

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 Quý - 2312864 (*Team L08_SysCallers*)
Trần Hoàng Uyên - 2313854 (*Team L08_SysCallers*)

HO CHI MINH CITY, MAY 2025

Mục lục

Danh mục hình vẽ	i
Danh mục bảng biểu	i
Danh sách thành viên và nhiệm vụ	ii
1 Mở đầu	1
1.1 Tổng quan bài tập lớn	1
1.2 Cách chạy mô phỏng	2
2 Scheduler	3
2.1 Giới thiệu về Scheduler	3
2.2 Hiện thực code	3
2.2.1 Hiện thực code trong file queue.c	3
2.2.2 Hiện thực code trong file sched.c	4
2.3 Phân tích output	6
2.3.1 Trường hợp input sched	6
2.3.2 Trường hợp input sched_0	9
2.3.3 Trường hợp input sched_1	13
2.4 Trả lời câu hỏi	18
3 Memory Management	19
3.1 Giới thiệu về Memory Management	19
3.2 Hiện thực code	20
3.2.1 Hiện thực code trong file libmem.c	20
3.2.2 Xử lý lời gọi hệ thống Syscall	29
3.2.2.1 Vai trò	29
3.2.2.2 Chức năng chính	30
3.2.3 Hiện thực code trong file mm-vm.c	31
3.2.4 Hiện thực code trong file mm.c	34

3.2.5	Hiện thực code trong file mm-memphy.c	37
3.3	Phân tích output	38
3.3.1	Trường hợp input os_0_mlq_paging	38
3.3.2	Trường hợp input os_1_mlq_paging_small_1K	46
3.3.3	Trường hợp input os_1_singleCPU_mlq	60
3.4	Trả lời câu hỏi	70
4	System Call	75
4.1	Giới thiệu về System Call	75
4.2	Nguyên lý hoạt động của System Call	75
4.3	Hiện thực code	75
4.3.1	Hiện thực code trong file sys_killall.c	75
4.3.2	Tương tác giữa các module trong System Call	78
4.4	Phân tích output	79
4.4.1	Trường hợp input os_syscall	79
4.4.2	Trường hợp input os_syscall_list	82
4.4.3	Trường hợp input os_sc	83
4.5	Trả lời câu hỏi	85
5	Put It All Together	90
5.1	Mục tiêu tổng hợp	90
5.2	Tích hợp các thành phần	90
5.3	Hiện thực Synchronization	90
5.4	Trả lời câu hỏi	91
6	Kết luận	93
6.1	Kết luận chung	93
6.2	Hướng phát triển	93
	Tài liệu tham khảo	94

Danh mục hình vẽ

1	Sơ đồ Gantt khi chạy input sched	9
2	Sơ đồ Gantt khi chạy input sched_0	12
3	Sơ đồ Gantt khi chạy input sched_1	17
4	Sơ đồ tổ chức các lớp bộ nhớ	19
5	Mô tả đơn giản về quá trình thực thi System Call	75
6	Mô tả đơn giản về quá trình truyền địa chỉ vào thanh ghi để System Call sử dụng	85

Danh mục bảng biểu

1	Danh sách thành viên và nhiệm vụ	ii
2	Các tiến trình của sched	7
3	Các tiến trình của sched_0	10
4	Các tiến trình của sched_1	13
5	Các tiến trình của os_0_mlq_paging	39
6	Các tiến trình của os_1_mlq_paging_small_1K	47
7	Các tiến trình của os_1_singleCPU_mlq	61

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 Quý	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_mfq`, 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.

```
1 void enqueue(struct queue_t *q, struct pcb_t *proc)
2 {
3     if (q == NULL || proc == NULL)
4         return;
5     if (q->size >= MAX_QUEUE_SIZE)
6         return;
7     q->proc[q->size] = proc;
8     q->size++;
```



```
9 }
10
11 struct pcb_t *dequeue(struct queue_t *q)
12 {
13     if (q == NULL || q->size == 0)
14         return NULL;
15     struct pcb_t *highest_priority_proc = q->proc[0];
16     int highest_priority_index = 0;
17     for (int i = 1; i < q->size; i++)
18     {
19         if (q->proc[i]->priority < highest_priority_proc->priority)
20         {
21             highest_priority_proc = q->proc[i];
22             highest_priority_index = i;
23         }
24     }
25     for (int i = highest_priority_index; i < q->size - 1; i++)
26     {
27         q->proc[i] = q->proc[i + 1];
28     }
29     q->size--;
30     return highest_priority_proc;
31 }
```

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.

```
1 struct pcb_t *get_mlq_proc(void)
2 {
3     struct pcb_t *proc = NULL;
4     pthread_mutex_lock(&queue_lock);
5     int all_zero = 1;
6     for (int i = 0; i < MAX_PRIO; i++)
```

```
7   {
8       if (slot[i] > 0)
9       {
10          all_zero = 0;
11          break;
12      }
13  }
14
15  if (all_zero)
16  {
17      for (int i = 0; i < MAX_PRIO; i++)
18      {
19          slot[i] = MAX_PRIO - i;
20      }
21  }
22
23  for (int prio = 0; prio < MAX_PRIO; prio++)
24  {
25      if (!empty(&mlq_ready_queue[prio]))
26      {
27          if (slot[prio] > 0)
28          {
29              proc = dequeue(&mlq_ready_queue[prio]);
30              slot[prio]--;
31              break;
32          }
33      }
34  }
35  pthread_mutex_unlock(&queue_lock);
36  return proc;
37 }
```

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()`).

```
1 void put_proc(struct pcb_t *proc)
2 {
3     proc->ready_queue = &ready_queue;
4     proc->mlq_ready_queue = mlq_ready_queue;
5     proc->running_list = &running_list;
6
7     enqueue(&running_list, proc);
8
9     return put_mlq_proc(proc);
10 }
11
12 void add_proc(struct pcb_t *proc)
13 {
14     proc->ready_queue = &ready_queue;
15     proc->mlq_ready_queue = mlq_ready_queue;
16     proc->running_list = &running_list;
17
18     enqueue(&running_list, proc);
19
20     return add_mlq_proc(proc);
21 }
```

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:

```
1 Time slot    0
2 ld_routine
3   Loaded a process at input/proc/p1s, PID: 1 PRI0: 1
4   CPU 0: Dispatched process  1
5 Time slot    1
6   Loaded a process at input/proc/p2s, PID: 2 PRI0: 0
7 Time slot    2
8   CPU 1: Dispatched process  2
9   Loaded a process at input/proc/p3s, PID: 3 PRI0: 0
```



```
10 Time slot    3
11 Time slot    4
12   CPU 0: Put process  1 to run queue
13   CPU 0: Dispatched process  3
14 Time slot    5
15 Time slot    6
16   CPU 1: Put process  2 to run queue
17   CPU 1: Dispatched process  2
18 Time slot    7
19 Time slot    8
20   CPU 0: Put process  3 to run queue
21   CPU 0: Dispatched process  3
22 Time slot    9
23 Time slot   10
24   CPU 1: Put process  2 to run queue
25   CPU 1: Dispatched process  2
26 Time slot   11
27 Time slot   12
28   CPU 0: Put process  3 to run queue
29   CPU 0: Dispatched process  3
30 Time slot   13
31 Time slot   14
32   CPU 1: Processed  2 has finished
33   CPU 1: Dispatched process  1
34 Time slot   15
35   CPU 0: Processed  3 has finished
36   CPU 0 stopped
37 Time slot   16
38 Time slot   17
39 Time slot   18
40   CPU 1: Put process  1 to run queue
41   CPU 1: Dispatched process  1
42 Time slot   19
43 Time slot   20
44   CPU 1: Processed  1 has finished
45   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 ($\text{prio} = 0$) 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:

Time slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
CPU 0		P1																			
CPU 1																					

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:

```
1 Time slot    0
2 ld_routine
3   Loaded a process at input/proc/s0, PID: 1 PRI0: 4
4 Time slot    1
5   CPU 0: Dispatched process  1
6 Time slot    2
7 Time slot    3
8   CPU 0: Put process  1 to run queue
```

```
9   CPU 0: Dispatched process  1
10  Time slot  4
11   Loaded a process at input/proc/s1, PID: 2 PRI0: 0
12  Time slot  5
13   CPU 0: Put process  1 to run queue
14   CPU 0: Dispatched process  2
15  Time slot  6
16  Time slot  7
17   CPU 0: Put process  2 to run queue
18   CPU 0: Dispatched process  2
19  Time slot  8
20  Time slot  9
21   CPU 0: Put process  2 to run queue
22   CPU 0: Dispatched process  2
23  Time slot 10
24  Time slot 11
25   CPU 0: Put process  2 to run queue
26   CPU 0: Dispatched process  2
27  Time slot 12
28   CPU 0: Processed  2 has finished
29   CPU 0: Dispatched process  1
30  Time slot 13
31  Time slot 14
32   CPU 0: Put process  1 to run queue
33   CPU 0: Dispatched process  1
34  Time slot 15
35  Time slot 16
36   CPU 0: Put process  1 to run queue
37   CPU 0: Dispatched process  1
38  Time slot 17
39  Time slot 18
40   CPU 0: Put process  1 to run queue
41   CPU 0: Dispatched process  1
42  Time slot 19
43  Time slot 20
44   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 slice.
- 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:

Time slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
CPU 0			P1	P1	P1	P2	P2	P2	P2	P2	P2	P2	P2	P1	P1	P1	P1	P1	P1	P1	P1	P1	P1	P1

