yêu cầu. Dữ liệu được đọc lần lược là 80, 48, -1 chuyển sang mã ASCII sẽ là P, 0, và ký hiệu kết thúc chuỗi "-1" tạo thành chuỗi đọc được là "P0" và kết thúc tiến trình. Như vậy, file `sc2` đã hoàn tất và không còn tiến trình nào để chạy nên CPU 0 sẽ dừng.

4.4.2 Trường hợp input os_syscall_list

Cấu hình đầu vào:

- Configuration file `os_syscall_list` để khởi động Simple Operating System:

```
2 1 1
2048 16777216 0 0 0
9 sc1  15
```

- File tiến trình `sc1`:

```
20 1
syscall 0
```

- Syscall 0 được định nghĩa là `sys_listsyscall` và trong file `sc1` sẽ thực hiện 1 lần gọi System Call.

Output thực thi:

```
ld_routine
        Loaded a process at input/proc/sc1, PID: 1 PRIO: 15
        CPU 0: Dispatched process  1
0-sys_listsyscall
17-sys_memmap
101-sys_killall
440-sys_xxxhandler
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Phân tích:

- Từ output trên, quá trình thực hiện System Call diễn ra bằng việc đọc dữ liệu từ file `sc1` để nạp các System Call theo yêu cầu với PID = 1 và mức độ ưu tiên (PRIO) là 15. CPU 0 sẽ đảm nhiệm việc thực thi trên.
- Tiến trình sẽ thực hiện "`syscall 0`" để liệt kê ra các Syscall đã được cài đặt là các System Call: `0-sys_listsyscall`, `17-sys_memmap`, `101-sys_killall`, và `440-sys_xxxhandler`.
- File `sc1` đã hoàn tất và không còn tiến trình nào để chạy nên CPU 0 sẽ dừng.

4.4.3 Trường hợp input os_sc

Cấu hình đầu vào:

- Configuration file `os_sc` để khởi động Simple Operating System:

```
2 1 1
```

```
2048 16777216 0 0 0
9 sc3  15
```

- File tiến trình `sc3`:

```
20 1
syscall 440 1
```

- Syscall 440 được định nghĩa là `sys_xxxhandler` và trong file `sc3` sẽ thực hiện 1 lần gọi System Call.

Hiện thực file `sys_xxxhandler.c`:

```c
#include "common.h"
#include "syscall.h"
#include "stdio.h"
#include "libmem.h"

int __sys_xxxhandler(struct pcb_t *caller, struct sc_regs *regs)
{
    printf("The first system call parameter %d\n", regs->a1);
    return 0;
}
```

Output thực thi:

```
ld_routine
        Loaded a process at input/proc/sc3, PID: 1 PRIO: 15
        CPU 0: Dispatched process  1
The first system call parameter 1
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Phân tích:

- Từ output trên, quá trình thực hiện System Call diễn ra bằng việc đọc dữ liệu từ file `sc3` để nạp các System Call theo yêu cầu với PID = 1 và mức độ ưu tiên (PRIO) là 15. CPU 0 sẽ đảm nhiệm việc thực thi trên.

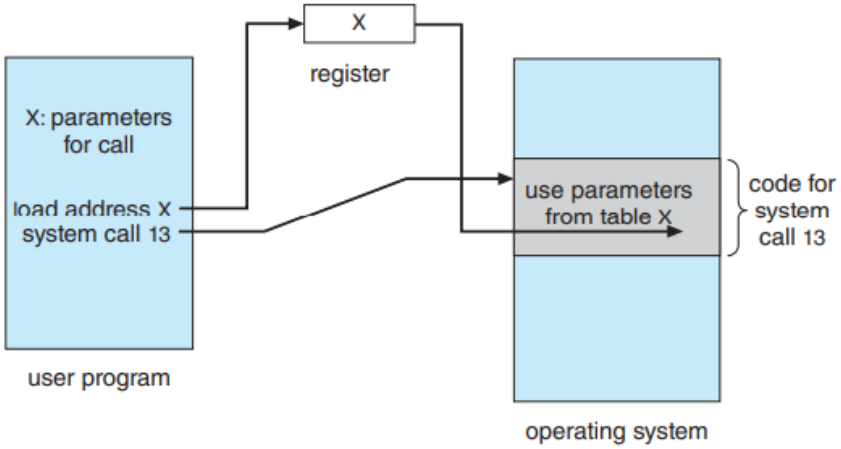
- Tiến trình sẽ thực hiện "`syscall 440 1`" để in ra tham số đầu tiên được truyền vào là `1`.
- File `sc3` đã hoàn tất và không còn tiến trình nào để chạy nên CPU 0 sẽ dừng.

