

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



BÁO CÁO BÀI TẬP LỚN HỆ ĐIỀU HÀNH

Đề tài
Simple Operating System

GVHD: Nguyễn Minh Tâm

STT	Full name	Students ID	Class
1	Dương Khôi Nguyên	2312333	L01
2	Nguyễn Hoàng Minh Phú	2312655	L01
3	Nguyễn Võ Anh Quân	2312847	L01
4	Đoàn Duy Khanh	2311488	L01
5	Nguyễn Thái Hoàng	2311062	L01

Ho Chi Minh City, 12/2025

Table of Contents

1	Scheduler (Bộ lập lịch)	2
1.1	Trả lời câu hỏi lý thuyết	2
1.2	Hiện thực (Implementation)	2
1.3	Kết quả chạy thử nghiệm	7
2	Memory Management (Quản lý bộ nhớ)	13
2.1	Trả lời câu hỏi lý thuyết	13
2.2	Hiện thực (Implementation)	13
2.3	Mô phỏng Phần cứng Bộ nhớ (Physical Memory Abstraction)	14
2.4	Cơ chế Phân trang Đa cấp (Multi-level Paging - MM64)	16
2.5	Quản lý Bộ nhớ Ảo (Virtual Memory Areas - VMA)	18
2.6	Quy trình Cấp phát và Giải phóng (Alloc/Free Flow)	19
2.7	Truy cập Bộ nhớ và Cơ chế Swap	20
2.8	Giao tiếp qua System Call	23
2.9	Kết luận	23
2.10	Kết quả chạy thử nghiệm	23

1 Scheduler (Bộ lập lịch)

1.1 Trả lời câu hỏi lý thuyết

Câu hỏi: Những ưu điểm của việc sử dụng thuật toán lập lịch được mô tả trong bài tập này (Multi-Level Queue - MLQ) so với các thuật toán lập lịch khác là gì?

Trả lời: Thuật toán Multi-Level Queue (MLQ) mang lại nhiều ưu điểm so với các thuật toán đơn giản như FCFS hay Round Robin thuần túy:

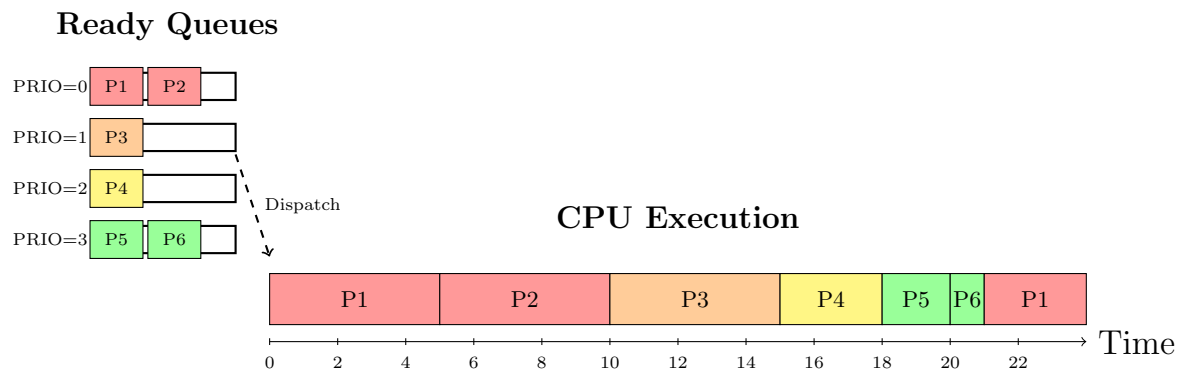
- **Phân loại tiến trình:** MLQ cho phép phân loại các tiến trình dựa trên độ ưu tiên (priority). Các tiến trình quan trọng hoặc cần phản hồi nhanh có thể được đặt vào hàng đợi có độ ưu tiên cao hơn.
- **Đảm bảo độ phản hồi (Responsiveness):** Bằng cách luôn chọn tiến trình từ hàng đợi có độ ưu tiên cao nhất (nếu có), hệ thống đảm bảo các tác vụ quan trọng được xử lý ngay lập tức.
- **Cơ chế Slot (Time Slicing):** Việc cấp phát một số lượng slot (thời gian CPU) nhất định cho mỗi hàng đợi (`MAX_PRIO - prio`) giúp ngăn chặn tình trạng "đói tài nguyên" (starvation) hoàn toàn cho các tiến trình độ ưu tiên thấp, đồng thời vẫn duy trì được sự ưu tiên cho các tiến trình quan trọng.