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:

```
1 Time slot    0
2 ld_routine
3   Loaded a process at input/proc/s0, PID: 1 PRI0: 4
4 Time slot    1
5   CPU 0: Dispatched process  1
6 Time slot    2
7 Time slot    3
8   CPU 0: Put process  1 to run queue
9   CPU 0: Dispatched process  1
10 Time slot   4
11   Loaded a process at input/proc/s1, PID: 2 PRI0: 0
12 Time slot   5
13   CPU 0: Put process  1 to run queue
14   CPU 0: Dispatched process  2
15 Time slot   6
16   Loaded a process at input/proc/s2, PID: 3 PRI0: 0
17 Time slot   7
18   CPU 0: Put process  2 to run queue
19   CPU 0: Dispatched process  3
20   Loaded a process at input/proc/s3, PID: 4 PRI0: 0
21 Time slot   8
22 Time slot   9
23   CPU 0: Put process  3 to run queue
24   CPU 0: Dispatched process  4
25 Time slot  10
26 Time slot  11
27   CPU 0: Put process  4 to run queue
28   CPU 0: Dispatched process  4
29 Time slot  12
30 Time slot  13
31   CPU 0: Put process  4 to run queue
32   CPU 0: Dispatched process  4
33 Time slot  14
34 Time slot  15
35   CPU 0: Put process  4 to run queue
```

```
36 CPU 0: Dispatched process 4
37 Time slot 16
38 Time slot 17
39 CPU 0: Put process 4 to run queue
40 CPU 0: Dispatched process 4
41 Time slot 18
42 Time slot 19
43 CPU 0: Put process 4 to run queue
44 CPU 0: Dispatched process 4
45 Time slot 20
46 CPU 0: Processed 4 has finished
47 CPU 0: Dispatched process 2
48 Time slot 21
49 Time slot 22
50 CPU 0: Put process 2 to run queue
51 CPU 0: Dispatched process 3
52 Time slot 23
53 Time slot 24
54 CPU 0: Put process 3 to run queue
55 CPU 0: Dispatched process 2
56 Time slot 25
57 Time slot 26
58 CPU 0: Put process 2 to run queue
59 CPU 0: Dispatched process 3
60 Time slot 27
61 Time slot 28
62 CPU 0: Put process 3 to run queue
63 CPU 0: Dispatched process 2
64 Time slot 29
65 CPU 0: Processed 2 has finished
66 CPU 0: Dispatched process 3
67 Time slot 30
68 Time slot 31
69 CPU 0: Put process 3 to run queue
70 CPU 0: Dispatched process 3
71 Time slot 32
```

```
72 Time slot 33
73   CPU 0: Put process 3 to run queue
74   CPU 0: Dispatched process 3
75 Time slot 34
76 Time slot 35
77   CPU 0: Processed 3 has finished
78   CPU 0: Dispatched process 1
79 Time slot 36
80 Time slot 37
81   CPU 0: Put process 1 to run queue
82   CPU 0: Dispatched process 1
83 Time slot 38
84 Time slot 39
85   CPU 0: Put process 1 to run queue
86   CPU 0: Dispatched process 1
87 Time slot 40
88 Time slot 41
89   CPU 0: Put process 1 to run queue
90   CPU 0: Dispatched process 1
91 Time slot 42
92 Time slot 43
93   CPU 0: Put process 1 to run queue
94   CPU 0: Dispatched process 1
95 Time slot 44
96 Time slot 45
97   CPU 0: Put process 1 to run queue
98   CPU 0: Dispatched process 1
99 Time slot 46
100  CPU 0: Processed 1 has finished
101  CPU 0 stopped
```

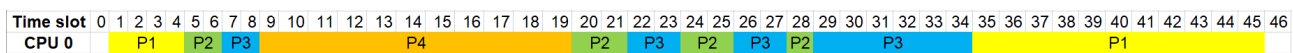
Phân tích:

- 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:

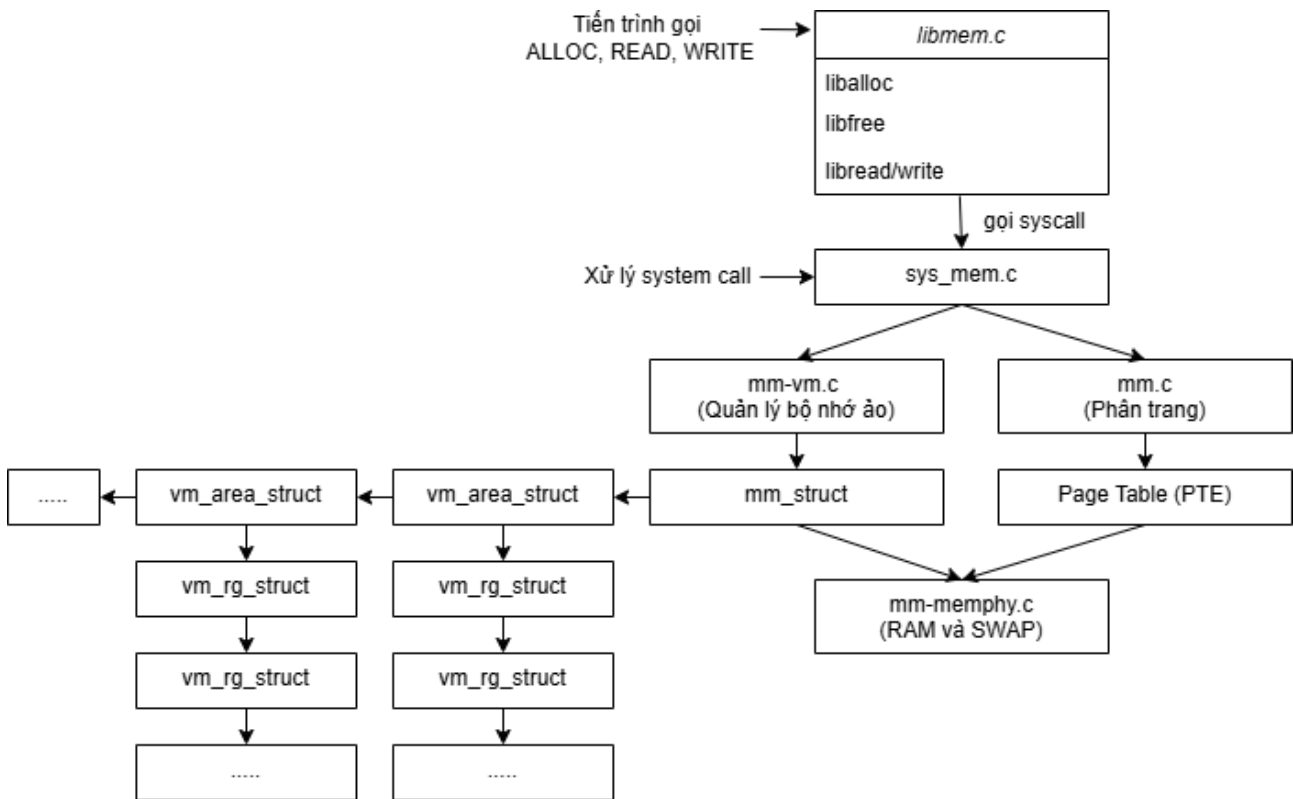
- Mỗi mức ưu tiên được gán cho một hàng đợi riêng biệt, giúp dễ dàng quản lý và chọn tiến trình có độ ưu tiên cao hơn để thực thi trước.
- Hệ thống có thể vận hành trên nhiều CPU một cách hiệu quả, mỗi CPU có thể chọn tiến trình từ hàng đợi theo nguyên tắc vòng tròn (round-robin), tối ưu hiệu suất xử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

```
1 int __alloc(struct pcb_t *caller, int vmaid, int rgid, int size, int *  
   alloc_addr)  
2 {  
3     pthread_mutex_lock(&mmvm_lock);  
4     struct vm_rg_struct rgnode;  
5     if (get_free_vmrg_area(caller, vmaid, size, &rgnode) == 0)  
6     {  
7         caller->mm->symrgtbl[rgid].rg_start = rgnode.rg_start;  
8         caller->mm->symrgtbl[rgid].rg_end = rgnode.rg_end;  
9  
10        *alloc_addr = rgnode.rg_start;  
11    }  
12    else  
13    {  
14        struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm,  
vmaid);  
15  
16        int inc_sz = PAGING_PAGE_ALIGNSZ(size);  
17  
18        int old_sbrk = cur_vma->sbrk;  
19  
20        struct sc_regs regs;  
21        regs.a1 = SYSMEM_INC_OP;  
22        regs.a2 = vmaid;  
23        regs.a3 = inc_sz;  
24  
25        int ret = syscall(caller, 17, &regs);  
26        if (ret != 0)  
27        {  
28            pthread_mutex_unlock(&mmvm_lock);  
29            return ret;  
30        }  
31        cur_vma->sbrk = old_sbrk + inc_sz;  
32        *alloc_addr = old_sbrk;  
33        if (get_free_vmrg_area(caller, vmaid, size, &rgnode) == 0)
```

```
34     {
35         caller->mm->symrgtbl[rgid].rg_start = old_sbrk;
36         caller->mm->symrgtbl[rgid].rg_end = old_sbrk + inc_sz;
37     }
38 }
39 pthread_mutex_unlock(&mmvm_lock);
40 printf("==== PHYSICAL MEMORY AFTER ALLOCATION ====\\n");
41 #ifdef IODUMP
42     printf("PID=%d - Region=%d - Address=%08lx - Size=%d byte\\n",
43         caller->pid, rgid, (unsigned long)*alloc_addr, size);
44 #ifdef PAGETBL_DUMP
45     print_pgtbl(caller, 0, -1);
46 #endif
47 #endif
48     return 0;
49 }
```