4.5 Trả lời câu hỏi

Question: What is the mechanism to pass a complex argument to a system call using the limited registers?

Trả lời

Thông thường, các tham số cần thiết để System Call hoạt động được truyền trực tiếp vào các thanh ghi. Tuy nhiên, trong trường hợp số lượng các tham số lớn hơn số lượng thanh ghi hiện có hoặc tham số truyền vào phức tạp thì phương pháp truyền này không khả thi. Do đó, để giải quyết trường hợp trên, các tham số cần truyền vào được lưu vào 1 block và truyền địa chỉ của block đó vào thanh ghi để System Call sử dụng.



Hình 6: Mô tả đơn giản về quá trình truyền địa chỉ vào thanh ghi để System Call sử dụng

Khi thực hiện một System Call có truyền tham số phức tạp, địa chỉ của khối có chứa tham số đó sẽ được lưu vào thanh ghi. Tuy nhiên, dữ liệu của tham số vẫn còn nằm trong vùng nhớ của tiến trình, vì thế nên khối quản lý bộ nhớ tiến trình (Memory storing process name) sẽ đảm bảo địa chỉ được truyền vào thanh ghi là hợp lệ và tiến trình có quyền truy cập vào vùng nhớ đó.

Khi System Call được thực thi, bộ quản lý tiến trình của hệ thống (OS process control) sẽ chuyển quyền thực thi từ ứng dụng sang nhân của hệ điều hành và tiến hành thực hiện các thao tác như đọc dữ liệu của thanh ghi, sao chép dữ liệu vào vùng nhớ của nhân hệ điều hành và xử lý System Call được gọi đến. Các thông tin liên quan đến trạng thái tiến trình, nội dung của System Call, và tham chiếu tới tham số sẽ được cập nhật vào khối PCB (Process Control Block).

Trong một số trường hợp, khi System Call được gọi đến là tác vụ đọc hoặc ghi dữ liệu vào vùng nhớ và cần được Blocking, khối quản lý hàng đợi (Scheduling queue management) sẽ chuyển System Call đó sang trạng thái chờ và đợi hoàn tất việc đọc hoặc ghi dữ liệu. Sau đó, tiến trình trên sẽ được chuyển vào Ready Queue và chờ CPU cấp quyền thực thi. Khi được thực thi trở lại thì toàn bộ trạng thái của tiến trình trước khi bị block sẽ được khôi phục lại và tiếp tục diễn ra như bình thường.

Tiến hành mô phỏng việc thực hiện System Call bằng cách truyền địa chỉ. Giả sử các thanh ghi của `syscall.h` được định nghĩa như sau:

```
struct sc_regs {
    uint32_t a1;
    void*    a2; //Changing from uint32_t to void* for demo only
    uint32_t a3;
    uint32_t a4;
    uint32_t a5;
    uint32_t a6;
    uint32_t orig_ax;
    int32_t flags;
};
```

Giả sử có một sturct dữ liệu `student` được định nghĩa như sau:

```
struct student
{
    int mssv;
    char name[100];
    int age;
};
```

Khi đó, nếu muốn truyền dữ liệu vào thanh ghi của System Call để xử lý sẽ gặp

nhiều trở ngại do đây là một tập hợp nhiều kiểu dữ liệu khác nhau, do vậy nên truyền địa chỉ của khối chứa dữ liệu đó để xử lý.