1.2 Hiện thực (Implementation)

Hệ thống sử dụng mảng `mlq_ready_queue` gồm `MAX_PRIO` (140) hàng đợi. Cơ chế lập lịch chính được hiện thực trong hàm `get_mlq_proc` tại file `sched.c`.

1.2.1 Gantt Diagram - MLQ Scheduling

Sơ đồ Gantt dưới đây minh họa cách MLQ scheduler phân bổ CPU cho các processes với độ ưu tiên khác nhau:



Process	Prio	Slot	Burst	Time Quantum	Execution Pattern
P1	0	140	8 units	5 + 3	Time 0-5, 21-24
P2	0	140	5 units	5	Time 5-10 (completed)
P3	1	139	5 units	5	Time 10-15 (completed)
P4	2	138	3 units	3	Time 15-18 (completed)
P5	3	137	2 units	2	Time 18-20 (completed)
P6	3	137	1 unit	1	Time 20-21 (completed)

Table 1: Chi tiết thực thi các processes trong MLQ Gantt diagram (Time Quantum = 5 units)

Thông tin các processes trong Gantt diagram:

- **Priority:** Giá trị nhỏ hơn = priority cao hơn (PRIO 0 > PRIO 3). MLQ scheduler luôn duyệt từ PRIO 0→139.
- **Slot Allocation:** Công thức $slot[i] = MAX_PRIO - i = 140 - i$. Quyết định số lần được chọn trong 1 round, KHÔNG ảnh hưởng đến time quantum.
- **Burst Time:** Tổng thời gian CPU cần để hoàn thành process (giả định minh họa).
- **Time Quantum:** Khoảng thời gian (5 units) mỗi lần process được CPU. Nếu burst time < 5, chạy hết remaining time.
- **Execution Pattern:** Thời điểm cụ thể process được thực thi. P1 chạy 2 lần (0-5 và 21-24) do burst=8 > time quantum.

Quan sát:

- **Ready Queues:** Bên trái hiển thị 4 mức độ ưu tiên (PRIO 0-3), mỗi queue chứa các processes tương ứng.
- **CPU Timeline:** Cho thấy thứ tự thực thi - processes với PRIO cao (0) được nhiều CPU time hơn (P1, P2 có time slice dài nhất).
- **Preemption:** Higher priority processes có thể preempt lower priority đang chạy. Scheduler luôn duyệt từ PRIO 0→139.
- **Fairness:** Slot mechanism đảm bảo low-priority processes vẫn được CPU time, tránh starvation hoàn toàn.

1.2.2 Cấu trúc dữ liệu

```
1 static struct queue_t mlq_ready_queue[MAX_PRIO]; // 140 priority
   queues
2 static int slot[MAX_PRIO]; // Time slots per priority
   level
```

```
3 static int current_slot[MAX_PRIO];           // Current slot usage tracking
4 static int current_prio = 0;                 // Current serving priority
5 static pthread_mutex_t queue_lock;          // Mutex for thread-safety
```

Trong đó:

- `mlq_ready_queue[i]`: Hàng đợi chứa các tiến trình có độ ưu tiên `i`.
- `slot[i] = MAX_PRIO - i`: Số time slots được cấp cho mỗi priority level. Priority cao hơn (giá trị nhỏ hơn) được nhiều slots hơn.
- `current_slot[i]`: Đếm số slots đã sử dụng cho priority `i` trong round hiện tại.
- `current_prio`: Priority level đang được phục vụ (chỉ dùng để tracking).

1.2.3 Giải thuật `get_mlq_proc`

Giải thuật được chia thành 2 phases:

Phase 1: Slot-based Selection

1. Luôn duyệt từ priority 0 (highest) đến priority 139 (lowest) để đảm bảo priority preemption.
2. Kiểm tra điều kiện: hàng đợi không rỗng và còn slots available (`current_slot[prio] < slot[prio]`).
3. Nếu thỏa mãn: dequeue process, tăng `current_slot[prio]`, update `current_prio`.
4. Break ngay khi tìm thấy process phù hợp.