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.

```
1 int __free(struct pcb_t *caller, int vmaid, int rgid)
2 {
3     if (rgid < 0 || rgid > PAGING_MAX_SYMTBL_SZ)
4         return -1;
5
6     struct vm_rg_struct *freerg = (struct vm_rg_struct *)malloc(sizeof
7 (struct vm_rg_struct));
8     if (freerg == NULL)
9         return -1;
10
11     pthread_mutex_lock(&mmvm_lock);
12     freerg->rg_start = caller->mm->symrgtbl[rgid].rg_start;
13     freerg->rg_end = caller->mm->symrgtbl[rgid].rg_end;
14     freerg->rg_next = NULL;
15
16     enlist_vm_freerg_list(caller->mm, freerg);
```

```
17 caller->mm->symrgtbl[rgid].rg_start = 0;
18 caller->mm->symrgtbl[rgid].rg_end = 0;
19 pthread_mutex_unlock(&mmvm_lock);
20 printf("==== PHYSICAL MEMORY AFTER DEALLOCATION ====\\n");
21 #ifdef IODUMP
22 printf("PID=%d - Region=%d\\n",
23        caller->pid, rgid);
24 #ifdef PAGETBL_DUMP
25 print_pgtbl(caller, 0, -1); // print max TBL
26 #endif
27 #endif
28 return 0;
29 }
```

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).

```
1 int pg_getpage(struct mm_struct *mm, int pgn, int *fpn, struct pcb_t *
  caller)
2 {
3     pthread_mutex_lock(&mmvm_lock);
4     uint32_t pte = mm->pgd[pgn];
5
6     if (!PAGING_PAGE_PRESENT(pte))
7     {
8         int vicpgn, vicfpn, tgtfpn, swpfpn;
9         uint32_t vicpte;
10
11         find_victim_page(caller->mm, &vicpgn);
12         vicpte = mm->pgd[vicpgn];
13         vicfpn = PAGING_PTE_FPN(vicpte);
14
15         MEMPHY_get_freefp(caller->active_mswp, &swpfpn);
16
17         struct sc_regs regs_out;
```

```
18     regs_out.a1 = SYSMEM_SWP_OP;
19     regs_out.a2 = vicfpn;
20     regs_out.a3 = swpfpn;
21     int status = syscall(caller, 17, &regs_out);
22     if (status != 0)
23     {
24         pthread_mutex_unlock(&mmvm_lock);
25         return status;
26     }
27
28     pte_set_swap(&mm->pgd[vicpgn], caller->active_mswp_id, swpfpn)
29 ;
30
31     tgtfpn = vicfpn;
32
33     int tgt_swpoft = PAGING_PTE_SWP(pte);
34
35     struct sc_regs regs_in;
36     regs_in.a1 = SYSMEM_SWP_OP;
37     regs_in.a2 = tgt_swpoft;
38     regs_in.a3 = tgtfpn;
39     status = syscall(caller, 17, &regs_in);
40     if (status != 0)
41     {
42         pthread_mutex_unlock(&mmvm_lock);
43         return status;
44     }
45
46     pte_set_fpn(&mm->pgd[pgn], tgtfpn);
47     PAGING_PTE_SET_PRESENT(mm->pgd[pgn]);
48
49     enlist_pgn_node(&caller->mm->fifo_pgn, pgn);
50
51     *fpn = PAGING_FPN(mm->pgd[pgn]);
52     pthread_mutex_unlock(&mmvm_lock);
```

```
53     return 0;
54 }
```

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.

```
1 int pg_getval(struct mm_struct *mm, int addr, BYTE *data, struct pcb_t
    *caller)
2 {
3     int pgn = PAGING_PGN(addr);
4     int off = PAGING_OFFST(addr);
5     int fpn;
6
7     /* Get the page to MEMRAM, swap from MEMSWAP if needed */
8     if (pg_getpage(mm, pgn, &fpn, caller) != 0)
9         return -1; /* invalid page access */
10
11     /* TODO
12      * MEMPHY_read(caller->mram, phyaddr, data);
13      * MEMPHY READ
14      * SYSCALL 17 sys_mmap with SYMEM_IO_READ
15      */
16     int phyaddr = fpn * PAGING_PAGESZ + off;
17     return MEMPHY_read(caller->mram, phyaddr, data);
18 }
```

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.

```
1 int pg_setval(struct mm_struct *mm, int addr, BYTE value, struct pcb_t
    *caller)
2 {
3     int pgn = PAGING_PGN(addr);
4     int off = PAGING_OFFST(addr);
5     int fpn;
6
7     /* Get the page to MEMRAM, swap from MEMSWAP if needed */
8     if (pg_getpage(mm, pgn, &fpn, caller) != 0)
```

```
9         return -1; /* invalid page access */
10
11     /* TODO
12      * MEMPHY_write(caller->mram, phyaddr, value);
13      * MEMPHY WRITE
14      * SYSCALL 17 sys_memmap with SYSMEM_IO_WRITE
15      */
16     int phyaddr = fpn * PAGING_PAGESZ + off;
17     return MEMPHY_write(caller->mram, phyaddr, value);
18 }
```

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

```
1 int libread(
2     struct pcb_t *proc, // Process executing the instruction
3     uint32_t source,    // Index of source register
4     uint32_t offset,    // Source address = [source] + [offset]
5     uint32_t *destination)
6 {
7     BYTE data;
8     int val = __read(proc, 0, source, offset, &data);
9
10    /* TODO update result of reading action*/
11    // destination
12    *destination = data;
13 #ifdef IODUMP
14     printf("==== PHYSICAL MEMORY AFTER READING ==== \n");
15     printf("read region=%d offset=%d value=%d \n", source, offset, data);
16 #endif
17 #ifdef PAGETBL_DUMP
18     print_pgtbl(proc, 0, -1); // print max TBL
19 #endif
20     MEMPHY_dump(proc->mram);
21 #endif
22     return val;
23 }
```

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.

```
1 int find_victim_page(struct mm_struct *mm, int *retpgn)
2 {
3     if (mm->fifo_pgn != NULL)
4     {
5         struct pgn_t *pPage = NULL;
6         struct pgn_t *lPage = mm->fifo_pgn;
7         while (lPage->pg_next != NULL)
8         {
9             pPage = lPage;
10            lPage = lPage->pg_next;
11        }
12        *retpgn = lPage->pgn;
13        if (pPage == NULL)
14        {
15            mm->fifo_pgn = lPage->pg_next;
16        }
17        else
18        {
19            pPage->pg_next = lPage->pg_next;
20        }
21
22        free(lPage);
23
24        return 0;
25    }
26    *retpgn = -1;
27    return -1;
28 }
```

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`.


```
1 int get_free_vmrg_area(struct pcb_t *caller, int vmaid, int size,
2     struct vm_rg_struct *newrg)
3 {
4     struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid)
5     ;
6     if (cur_vma == NULL)
7         return -1;
8
9     struct vm_rg_struct *rgit = cur_vma->vm_freerg_list;
10    if (rgit == NULL)
11        return -1;
12
13    /* Probe unintialized newrg */
14    newrg->rg_start = newrg->rg_end = -1;
15
16    /* TODO Traverse on list of free vm region to find a fit space */
17    while (rgit != NULL)
18    {
19        if (rgit->rg_start + size <= rgit->rg_end)
20        {
21            newrg->rg_start = rgit->rg_start;
22            newrg->rg_end = rgit->rg_start + size;
23
24            if (rgit->rg_start + size < rgit->rg_end)
25            {
26                rgit->rg_start = rgit->rg_start + size;
27            }
28            else
29            {
30
31                struct vm_rg_struct *nextrg = rgit->rg_next;
32
33                if (nextrg != NULL)
34                {
35                    rgit->rg_start = nextrg->rg_start;
36                    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 lỗi 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_mmap(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 tạo 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_mmap` khi xảy ra trường hợp `SYSMEM_INC_OP` thì sẽ trả về giá trị của hàm `inc_vma_limit`.

```
1 case SYSMEM_INC_OP:
2     int ret = inc_vma_limit(caller, regs->a2, regs->a3);
3     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 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 retrieve 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
```

```
16  /* TODO: update the newrg boundary*/
17  newrg->rg_start = cur_vma->sbrk;
18  newrg->rg_end = newrg->rg_start + alignedsz;
19  newrg->rg_next = NULL;
20  cur_vma->sbrk = newrg->rg_end;
21  return newrg;
22 }
```

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.