Giả sử quá trình thực hiện System Call được diễn ra như sau:

```c
#include "common.h"
#include "syscall.h"
#include "stdio.h"
#include <string.h>

struct student
{
    int mssv;
    char name[100];
    int age;
};

int __sys_534handler(struct pcb_t *caller, struct sc_regs *regs)
{
    // Dữ liệu khởi tạo ban đầu
    struct student test_student;
    test_student.mssv = 2312345;
    strcpy(test_student.name, "10. He Dieu Hanh");
    test_student.age = 20;

    //Thanh ghi a2 lưu địa chỉ của dữ liệu cần xử lý
    regs->a2 = &test_student;
    printf("Gia tri cua thanh ghi a2 la: %p\n", regs->a2);

    // Thực hiện thay đổi dữ liệu của người dùng
    printf("Tuoi truoc khi thay doi: %d\n", test_student.age);
    printf("Ten truoc khi thay doi: %s\n", test_student.name);

    struct student* p_student = regs->a2;
    if (p_student == NULL) return -1;

    p_student->age = 21;
```

```
33      strcpy(p_student->name, "Tot nghiep xuat sac");
34      printf("Tuoi sau khi thay doi: %d\n", test_student.age);
35      printf("Ten sau khi thay doi: %s\n", test_student.name);
36
37      return 0;
38  }
```

Cấu hình đầu vào:

- Configuration file `os_sc_test` để khởi động Simple Operating System:

```
2 1 1
2048 16777216 0 0 0
9 sc3  15
```

- File tiến trình `sc3`:

```
20 1
syscall 440 1
```

- Syscall 440 được định nghĩa là `sys_534handler` và trong file `sc3` sẽ thực hiện 1 lần gọi System Call.

Output thực thi:

```
1  ld_routine
2          Loaded a process at input/proc/sc3, PID: 1 PRIO: 15
3          CPU 0: Dispatched process  1
4  Gia tri cua thanh ghi a2 la: 0x7fea259dcd20
5  Tuoi truoc khi thay doi: 20
6  Ten truoc khi thay doi: 10. He Dieu Hanh
7  Tuoi sau khi thay doi: 21
8  Ten sau khi thay doi: Tot nghiep xuat sac
9          CPU 0: Processed  1 has finished
10         CPU 0 stopped
```

Phân tích:

- Từ output trên, quá trình thực hiện System Call diễn ra bằng việc đọc dữ liệu từ file `sc3` để nạp các System Call theo yêu cầu với PID = 1 và mức độ ưu tiên (PRIO) là 15. CPU 0 sẽ đảm nhiệm việc thực thi trên.
- Tiến trình sẽ thực hiện "`syscall 440 1`" để thực hiện yêu cầu thay đổi giá trị của biến `age` và biến `name` của struct `test_student`. Do địa chỉ của struct `test_student` thay đổi với mỗi lần thực thi nên không thể truyền địa chỉ thông qua file `sc3` mà phải truyền trực tiếp trong `__sys_534handler`.
- File `sc3` đã hoàn tất và không còn tiến trình nào để chạy nên CPU 0 sẽ dừng.

Như vậy, việc truyền địa chỉ vào thanh ghi để thực hiện System Call giúp hạn chế rào cản về số lượng thanh ghi cũng như số lượng tham số cần truyền vào, hỗ trợ System Call xử lý được nhiều tham số.

Question: What happens if the syscall job implementation takes too long execution time?

Trả lời

Như minh họa trong Hình 5, việc thực thi một System Call chính là thao tác tạm dừng hoạt động của tiến trình đã gọi System Call đó và phải chờ đến khi System Call hoàn tất thì tiến trình đó mới được khôi phục lại. Khi một System Call thực hiện quá lâu thì tiến trình sẽ bị kẹt lại ở nhân hệ điều hành (kernel), từ đó kéo dài thực thi của chương trình. Ngoài ra, nếu System Call đó bị giữ lại ở kernel mode quá lâu cũng sẽ ảnh hưởng đến việc thực hiện System Call của các tiến trình khác.