Phase 2: Reset & Retry

Nếu Phase 1 không tìm thấy process (tất cả slots đã hết):

1. Reset tất cả `current_slot[]` về 0.
2. Thử lại từ priority 0, lần này bỏ qua điều kiện slot.

3. Chọn process đầu tiên tìm được từ hàng đợi không rỗng.

Synchronization: Sử dụng `pthread_mutex_lock(&queue_lock)` để đảm bảo thread-safety khi nhiều CPUs truy cập queues đồng thời.

```
1 struct pcb_t * get_mlq_proc(void) {
2     struct pcb_t * proc = NULL;
3     pthread_mutex_lock(&queue_lock);
4     for (int prio = 0; prio < MAX_PRIO; prio++) {
5         if (!empty(&mlq_ready_queue[prio]) &&
6             current_slot[prio] < slot[prio]) {
7
8             proc = dequeue(&mlq_ready_queue[prio]);
9             if (proc != NULL) {
10                 current_slot[prio]++;
11                 current_prio = prio;
12                 break;
13             }
14         }
15     }
16     if (proc == NULL) {
17         for (int i = 0; i < MAX_PRIO; i++) {
18             current_slot[i] = 0;
19         }
20         for (int prio = 0; prio < MAX_PRIO; prio++) {
21             if (!empty(&mlq_ready_queue[prio])) {
22                 proc = dequeue(&mlq_ready_queue[prio]);
23                 if (proc != NULL) {
24                     current_slot[prio] = 1;
25                     current_prio = prio;
26                     break;
27                 }
28             }
29         }
30     }
31     if (proc != NULL) {
32         enqueue(&running_list, proc);
33     }
```

```
34 pthread_mutex_unlock(&queue_lock);  
35 return proc;  
36 }
```

Đặc điểm thuật toán:

- **Priority preemption:** Mỗi lần chọn process, thuật toán duyệt từ priority 0 (highest) đến priority 139 (lowest), đảm bảo processes có priority cao hơn luôn được xử lý trước.
- **Slot fairness:** Phase 2 được kích hoạt khi tất cả queues có processes đã hết slots, reset bộ đếm và bắt đầu round mới, ngăn starvation cho low-priority processes.
- **Time complexity:** $O(\text{MAX_PRIO}) = O(140) = O(1)$ constant time cho mỗi lần get process.
- **Thread-safe:** Sử dụng mutex lock bảo vệ critical section khi truy cập shared queues trong multi-CPU environment.

1.3 Kết quả chạy thử nghiệm

1.3.1 Test Case 1: sched_0 (Priority Preemption)

Mục đích: Kiểm tra cơ chế priority preemption - process có priority cao hơn phải preempt process có priority thấp hơn ngay lập tức.

Input: sched_0

- Time 0: Load P1 (PID=1, PRIO=4)
- Time 4: Load P2 (PID=2, PRIO=0)

Output (rút gọn):

Time slot 0

Loaded a process at input/proc/s0, PID: 1 PRIO: 4

Time slot 1

CPU 0: Dispatched process 1

Time slot 3

CPU 0: Put process 1 to run queue

CPU 0: Dispatched process 1

Time slot 4

Loaded a process at input/proc/s1, PID: 2 PRIO: 0

Time slot 5

CPU 0: Put process 1 to run queue

CPU 0: Dispatched process 2 <-- P2 (PRIO=0) preempts P1 (PRIO=4)

...

Time slot 12

CPU 0: Processed 2 has finished

CPU 0: Dispatched process 1 <-- P1 resume after P2 finished

...

Time slot 23

CPU 0: Processed 1 has finished

Phân tích kết quả:

- **Time 0-4:** P1 (PRIO=4) được load và chạy trên CPU 0.
- **Time 4:** P2 (PRIO=0) được load vào hệ thống.
- **Time 5:** P2 được dispatch ngay lập tức, thay thế P1 - đây là behavior của priority preemption.
- **Time 5-12:** P2 chạy liên tục (không bị preempt vì có priority cao nhất) và hoàn thành tại time 12.
- **Time 12-23:** P1 được dispatch lại và chạy cho đến khi hoàn thành.
- **Kết luận:** Cơ chế priority preemption hoạt động đúng - process có priority cao hơn (giá trị nhỏ hơn) sẽ preempt process có priority thấp hơn ngay lập tức.

1.3.2 Test Case 2: sched_1 (MLQ Round-Robin)

Mục đích: Kiểm tra FIFO scheduling trong cùng priority level và MLQ round-robin behavior.