```
1  int validate_overlap_vm_area(struct pcb_t *caller, int vmaid, int
   vmastart, int vmaend)
2  {
3      struct vm_area_struct *vma = caller->mm->mmap;
4
5      /* TODO validate the planned memory area is not overlapped */
6      while (vma != NULL)
7      {
8          if (vma->vm_id == vmaid)
9          {
10             vma = vma->vm_next;
11             continue;
12          }
13          if (!(vma->vm_end <= vmastart || vma->vm_start >= vmaend))
14          {
15             return -1;
16          }
17          vma = vma->vm_next;
18      }
19      return 0;
20 }
```

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.

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

```
2 {
3     struct vm_rg_struct *newrg = malloc(sizeof(struct vm_rg_struct));
4     int inc_amt = PAGING_PAGE_ALIGNSZ(inc_sz);
5     int incnumpage = inc_amt / PAGING_PAGESZ;
6     struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid)
7     ;
8     int old_sbrk = cur_vma->sbrk;
9
10    struct vm_rg_struct *area = get_vm_area_node_at_brk(caller, vmaid,
11    inc_sz, inc_amt);
12    if (!area || !cur_vma)
13    {
14        cur_vma->sbrk = old_sbrk;
15        return -1;
16    }
17    int old_end = cur_vma->vm_end;
18
19    /*Validate overlap of obtained region */
20    if (validate_overlap_vm_area(caller, vmaid, area->rg_start, area->
21    rg_end) < 0)
22    {
23        cur_vma->sbrk = old_sbrk;
24        free(area);
25        return -1;
26    }
27
28    if (vm_map_ram(caller, area->rg_start, area->rg_end, old_end,
29    incnumpage, newrg) < 0)
30    {
31        cur_vma->sbrk = old_sbrk;
32        free(area);
33        return -1;
34    }
35    cur_vma->vm_end = area->rg_end;
36    enlist_vm_rg_node(&cur_vma->vm_freerg_list, newrg);
37    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`.

```
1 int vmap_page_range(struct pcb_t *caller,
2                     int addr,
3                     int pgnum,
4                     struct framephy_struct *frames,
5                     struct vm_rg_struct *ret_rg)
6 {
7     int pgit = 0;
8     int pgn = PAGING_PGN(addr);
9     struct framephy_struct *fpit = frames;
10    // TODO: update the rg_end and rg_start of ret_rg
11    ret_rg->rg_start = addr;
```

```
12     ret_rg->rg_end = addr + pgnum * PAGING_PAGESZ;
13
14     /* TODO map range of frame to address space
15      *      [addr to addr + pgnum*PAGING_PAGESZ
16      *      in page table caller->mm->pgd[]
17      */
18     for (pgit = 0; pgit < pgnum; pgit++)
19     {
20         if (fpit == NULL)
21         {
22             return -1;
23         }
24         init_pte(&caller->mm->pgd[pgn + pgit], 1, fpit->fpn, 0, 0, 0,
25 0);
26         enlist_pgn_node(&caller->mm->fifo_pgn, pgn + pgit);
27         fpit = fpit->fp_next;
28     }
29     return 0;
30 }
```

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.

```
1 int alloc_pages_range(struct pcb_t *caller, int req_pgnum, struct
   framephy_struct **frm_lst)
2 {
3     int pgit, fpn;
4     struct framephy_struct *newfp_str = NULL;
5     // TODO: allocate the page
6     for (pgit = 0; pgit < req_pgnum; pgit++)
7     {
8         // TODO: allocate the page
9         if (MEMPHY_get_freefp(caller->mram, &fpn) == 0)
10        {
11            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 some 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;
```

```
9     vma0->vm_start = 0;
10    vma0->vm_end = vma0->vm_start;
11    vma0->sbrk = vma0->vm_start;
12    struct vm_rg_struct *first_rg = init_vm_rg(vma0->vm_start, vma0->
vm_end);
13    enlist_vm_rg_node(&vma0->vm_freerg_list, first_rg);
14
15    /* TODO update VMA0 next */
16    vma0->vm_next = NULL;
17
18    /* Point vma owner backward */
19    vma0->vm_mm = mm;
20
21    /* TODO: update mmap */
22    mm->mmap = vma0;
23
24    return 0;
25 }
```

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.

```
1 int MEMPHY_dump(struct memphy_struct *mp)
2 {
3     /*TODO dump memphy content mp->storage
4      *      for tracing the memory content
5      */
6     printf("==== PHYSICAL MEMORY DUMP ====\n");
7     for (int i = 0; i < mp->maxsz; ++i)
8     {
9         if (mp->storage[i] != 0)
10        {
11            printf("BYTE %08x: %d\n", i, mp->storage[i]);
12        }
13    }
14    printf("==== PHYSICAL MEMORY END-DUMP ====\n");
15    printf("
16    =====\n");
17    );
18    return 0;
19 }
```

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_mfq_paging

Output thực thi:

```
1 ld_routine
2   Loaded a process at input/proc/p0s, PID: 1 PRI0: 0
3   CPU 0: Dispatched process 1
4 ===== PHYSICAL MEMORY AFTER ALLOCATION =====
```



```
5 PID=1 - Region=0 - Address=00000000 - Size=300 byte
6 print_pgtbl: 0 - 512
7 00000000: 80000001
8 00000004: 80000000
9 Page Number: 0 -> Frame Number: 1
10 Page Number: 1 -> Frame Number: 0
11 =====
12 Loaded a process at input/proc/p1s, PID: 2 PRI0: 15
13 CPU 1: Dispatched process 2
14 ===== PHYSICAL MEMORY AFTER ALLOCATION =====
15 PID=1 - Region=4 - Address=00000200 - Size=300 byte
16 print_pgtbl: 0 - 1024
17 00000000: 80000001
18 00000004: 80000000
19 00000008: 80000003
20 00000012: 80000002
21 Page Number: 0 -> Frame Number: 1
22 Page Number: 1 -> Frame Number: 0
23 Page Number: 2 -> Frame Number: 3
24 Page Number: 3 -> Frame Number: 2
25 =====
26 ===== PHYSICAL MEMORY AFTER DEALLOCATION =====
27 PID=1 - Region=0
28 print_pgtbl: 0 - 1024
29 00000000: 80000001
30 00000004: 80000000
31 00000008: 80000003
32 00000012: 80000002
33 Page Number: 0 -> Frame Number: 1
34 Page Number: 1 -> Frame Number: 0
35 Page Number: 2 -> Frame Number: 3
36 Page Number: 3 -> Frame Number: 2
37 =====
38 ===== PHYSICAL MEMORY AFTER ALLOCATION =====
39 PID=1 - Region=1 - Address=00000000 - Size=100 byte
40 print_pgtbl: 0 - 1024
```



```
41 00000000: 80000001
42 00000004: 80000000
43 00000008: 80000003
44 00000012: 80000002
45 Page Number: 0 -> Frame Number: 1
46 Page Number: 1 -> Frame Number: 0
47 Page Number: 2 -> Frame Number: 3
48 Page Number: 3 -> Frame Number: 2
49 =====
50   Loaded a process at input/proc/p1s, PID: 3 PRIO: 0
51 ===== PHYSICAL MEMORY AFTER WRITING =====
52 write region=1 offset=20 value=100
53 print_pgtbl: 0 - 1024
54 00000000: 80000001
55 00000004: 80000000
56 00000008: 80000003
57 00000012: 80000002
58 Page Number: 0 -> Frame Number: 1
59 Page Number: 1 -> Frame Number: 0
60 Page Number: 2 -> Frame Number: 3
61 Page Number: 3 -> Frame Number: 2
62 =====
63 ===== PHYSICAL MEMORY DUMP =====
64 BYTE 00000114: 100
65 ===== PHYSICAL MEMORY END-DUMP =====
66 =====
67   Loaded a process at input/proc/p1s, PID: 4 PRIO: 0
68   CPU 0: Put process 1 to run queue
69   CPU 0: Dispatched process 3
70   CPU 1: Put process 2 to run queue
71   CPU 1: Dispatched process 4
72   CPU 0: Put process 3 to run queue
73   CPU 0: Dispatched process 1
74 ===== PHYSICAL MEMORY AFTER READING =====
75 read region=1 offset=20 value=100
76 print_pgtbl: 0 - 1024
```



```
77 00000000: 80000001
78 00000004: 80000000
79 00000008: 80000003
80 00000012: 80000002
81 Page Number: 0 -> Frame Number: 1
82 Page Number: 1 -> Frame Number: 0
83 Page Number: 2 -> Frame Number: 3
84 Page Number: 3 -> Frame Number: 2
85 =====
86 ===== PHYSICAL MEMORY DUMP =====
87 BYTE 00000114: 100
88 ===== PHYSICAL MEMORY END-DUMP =====
89 =====
90 ===== PHYSICAL MEMORY AFTER WRITING =====
91 write region=2 offset=20 value=102
92 print_pgtbl: 0 - 1024
93 00000000: 80000001
94 00000004: 80000000
95 00000008: 80000003
96 00000012: 80000002
97 Page Number: 0 -> Frame Number: 1
98 Page Number: 1 -> Frame Number: 0
99 Page Number: 2 -> Frame Number: 3
100 Page Number: 3 -> Frame Number: 2
101 =====
102 ===== PHYSICAL MEMORY DUMP =====
103 BYTE 00000114: 102
104 ===== PHYSICAL MEMORY END-DUMP =====
105 =====
106 CPU 1: Put process 4 to run queue
107 CPU 1: Dispatched process 3
108 ===== PHYSICAL MEMORY AFTER READING =====
109 read region=2 offset=20 value=102
110 print_pgtbl: 0 - 1024
111 00000000: 80000001
112 00000004: 80000000
```



```
113 00000008: 80000003
114 00000012: 80000002
115 Page Number: 0 -> Frame Number: 1
116 Page Number: 1 -> Frame Number: 0
117 Page Number: 2 -> Frame Number: 3
118 Page Number: 3 -> Frame Number: 2
119 =====
120 ===== PHYSICAL MEMORY DUMP =====
121 BYTE 00000114: 102
122 ===== PHYSICAL MEMORY END-DUMP =====
123 =====
124 ===== PHYSICAL MEMORY AFTER WRITING =====
125 write region=3 offset=20 value=103
126 print_pgtbl: 0 - 1024
127 00000000: 80000001
128 00000004: 80000000
129 00000008: 80000003
130 00000012: 80000002
131 Page Number: 0 -> Frame Number: 1
132 Page Number: 1 -> Frame Number: 0
133 Page Number: 2 -> Frame Number: 3
134 Page Number: 3 -> Frame Number: 2
135 =====
136 ===== PHYSICAL MEMORY DUMP =====
137 BYTE 00000114: 103
138 ===== PHYSICAL MEMORY END-DUMP =====
139 =====
140 ===== PHYSICAL MEMORY AFTER READING =====
141 read region=3 offset=20 value=103
142 print_pgtbl: 0 - 1024
143 00000000: 80000001
144 00000004: 80000000
145 00000008: 80000003
146 00000012: 80000002
147 Page Number: 0 -> Frame Number: 1
148 Page Number: 1 -> Frame Number: 0
```



```
149 Page Number: 2 -> Frame Number: 3
150 Page Number: 3 -> Frame Number: 2
151 =====
152 ===== PHYSICAL MEMORY DUMP =====
153 BYTE 00000114: 103
154 ===== PHYSICAL MEMORY END-DUMP =====
155 =====
156 CPU 1: Processed 3 has finished
157 CPU 1: Dispatched process 4
158 CPU 0: Put process 1 to run queue
159 CPU 0: Dispatched process 1
160 ===== PHYSICAL MEMORY AFTER DEALLOCATION =====
161 PID=1 - Region=4
162 print_pgtbl: 0 - 1024
163 00000000: 80000001
164 00000004: 80000000
165 00000008: 80000003
166 00000012: 80000002
167 Page Number: 0 -> Frame Number: 1
168 Page Number: 1 -> Frame Number: 0
169 Page Number: 2 -> Frame Number: 3
170 Page Number: 3 -> Frame Number: 2
171 =====
172 CPU 0: Processed 1 has finished
173 CPU 0: Dispatched process 2
174 CPU 1: Processed 4 has finished
175 CPU 1 stopped
176 CPU 0: Processed 2 has finished
177 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 $0x100 + 0x14 = 0x114 \rightarrow \text{BYTE } 0x114 = 100$.
- 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		calc			calc	
calc			calc			calc	
free 4						calc	
calc						calc	
						calc	

Bảng 6: Các tiến trình của os_1_mlq_paging_small_1K

Output thực thi:

```

1   Time slot    0
2 ld_routine
3 Time slot    1
4   Loaded a process at input/proc/p0s, PID: 1 PRI0: 130
5   CPU 3: Dispatched process 1
6 Time slot    2
7 ===== PHYSICAL MEMORY AFTER ALLOCATION =====
8 PID=1 - Region=0 - Address=00000000 - Size=300 byte
9 print_pgtbl: 0 - 512
10 00000000: 80000001
11 00000004: 80000000
12 Page Number: 0 -> Frame Number: 1
13 Page Number: 1 -> Frame Number: 0
14 =====
15   Loaded a process at input/proc/s3, PID: 2 PRI0: 39
16 Time slot    3
17   CPU 3: Put process 1 to run queue
18   CPU 3: Dispatched process 2
19   CPU 1: Dispatched process 1
20 ===== PHYSICAL MEMORY AFTER ALLOCATION =====
21 PID=1 - Region=4 - Address=00000200 - Size=300 byte
22 print_pgtbl: 0 - 1024
23 00000000: 80000001
24 00000004: 80000000
25 00000008: 80000003
26 00000012: 80000002

```



```
27 Page Number: 0 -> Frame Number: 1
28 Page Number: 1 -> Frame Number: 0
29 Page Number: 2 -> Frame Number: 3
30 Page Number: 3 -> Frame Number: 2
31 =====
32 Time slot    4
33   Loaded a process at input/proc/m1s, PID: 3 PRI0: 15
34   CPU 2: Dispatched process 3
35 ===== PHYSICAL MEMORY AFTER ALLOCATION =====
36 PID=3 - Region=0 - Address=00000000 - Size=300 byte
37 print_pgtbl: 0 - 512
38 00000000: 80000005
39 00000004: 80000004
40 Page Number: 0 -> Frame Number: 5
41 Page Number: 1 -> Frame Number: 4
42 =====
43 ===== PHYSICAL MEMORY AFTER DEALLOCATION =====
44 PID=1 - Region=0
45 print_pgtbl: 0 - 1024
46 00000000: 80000001
47 00000004: 80000000
48 00000008: 80000003
49 00000012: 80000002
50 Page Number: 0 -> Frame Number: 1
51 Page Number: 1 -> Frame Number: 0
52 Page Number: 2 -> Frame Number: 3
53 Page Number: 3 -> Frame Number: 2
54 =====
55 Time slot    5
56   CPU 3: Put process 2 to run queue
57   CPU 3: Dispatched process 2
58 ===== PHYSICAL MEMORY AFTER ALLOCATION =====
59 PID=3 - Region=1 - Address=0000012c - Size=100 byte
60   CPU 1: Put process 1 to run queue
61   CPU 1: Dispatched process 1
62 print_pgtbl: 0 - 512
```



```
63 00000000: 80000005
64 00000004: 80000004
65 Page Number: 0 -> Frame Number: 5
66 Page Number: 1 -> Frame Number: 4
67 =====
68 ===== PHYSICAL MEMORY AFTER ALLOCATION =====
69 PID=1 - Region=1 - Address=00000000 - Size=100 byte
70 print_pgtbl: 0 - 1024
71 00000000: 80000001
72 00000004: 80000000
73 00000008: 80000003
74 00000012: 80000002
75 Page Number: 0 -> Frame Number: 1
76 Page Number: 1 -> Frame Number: 0
77 Page Number: 2 -> Frame Number: 3
78 Page Number: 3 -> Frame Number: 2
79 =====
80 Time slot 6
81 Loaded a process at input/proc/s2, PID: 4 PRI0: 120
82 ===== PHYSICAL MEMORY AFTER WRITING =====
83 write region=1 offset=20 value=100
84 print_pgtbl: 0 - 1024
85 00000000: 80000001
86 00000004: 80000000
87 00000008: 80000003
88 00000012: 80000002
89 Page Number: 0 -> Frame Number: 1
90 Page Number: 1 -> Frame Number: 0
91 Page Number: 2 -> Frame Number: 3
92 Page Number: 3 -> Frame Number: 2
93 =====
94 ===== PHYSICAL MEMORY DUMP =====
95 BYTE 00000114: 100
96 ===== PHYSICAL MEMORY END-DUMP =====
97 =====
98 CPU 2: Put process 3 to run queue
```

```
99   CPU 2: Dispatched process  3
100  ===== PHYSICAL MEMORY AFTER DEALLOCATION =====
101  PID=3 - Region=0
102  print_pgtbl: 0 - 512
103  00000000: 80000005
104  00000004: 80000004
105  Page Number: 0 -> Frame Number: 5
106  Page Number: 1 -> Frame Number: 4
107  =====
108   CPU 0: Dispatched process  4
109  Time slot    7
110   Loaded a process at input/proc/m0s, PID: 5 PRI0: 120
111  ===== PHYSICAL MEMORY AFTER ALLOCATION =====
112  PID=3 - Region=2 - Address=00000000 - Size=100 byte
113  print_pgtbl: 0 - 512
114  00000000: 80000005
115  00000004: 80000004
116  Page Number: 0 -> Frame Number: 5
117  Page Number: 1 -> Frame Number: 4
118  =====
119   CPU 3: Put process  2 to run queue
120   CPU 3: Dispatched process  2
121   CPU 1: Put process  1 to run queue
122   CPU 1: Dispatched process  5
123  ===== PHYSICAL MEMORY AFTER ALLOCATION =====
124  PID=5 - Region=0 - Address=00000000 - Size=300 byte
125  print_pgtbl: 0 - 512
126  00000000: 80000007
127  00000004: 80000006
128  Page Number: 0 -> Frame Number: 7
129  Page Number: 1 -> Frame Number: 6
130  =====
131  Time slot    8
132   CPU 2: Put process  3 to run queue
133   CPU 2: Dispatched process  3
134  ===== PHYSICAL MEMORY AFTER DEALLOCATION =====
```

```
135 PID=3 - Region=2
136 print_pgtbl: 0 - 512
137 00000000: 80000005
138 00000004: 80000004
139 Page Number: 0 -> Frame Number: 5
140 Page Number: 1 -> Frame Number: 4
141 =====
142 CPU 0: Put process 4 to run queue
143 CPU 0: Dispatched process 4
144 ===== PHYSICAL MEMORY AFTER ALLOCATION =====
145 PID=5 - Region=1 - Address=0000012c - Size=100 byte
146 print_pgtbl: 0 - 512
147 00000000: 80000007
148 00000004: 80000006
149 Page Number: 0 -> Frame Number: 7
150 Page Number: 1 -> Frame Number: 6
151 =====
152 Time slot 9
153 Loaded a process at input/proc/pls, PID: 6 PRIO: 15
154 CPU 3: Put process 2 to run queue
155 CPU 3: Dispatched process 6
156 CPU 1: Put process 5 to run queue
157 CPU 1: Dispatched process 2
158 ===== PHYSICAL MEMORY AFTER DEALLOCATION =====
159 PID=3 - Region=1
160 print_pgtbl: 0 - 512
161 00000000: 80000005
162 00000004: 80000004
163 Page Number: 0 -> Frame Number: 5
164 Page Number: 1 -> Frame Number: 4
165 =====
166 Time slot 10
167 CPU 2: Processed 3 has finished
168 CPU 2: Dispatched process 5
169 ===== PHYSICAL MEMORY AFTER DEALLOCATION =====
170 PID=5 - Region=0
```