Khi tiến trình gọi System Call, nhân hệ điều hành sẽ lưu thông tin tiến trình vào PCB (Process Control Block). Tuy nhiên, nếu syscall bị treo hoặc xử lý quá lâu, tiến trình sẽ bị chuyển sang trạng thái "BLOCKING" hoặc "WAITING", và tạm thời không được lập lịch để chạy. Điều này khiến tiến trình bị loại khỏi hàng đợi ready của khối quản lý hàng đợi (Scheduling queue management). Nếu nhiều tiến trình cùng rơi vào trạng thái này, hệ thống có thể trở nên kém hiệu quả do thiếu tiến trình sẵn sàng chạy, dẫn đến CPU rơi vào trạng thái nhàn rỗi do tiến trình đó sẽ không thể tiếp tục cho đến khi syscall hoàn tất.

5 Put It All Together

5.1 Mục tiêu tổng hợp

Sau khi hoàn thiện từng module riêng biệt như Scheduler, Memory Management và System Call, bước cuối cùng là tích hợp tất cả lại để tạo thành một hệ điều hành mô phỏng hoàn chỉnh. Ngoài ra, phần này cũng yêu cầu xử lý đồng bộ (synchronization) nhằm đảm bảo tính nhất quán khi truy cập tài nguyên dùng chung trong môi trường đa tiến trình.

5.2 Tích hợp các thành phần

- Scheduler: Quản lý cấp phát CPU cho tiến trình theo thuật toán Multi-Level Queue (MLQ).
- Memory Management: Hỗ trợ phân trang (paging), ánh xạ địa chỉ ảo sang vật lý, và xử lý hoán đổi bộ nhớ.
- System Call: Cung cấp giao diện cho tiến trình gọi đến các hàm hệ điều hành như `killall`, `listsyscall`, `memmap`.

Việc tích hợp các thành phần được thực hiện trong file `os.c`, nơi khởi động và điều phối toàn bộ hệ thống.

5.3 Hiện thực Synchronization

Trong môi trường đa CPU, có khả năng nhiều tiến trình truy cập cùng một tài nguyên (như ready queue, bảng trang, bộ nhớ vật lý) cùng lúc. Để xử lý điều này, nhóm đã sử dụng `pthread_mutex_t` để khóa (lock) các vùng tài nguyên chia sẻ, đảm bảo rằng tại một thời điểm chỉ một tiến trình được phép truy cập vào vùng đó.

Các vùng sử dụng lock gồm:

- `queue_lock`: bảo vệ `mlq_ready_queue` trong quá trình `enqueue/dequeue`.
- `mmvm_lock`: bảo vệ các thao tác cấp phát, hoán đổi bộ nhớ.

- `syscall_lock`: bảo vệ các thao tác trên danh sách tiến trình (`running_list`, `mlq_ready_queue`) trong các syscall như `killall`, tránh xung đột giữa các CPU khi đồng thời huỷ và truy xuất tiến trình.

5.4 Trả lời câu hỏi

Question: What happens if the synchronization is not handled in your Simple OS? Illustrate the problem of your simple OS (assignment outputs) by example if you have any.

Trả lời

Trong một hệ điều hành mô phỏng cho nhiều CPU, các thành phần như hàng đợi tiến trình, bảng trang và danh sách bộ nhớ vật lý là những tài nguyên chung mà nhiều luồng (CPU) phải truy cập cùng lúc. Nếu không có cơ chế đồng bộ cụ thể là các mutex lock thì việc truy cập đồng thời này sẽ dẫn đến tình trạng race condition, gây ra những lỗi khó lường và kết quả mô phỏng bị sai lệch:

- Khi nhiều CPU cùng gọi `dequeue()` trên cùng một hàng đợi `mlq_ready_queue` mà không khoá, mỗi luồng có thể nhìn thấy "hàng đợi còn ít nhất một tiến trình" rồi đồng thời lấy ra cùng một PCB. Kết quả là cùng một tiến trình được `dispatch` hai lần trên hai CPU khác nhau, và sau đó `finish` hai lần. Điều này phá vỡ hoàn toàn logic một tiến trình chỉ được chạy và kết thúc một lần, làm cho biểu đồ Gantt xuất hiện các ô trùng lặp, khiến kết quả không thể giải thích được về mặt lý thuyết.
- Trong module bộ nhớ, danh sách các vùng nhớ tự do (free-list) hoặc danh sách frame vật lý cũng là tài nguyên chung. Nếu hai CPU cùng gọi `__free()` hay `libfree()` mà không khóa, việc chèn hoặc gộp các vùng nhớ vào danh sách có thể bị chồng lấn hoặc mất liên kết.
- Hàm `pg_getpage()` xử lý hoán đổi trang giữa RAM và SWAP thông qua hai `syscall swap-out` và `swap-in`. Nếu hai CPU cùng chọn trang victim và thực hiện swap mà không khoá, chúng sẽ gửi các lệnh trao đổi khung sự kiện chồng chéo, khiến các bit `present` và `swapped` trong page table bị đặt không

nhất quán. Khi tiến trình tiếp tục truy cập trang đó, nó sẽ gặp segmentation fault hoặc đọc dữ liệu cũ.

Giả sử chạy input `sched` mà không có khóa trong hàm `get_mlq_proc` có thể cho output:

```
CPU 0: Dispatched process 3
CPU 1: Dispatched process 3
...
CPU 0: Processed 3 has finished
CPU 1: Processed 3 has finished
```

Tiến trình 3 được `dispatch` và `finish` hai lần vì cả hai CPU cùng tháo PCB ra trước khi cập nhật lại kích thước hàng đợi.

6 Kết luận

6.1 Kết luận chung

Trong bài tập lớn này, nhóm đã triển khai và tích hợp thành công bốn thành phần chính của một hệ điều hành mô phỏng:

- **Scheduler:** Áp dụng thuật toán Multilevel Queue (MLQ) kết hợp cơ chế vòng quay (round-robin) để quản lý và phân phối CPU cho các tiến trình, đảm bảo cân bằng giữa hiệu suất và tính công bằng.
- **Memory Management:** Xây dựng hệ thống phân trang ảo (paging) với khả năng ánh xạ địa chỉ ảo sang vật lý và hoán trang giữa RAM và SWAP, giúp mỗi tiến trình có không gian địa chỉ riêng biệt và an toàn.
- **System Call:** Thiết kế và cài đặt cơ chế syscall cho phép tiến trình gọi các dịch vụ hệ điều hành như `killall`, `listsyscall`, `memmap`, mô phỏng giao diện kernel thực thụ.
- **Synchronization:** Sử dụng `pthread_mutex_t` để khóa các tài nguyên chung (hàng đợi ready queue, bảng trang, danh sách tiến trình) trong môi trường đa CPU, ngăn ngừa race condition và deadlock.

Nhờ vậy, nhóm không chỉ nắm vững các khái niệm lý thuyết về lập lịch, quản lý bộ nhớ và lời gọi hệ thống, mà còn có kinh nghiệm thực tiễn trong việc thiết kế, triển khai và gỡ lỗi một hệ thống đa tiến trình đồng bộ đầy đủ.

6.2 Hướng phát triển

Để nâng cao hơn nữa năng lực và tính hoàn thiện của hệ điều hành mô phỏng, có thể xem xét các hướng sau:

- Tối ưu Scheduler với cơ chế feedback (MLFQ), tự động điều chỉnh mức ưu tiên dựa trên hành vi tiến trình để tránh starvation và cải thiện độ phản hồi.
- Thêm cơ chế hoán đổi ưu tiên (swap priority) và cache page để giảm thiểu chi phí hoán trang.

Tài liệu tham khảo

[1] A. Silberschatz, J. L. Peterson, and P. B. Galvin, *Operating system concepts*. Addison-Wesley Longman Publishing Co., Inc., 1991.

[2] N. Weizer and G. Oppenheimer, "Virtual memory management in a paging environment," in *Proceedings of the May 14-16, 1969, spring joint computer conference*, 1969, pp. 249–256.

[3] B. W. Kernighan and D. M. Ritchie, *The C programming language*. prentice-Hall, 1988.

[4] R. C. Seacord, *Secure Coding in C and C++*. Addison-Wesley, 2013.