```
171 print_pgtbl: 0 - 512
172 00000000: 80000007
173 00000004: 80000006
174 Page Number: 0 -> Frame Number: 7
175 Page Number: 1 -> Frame Number: 6
176 =====
177 CPU 0: Put process 4 to run queue
178 CPU 0: Dispatched process 4
179 Time slot 11
180 CPU 3: Put process 6 to run queue
181 CPU 3: Dispatched process 6
182 CPU 1: Put process 2 to run queue
183 CPU 1: Dispatched process 2
184 ===== PHYSICAL MEMORY AFTER ALLOCATION =====
185 PID=5 - Region=2 - Address=00000000 - Size=100 byte
186 print_pgtbl: 0 - 512
187 00000000: 80000007
188 00000004: 80000006
189 Page Number: 0 -> Frame Number: 7
190 Page Number: 1 -> Frame Number: 6
191 =====
192 Loaded a process at input/proc/s0, PID: 7 PRI0: 38
193 Time slot 12
194 CPU 2: Put process 5 to run queue
195 CPU 2: Dispatched process 7
196 CPU 0: Put process 4 to run queue
197 CPU 0: Dispatched process 5
198 ===== PHYSICAL MEMORY AFTER WRITING =====
199 write region=1 offset=20 value=102
200 print_pgtbl: 0 - 512
201 00000000: 80000007
202 00000004: 80000006
203 Page Number: 0 -> Frame Number: 7
204 Page Number: 1 -> Frame Number: 6
205 =====
206 ===== PHYSICAL MEMORY DUMP =====
```



```
207 BYTE 00000114: 100
208 BYTE 00000640: 102
209 ===== PHYSICAL MEMORY END-DUMP =====
210 =====
211 Time slot 13
212 CPU 3: Put process 6 to run queue
213 CPU 3: Dispatched process 6
214 CPU 1: Put process 2 to run queue
215 CPU 1: Dispatched process 2
216 ===== PHYSICAL MEMORY AFTER WRITING =====
217 write region=2 offset=1000 value=1
218 print_pgtbl: 0 - 512
219 00000000: c0000000
220 00000004: 80000006
221 Page Number: 0 -> Frame Number: 0
222 Page Number: 1 -> Frame Number: 6
223 =====
224 ===== PHYSICAL MEMORY DUMP =====
225 BYTE 00000114: 100
226 BYTE 00000640: 102
227 BYTE 000007e8: 1
228 ===== PHYSICAL MEMORY END-DUMP =====
229 =====
230 Time slot 14
231 CPU 1: Processed 2 has finished
232 CPU 1: Dispatched process 4
233 CPU 0: Processed 5 has finished
234 CPU 0: Dispatched process 1
235 ===== PHYSICAL MEMORY AFTER READING =====
236 read region=1 offset=20 value=100
237 print_pgtbl: 0 - 1024
238 00000000: 80000001
239 00000004: 80000000
240 00000008: 80000003
241 00000012: 80000002
242 Page Number: 0 -> Frame Number: 1
```



```
243 Page Number: 1 -> Frame Number: 0
244 Page Number: 2 -> Frame Number: 3
245 Page Number: 3 -> Frame Number: 2
246 =====
247 ===== PHYSICAL MEMORY DUMP =====
248 BYTE 00000114: 100
249 BYTE 00000640: 102
250 BYTE 000007e8: 1
251 ===== PHYSICAL MEMORY END-DUMP =====
252 =====
253 CPU 2: Put process 7 to run queue
254 CPU 2: Dispatched process 7
255 Time slot 15
256 CPU 3: Put process 6 to run queue
257 CPU 3: Dispatched process 6
258 ===== PHYSICAL MEMORY AFTER WRITING =====
259 write region=2 offset=20 value=102
260 print_pgtbl: 0 - 1024
261 00000000: 80000001
262 00000004: 80000000
263 00000008: 80000003
264 00000012: 80000002
265 Page Number: 0 -> Frame Number: 1
266 Page Number: 1 -> Frame Number: 0
267 Page Number: 2 -> Frame Number: 3
268 Page Number: 3 -> Frame Number: 2
269 =====
270 ===== PHYSICAL MEMORY DUMP =====
271 BYTE 00000114: 102
272 BYTE 00000640: 102
273 BYTE 000007e8: 1
274 ===== PHYSICAL MEMORY END-DUMP =====
275 =====
276 Time slot 16
277 Loaded a process at input/proc/s1, PID: 8 PRI0: 0
278 CPU 2: Put process 7 to run queue
```



```
279 CPU 2: Dispatched process 8
280 CPU 1: Put process 4 to run queue
281 CPU 1: Dispatched process 7
282 CPU 0: Put process 1 to run queue
283 CPU 0: Dispatched process 4
284 Time slot 17
285 CPU 3: Put process 6 to run queue
286 CPU 3: Dispatched process 6
287 Time slot 18
288 CPU 1: Put process 7 to run queue
289 CPU 1: Dispatched process 7
290 CPU 0: Put process 4 to run queue
291 CPU 0: Dispatched process 4
292 CPU 2: Put process 8 to run queue
293 CPU 2: Dispatched process 8
294 Time slot 19
295 CPU 3: Processed 6 has finished
296 CPU 3: Dispatched process 1
297 ===== PHYSICAL MEMORY AFTER READING =====
298 read region=2 offset=20 value=102
299 print_pgtbl: 0 - 1024
300 00000000: 80000001
301 00000004: 80000000
302 00000008: 80000003
303 00000012: 80000002
304 Page Number: 0 -> Frame Number: 1
305 Page Number: 1 -> Frame Number: 0
306 Page Number: 2 -> Frame Number: 3
307 Page Number: 3 -> Frame Number: 2
308 =====
309 ===== PHYSICAL MEMORY DUMP =====
310 BYTE 00000114: 102
311 BYTE 00000640: 102
312 BYTE 000007e8: 1
313 ===== PHYSICAL MEMORY END-DUMP =====
314 =====
```



```
315 Time slot 20
316 CPU 1: Put process 7 to run queue
317 CPU 1: Dispatched process 7
318 ===== PHYSICAL MEMORY AFTER WRITING =====
319 write region=3 offset=20 value=103
320 print_pgtbl: 0 - 1024
321 00000000: 80000001
322 00000004: 80000000
323 00000008: 80000003
324 00000012: 80000002
325 Page Number: 0 -> Frame Number: 1
326 Page Number: 1 -> Frame Number: 0
327 Page Number: 2 -> Frame Number: 3
328 Page Number: 3 -> Frame Number: 2
329 =====
330 ===== PHYSICAL MEMORY DUMP =====
331 BYTE 00000114: 103
332 BYTE 00000640: 102
333 BYTE 000007e8: 1
334 ===== PHYSICAL MEMORY END-DUMP =====
335 =====
336 CPU 0: Processed 4 has finished
337 CPU 0 stopped
338 CPU 2: Put process 8 to run queue
339 CPU 2: Dispatched process 8
340 Time slot 21
341 CPU 3: Put process 1 to run queue
342 CPU 3: Dispatched process 1
343 ===== PHYSICAL MEMORY AFTER READING =====
344 read region=3 offset=20 value=103
345 print_pgtbl: 0 - 1024
346 00000000: 80000001
347 00000004: 80000000
348 00000008: 80000003
349 00000012: 80000002
350 Page Number: 0 -> Frame Number: 1
```



```
351 Page Number: 1 -> Frame Number: 0
352 Page Number: 2 -> Frame Number: 3
353 Page Number: 3 -> Frame Number: 2
354 =====
355 ===== PHYSICAL MEMORY DUMP =====
356 BYTE 00000114: 103
357 BYTE 00000640: 102
358 BYTE 000007e8: 1
359 ===== PHYSICAL MEMORY END-DUMP =====
360 =====
361 Time slot 22
362   CPU 1: Put process 7 to run queue
363   CPU 1: Dispatched process 7
364   CPU 2: Put process 8 to run queue
365   CPU 2: Dispatched process 8
366 Time slot 23
367   CPU 3: Put process 1 to run queue
368   CPU 3: Dispatched process 1
369 ===== PHYSICAL MEMORY AFTER DEALLOCATION =====
370 PID=1 - Region=4
371 print_pgtbl: 0 - 1024
372 00000000: 80000001
373 00000004: 80000000
374 00000008: 80000003
375 00000012: 80000002
376 Page Number: 0 -> Frame Number: 1
377 Page Number: 1 -> Frame Number: 0
378 Page Number: 2 -> Frame Number: 3
379 Page Number: 3 -> Frame Number: 2
380 =====
381   CPU 2: Processed 8 has finished
382   CPU 2 stopped
383 Time slot 24
384   CPU 1: Put process 7 to run queue
385   CPU 1: Dispatched process 7
386 Time slot 25
```

```
387 CPU 3: Processed 1 has finished
388 CPU 3 stopped
389 Time slot 26
390 CPU 1: Put process 7 to run queue
391 CPU 1: Dispatched process 7
392 Time slot 27
393 CPU 1: Processed 7 has finished
394 CPU 1 stopped
395
```

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 $6 * 256 = 1536 = 0x600$) \rightarrow địa chỉ vật lý là $0x600 + 0x40 = 0x640 \rightarrow \text{BYTE } 0x640 = 102$

- 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

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:

```

1      Time slot    0
2  ld_routine
3  Time slot    1
4      Loaded a process at input/proc/s4, PID: 1 PRI0: 4
5  Time slot    2
6      Loaded a process at input/proc/s3, PID: 2 PRI0: 3

```

```
7   CPU 0: Dispatched process  2
8   Time slot  3
9   Time slot  4
10  CPU 0: Put process  2 to run queue
11  CPU 0: Dispatched process  2
12  Loaded a process at input/proc/m1s, PID: 3 PRI0: 2
13  Time slot  5
14  Time slot  6
15  Loaded a process at input/proc/s2, PID: 4 PRI0: 3
16  CPU 0: Put process  2 to run queue
17  CPU 0: Dispatched process  3
18  ===== PHYSICAL MEMORY AFTER ALLOCATION =====
19  PID=3 - Region=0 - Address=00000000 - Size=300 byte
20  print_pgtbl: 0 - 512
21  00000000: 80000001
22  00000004: 80000000
23  Page Number: 0 -> Frame Number: 1
24  Page Number: 1 -> Frame Number: 0
25  =====
26  Time slot  7
27  ===== PHYSICAL MEMORY AFTER ALLOCATION =====
28  PID=3 - Region=1 - Address=0000012c - Size=100 byte
29  print_pgtbl: 0 - 512
30  00000000: 80000001
31  00000004: 80000000
32  Page Number: 0 -> Frame Number: 1
33  Page Number: 1 -> Frame Number: 0
34  =====
35  Loaded a process at input/proc/m0s, PID: 5 PRI0: 3
36  Time slot  8
37  CPU 0: Put process  3 to run queue
38  CPU 0: Dispatched process  3
39  ===== PHYSICAL MEMORY AFTER DEALLOCATION =====
40  PID=3 - Region=0
41  print_pgtbl: 0 - 512
42  00000000: 80000001
```



```
43 00000004: 80000000
44 Page Number: 0 -> Frame Number: 1
45 Page Number: 1 -> Frame Number: 0
46 =====
47 Time slot 9
48   Loaded a process at input/proc/p1s, PID: 6 PRI0: 2
49 ===== PHYSICAL MEMORY AFTER ALLOCATION =====
50 PID=3 - Region=2 - Address=00000000 - Size=100 byte
51 print_pgtbl: 0 - 512
52 00000000: 80000001
53 00000004: 80000000
54 Page Number: 0 -> Frame Number: 1
55 Page Number: 1 -> Frame Number: 0
56 =====
57 Time slot 10
58   CPU 0: Put process 3 to run queue
59   CPU 0: Dispatched process 6
60 Time slot 11
61   Loaded a process at input/proc/s0, PID: 7 PRI0: 1
62 Time slot 12
63   CPU 0: Put process 6 to run queue
64   CPU 0: Dispatched process 7
65 Time slot 13
66 Time slot 14
67   CPU 0: Put process 7 to run queue
68   CPU 0: Dispatched process 7
69 Time slot 15
70 Time slot 16
71   Loaded a process at input/proc/s1, PID: 8 PRI0: 0
72   CPU 0: Put process 7 to run queue
73   CPU 0: Dispatched process 8
74 Time slot 17
75 Time slot 18
76   CPU 0: Put process 8 to run queue
77   CPU 0: Dispatched process 8
78 Time slot 19
```

```
79 Time slot 20
80 CPU 0: Put process 8 to run queue
81 CPU 0: Dispatched process 8
82 Time slot 21
83 Time slot 22
84 CPU 0: Put process 8 to run queue
85 CPU 0: Dispatched process 8
86 Time slot 23
87 CPU 0: Processed 8 has finished
88 CPU 0: Dispatched process 7
89 Time slot 24
90 Time slot 25
91 CPU 0: Put process 7 to run queue
92 CPU 0: Dispatched process 7
93 Time slot 26
94 Time slot 27
95 CPU 0: Put process 7 to run queue
96 CPU 0: Dispatched process 7
97 Time slot 28
98 Time slot 29
99 CPU 0: Put process 7 to run queue
100 CPU 0: Dispatched process 7
101 Time slot 30
102 Time slot 31
103 CPU 0: Put process 7 to run queue
104 CPU 0: Dispatched process 7
105 Time slot 32
106 Time slot 33
107 CPU 0: Put process 7 to run queue
108 CPU 0: Dispatched process 7
109 Time slot 34
110 CPU 0: Processed 7 has finished
111 CPU 0: Dispatched process 3
112 ===== PHYSICAL MEMORY AFTER DEALLOCATION =====
113 PID=3 - Region=2
114 print_pgtbl: 0 - 512
```



```
115 00000000: 80000001
116 00000004: 80000000
117 Page Number: 0 -> Frame Number: 1
118 Page Number: 1 -> Frame Number: 0
119 =====
120 Time slot 35
121 ===== PHYSICAL MEMORY AFTER DEALLOCATION =====
122 PID=3 - Region=1
123 print_pgtbl: 0 - 512
124 00000000: 80000001
125 00000004: 80000000
126 Page Number: 0 -> Frame Number: 1
127 Page Number: 1 -> Frame Number: 0
128 =====
129 Time slot 36
130 CPU 0: Processed 3 has finished
131 CPU 0: Dispatched process 6
132 Time slot 37
133 Time slot 38
134 CPU 0: Put process 6 to run queue
135 CPU 0: Dispatched process 6
136 Time slot 39
137 Time slot 40
138 CPU 0: Put process 6 to run queue
139 CPU 0: Dispatched process 6
140 Time slot 41
141 Time slot 42
142 CPU 0: Put process 6 to run queue
143 CPU 0: Dispatched process 6
144 Time slot 43
145 Time slot 44
146 CPU 0: Processed 6 has finished
147 CPU 0: Dispatched process 5
148 ===== PHYSICAL MEMORY AFTER ALLOCATION =====
149 PID=5 - Region=0 - Address=00000000 - Size=300 byte
150 print_pgtbl: 0 - 512
```



```
151 00000000: 80000003
152 00000004: 80000002
153 Page Number: 0 -> Frame Number: 3
154 Page Number: 1 -> Frame Number: 2
155 =====
156 Time slot 45
157 ===== PHYSICAL MEMORY AFTER ALLOCATION =====
158 PID=5 - Region=1 - Address=0000012c - Size=100 byte
159 print_pgtbl: 0 - 512
160 00000000: 80000003
161 00000004: 80000002
162 Page Number: 0 -> Frame Number: 3
163 Page Number: 1 -> Frame Number: 2
164 =====
165 Time slot 46
166 CPU 0: Put process 5 to run queue
167 CPU 0: Dispatched process 5
168 ===== PHYSICAL MEMORY AFTER DEALLOCATION =====
169 PID=5 - Region=0
170 print_pgtbl: 0 - 512
171 00000000: 80000003
172 00000004: 80000002
173 Page Number: 0 -> Frame Number: 3
174 Page Number: 1 -> Frame Number: 2
175 =====
176 Time slot 47
177 ===== PHYSICAL MEMORY AFTER ALLOCATION =====
178 PID=5 - Region=2 - Address=00000000 - Size=100 byte
179 print_pgtbl: 0 - 512
180 00000000: 80000003
181 00000004: 80000002
182 Page Number: 0 -> Frame Number: 3
183 Page Number: 1 -> Frame Number: 2
184 =====
185 Time slot 48
186 CPU 0: Put process 5 to run queue
```



```
187 CPU 0: Dispatched process 5
188 ===== PHYSICAL MEMORY AFTER WRITING =====
189 write region=1 offset=20 value=102
190 print_pgtbl: 0 - 512
191 00000000: 80000003
192 00000004: 80000002
193 Page Number: 0 -> Frame Number: 3
194 Page Number: 1 -> Frame Number: 2
195 =====
196 ===== PHYSICAL MEMORY DUMP =====
197 BYTE 00000240: 102
198 ===== PHYSICAL MEMORY END-DUMP =====
199 =====
200 Time slot 49
201 ===== PHYSICAL MEMORY AFTER WRITING =====
202 write region=2 offset=1000 value=1
203 print_pgtbl: 0 - 512
204 00000000: c0000000
205 00000004: 80000002
206 Page Number: 0 -> Frame Number: 0
207 Page Number: 1 -> Frame Number: 2
208 =====
209 ===== PHYSICAL MEMORY DUMP =====
210 BYTE 00000240: 102
211 BYTE 000003e8: 1
212 ===== PHYSICAL MEMORY END-DUMP =====
213 =====
214 Time slot 50
215 CPU 0: Processed 5 has finished
216 CPU 0: Dispatched process 2
217 Time slot 51
218 Time slot 52
219 CPU 0: Put process 2 to run queue
220 CPU 0: Dispatched process 2
221 Time slot 53
222 Time slot 54
```




```
223 CPU 0: Put process 2 to run queue
224 CPU 0: Dispatched process 2
225 Time slot 55
226 Time slot 56
227 CPU 0: Put process 2 to run queue
228 CPU 0: Dispatched process 2
229 Time slot 57
230 CPU 0: Processed 2 has finished
231 CPU 0: Dispatched process 4
232 Time slot 58
233 Time slot 59
234 CPU 0: Put process 4 to run queue
235 CPU 0: Dispatched process 4
236 Time slot 60
237 Time slot 61
238 CPU 0: Put process 4 to run queue
239 CPU 0: Dispatched process 4
240 Time slot 62
241 Time slot 63
242 CPU 0: Put process 4 to run queue
243 CPU 0: Dispatched process 4
244 Time slot 64
245 Time slot 65
246 CPU 0: Put process 4 to run queue
247 CPU 0: Dispatched process 4
248 Time slot 66
249 Time slot 67
250 CPU 0: Put process 4 to run queue
251 CPU 0: Dispatched process 4
252 Time slot 68
253 Time slot 69
254 CPU 0: Processed 4 has finished
255 CPU 0: Dispatched process 1
256 Time slot 70
257 Time slot 71
258 CPU 0: Put process 1 to run queue
```

```
259 CPU 0: Dispatched process 1
260 Time slot 72
261 Time slot 73
262 CPU 0: Put process 1 to run queue
263 CPU 0: Dispatched process 1
264 Time slot 74
265 Time slot 75
266 CPU 0: Put process 1 to run queue
267 CPU 0: Dispatched process 1
268 Time slot 76
269 CPU 0: Processed 1 has finished
270 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.

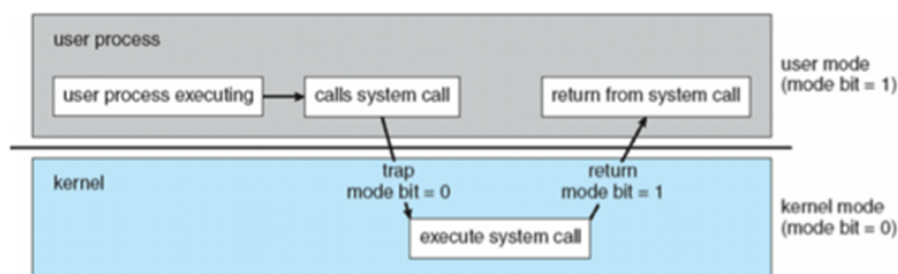
4 System Call

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)
```



```
3 {
4     for (int i = 0; i < PAGING_MAX_SYMTBL_SZ; i++)
5     {
6         if (proc->mm->symrgtbl[i].rg_start != 0 || proc->mm->symrgtbl[
7         i].rg_end != 0)
8         {
9             libfree(proc, i);
10        }
11    }
12
13    int __sys_killall(struct pcb_t *caller, struct sc_regs *regs)
14    {
15        char proc_name[100];
16        uint32_t raw;
17        BYTE ch;
18        uint32_t memrg = regs->a1;
19
20        memset(proc_name, 0, sizeof(proc_name));
21        int idx;
22        for (idx = 0; idx < (int)sizeof(proc_name) - 1; idx++)
23        {
24            if (libread(caller, memrg, idx, &raw) < 0)
25            {
26                return -1;
27            }
28            ch = (BYTE)(raw & 0xFF);
29            if (ch == '\\0')
30            {
31                break;
32            }
33            if (!isprint(ch))
34            {
35                break;
36            }
37            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:

```
1 ld_routine
2   Loaded a process at input/proc/sc2, PID: 1 PRI0: 15
3   CPU 0: Dispatched process 1
4   ===== PHYSICAL MEMORY AFTER ALLOCATION =====
5   PID=1 - Region=1 - Address=00000000 - Size=100 byte
6   print_pgtbl: 0 - 256
7   00000000: 80000000
8   Page Number: 0 -> Frame Number: 0
9   =====
10  ===== PHYSICAL MEMORY AFTER WRITING =====
```

```
11 write region=1 offset=0 value=80
12 print_pgtbl: 0 - 256
13 00000000: 80000000
14 Page Number: 0 -> Frame Number: 0
15 =====
16 ===== PHYSICAL MEMORY DUMP =====
17 BYTE 00000000: 80
18 ===== PHYSICAL MEMORY END-DUMP =====
19 =====
20 CPU 0: Put process 1 to run queue
21 CPU 0: Dispatched process 1
22 ===== PHYSICAL MEMORY AFTER WRITING =====
23 write region=1 offset=1 value=48
24 print_pgtbl: 0 - 256
25 00000000: 80000000
26 Page Number: 0 -> Frame Number: 0
27 =====
28 ===== PHYSICAL MEMORY DUMP =====
29 BYTE 00000000: 80
30 BYTE 00000001: 48
31 ===== PHYSICAL MEMORY END-DUMP =====
32 =====
33 ===== PHYSICAL MEMORY AFTER WRITING =====
34 write region=1 offset=2 value=-1
35 print_pgtbl: 0 - 256
36 00000000: 80000000
37 Page Number: 0 -> Frame Number: 0
38 =====
39 ===== PHYSICAL MEMORY DUMP =====
40 BYTE 00000000: 80
41 BYTE 00000001: 48
42 BYTE 00000002: -1
43 ===== PHYSICAL MEMORY END-DUMP =====
44 =====
45 CPU 0: Put process 1 to run queue
46 CPU 0: Dispatched process 1
```



```
47 ===== PHYSICAL MEMORY AFTER READING =====
48 read region=1 offset=0 value=80
49 print_pgtbl: 0 - 256
50 00000000: 80000000
51 Page Number: 0 -> Frame Number: 0
52 =====
53 ===== PHYSICAL MEMORY DUMP =====
54 BYTE 00000000: 80
55 BYTE 00000001: 48
56 BYTE 00000002: -1
57 ===== PHYSICAL MEMORY END-DUMP =====
58 =====
59 ===== PHYSICAL MEMORY AFTER READING =====
60 read region=1 offset=1 value=48
61 print_pgtbl: 0 - 256
62 00000000: 80000000
63 Page Number: 0 -> Frame Number: 0
64 =====
65 ===== PHYSICAL MEMORY DUMP =====
66 BYTE 00000000: 80
67 BYTE 00000001: 48
68 BYTE 00000002: -1
69 ===== PHYSICAL MEMORY END-DUMP =====
70 =====
71 ===== PHYSICAL MEMORY AFTER READING =====
72 read region=1 offset=2 value=-1
73 print_pgtbl: 0 - 256
74 00000000: 80000000
75 Page Number: 0 -> Frame Number: 0
76 =====
77 ===== PHYSICAL MEMORY DUMP =====
78 BYTE 00000000: 80
79 BYTE 00000001: 48
80 BYTE 00000002: -1
81 ===== PHYSICAL MEMORY END-DUMP =====
82 =====
```

```
83 The procname retrieved from memregionid 1 is "P0"  
84 CPU 0: Processed 1 has finished  
85 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 lượt là 80, 48, và -1 với 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 lượt 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:

```
1 ld_routine
2     Loaded a process at input/proc/sc1, PID: 1 PRIO: 15
3     CPU 0: Dispatched process 1
4 0-sys_listsyscall
5 17-sys_mmap
6 101-sys_killall
7 440-sys_terminate
8     CPU 0: Processed 1 has finished
9     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_mmap`, `101-sys_killall`, và `440-sys_terminate`.
- 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`:

```
1 #include "common.h"
2 #include "syscall.h"
3 #include "stdio.h"
4 #include "libmem.h"
5
6 int __sys_xxxhandler(struct pcb_t *caller, struct sc_regs *regs)
7 {
8     printf("The first system call parameter %d\n", regs->a1);
9     return 0;
10 }
```

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 The first system call parameter 1
5     CPU 0: Processed 1 has finished
6     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.

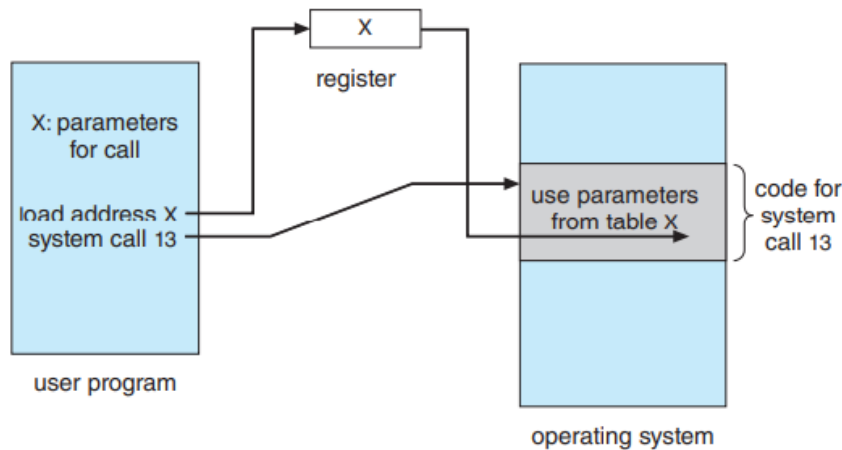
- Tiến trình sẽ thực hiện "syscall 440 1" để in ra tham số đầu tiên được truyền vào là 1.
- File sc3 đã hoàn tất và không còn tiến trình nào để chạy nên CPU 0 sẽ dừng.

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:

```
1 struct sc_regs {
2     uint32_t a1;
3     void*     a2; //Changing from uint32_t to void* for demo only
4     uint32_t a3;
5     uint32_t a4;
6     uint32_t a5;
7     uint32_t a6;
8     uint32_t orig_ax;
9     int32_t flags;
10 };
```

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

```
1 struct student
2 {
3     int mssv;
4     char name[100];
5     int age;
6 };
```

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:

```
1 #include "common.h"
2 #include "syscall.h"
3 #include "stdio.h"
4 #include <string.h>
5
6 struct student
7 {
8     int mssv;
9     char name[100];
10    int age;
11 };
12
13 int __sys_534handler(struct pcb_t *caller, struct sc_regs *regs)
14 {
15     // Dữ liệu khởi tạo ban đầu
16     struct student test_student;
17     test_student.mssv = 2312345;
18     strcpy(test_student.name, "10. He Dieu Hanh");
19     test_student.age = 20;
20
21     //Thanh ghi a2 lưu địa chỉ của dữ liệu cần xử lý
22     regs->a2 = &test_student;
23     printf("Gia tri cua thanh ghi a2 la: %p\n", regs->a2);
24
25     // Thực hiện thay đổi dữ liệu của người dùng
26     printf("Tuoi truoc khi thay doi: %d\n", test_student.age);
27     printf("Ten truoc khi thay doi: %s\n", test_student.name);
28
29     struct student* p_student = regs->a2;
30     if (p_student == NULL) return -1;
31
32     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 PRI0: 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 chen 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:

```
1 CPU 0: Dispatched process 3
2 CPU 1: Dispatched process 3
3 ...
4 CPU 0: Processed 3 has finished
5 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`, `mmap`, 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.