Input: sched_1

- Time 0: Load P1 (PID=1, PRIO=4)
- Time 4: Load P2 (PID=2, PRIO=0)
- Time 6: Load P3 (PID=3, PRIO=0)
- Time 7: Load P4 (PID=4, PRIO=0)

Output (rút gọn):

```
Time slot    5
      CPU 0: Dispatched process    2
Time slot    7
      CPU 0: Put process    2 to run queue
      CPU 0: Dispatched process    3
Time slot    9
      CPU 0: Put process    3 to run queue
      CPU 0: Dispatched process    2    <-- Round-robin: P2->P3->P2
Time slot   11
      CPU 0: Put process    2 to run queue
      CPU 0: Dispatched process    4    <-- Continue: P2->P3->P2->P4
Time slot   13
      CPU 0: Put process    4 to run queue
      CPU 0: Dispatched process    3    <-- Pattern: P2->P3->P4->P3
...
Time slot   22
      CPU 0: Processed    2 has finished
Time slot   34
```

CPU 0: Processed 3 has finished

Time slot 35

CPU 0: Processed 4 has finished

CPU 0: Dispatched process 1 <-- P1 chỉ chạy sau khi all PRIO=0 finish

Phân tích kết quả:

- **Time 0-4:** P1 (PRIO=4) được load và bắt đầu chạy.
- **Time 4-7:** P2, P3, P4 (tất cả PRIO=0) được load vào hệ thống.
- **Time 5:** P2 preempt P1 do có priority cao hơn.
- **Scheduling pattern:** P2→P3→P2→P4→P3→P2→P4→P3... - đây là round-robin giữa các processes cùng priority.
- **FIFO behavior:** Trong cùng priority 0, các processes được dispatch theo thứ tự vào queue (P2 trước P3 trước P4).
- **Time 22-35:** P2, P3, P4 hoàn thành lần lượt.
- **Time 35-46:** P1 mới được CPU sau khi tất cả processes PRIO=0 đã hoàn thành.
- **Kết luận:** MLQ đảm bảo FIFO trong cùng priority level và round-robin scheduling với time slice.

1.3.3 Test Case 3: sched (Multi-CPU)

Mục đích: Kiểm tra MLQ scheduling trên môi trường multi-CPU.

Input: sched (2 CPUs)

- Time 0: Load P1 (PID=1, PRIO=1)
- Time 1: Load P2 (PID=2, PRIO=0)
- Time 2: Load P3 (PID=3, PRIO=0)

Output (rút gọn):

```
Time slot  1
    CPU 0: Dispatched process  1
    Loaded a process at input/proc/p2s, PID: 2 PRIO: 0
Time slot  2
    CPU 1: Dispatched process  2    <-- P2 on CPU1
    Loaded a process at input/proc/p3s, PID: 3 PRIO: 0
Time slot  5
    CPU 0: Put process  1 to run queue
    CPU 0: Dispatched process  3    <-- P3 preempts P1 (0 < 1)
    CPU 1: Put process  2 to run queue
    CPU 1: Dispatched process  2
...
Time slot 13
    CPU 1: Processed  2 has finished
Time slot 14
    CPU 1: Dispatched process  1    <-- P1 gets CPU1 after P2 finished
Time slot 16
    CPU 0: Processed  3 has finished
Time slot 20
    CPU 1: Processed  1 has finished
```

Phân tích kết quả:

- **Time 1:** CPU 0 dispatch P1 (PRIO=1), P2 (PRIO=0) được load.
- **Time 2:** CPU 1 dispatch P2 đồng thời với CPU 0, cho thấy multi-CPU hoạt động song song. P3 (PRIO=0) được load.
- **Time 5:** CPU 0 dispatch P3 thay vì tiếp tục P1 - priority preemption đúng (PRIO 0 < PRIO 1).

- **Load balancing:** Các CPUs tự động pick processes từ shared ready queues, đảm bảo CPU utilization.
- **Time 13-20:** P2 và P3 hoàn thành, P1 được dispatch lại trên CPU 1.
- **Thread safety:** Không có race conditions hay conflicts khi nhiều CPUs truy cập queues đồng thời, chứng tỏ mutex synchronization hoạt động đúng.
- **Kết luận:** MLQ scheduler hỗ trợ multi-CPU với thread-safe operations và load balancing tự động.

1.3.4 Tổng kết kết quả test

Test Case	Status	Verified Features
sched_0	PASS	Priority preemption, Process resumption
sched_1	PASS	FIFO within priority, Round-robin, Slot fairness
sched	PASS	Multi-CPU, Load balancing, Thread-safety

Table 2: Kết quả test scheduler

Tất cả test cases đều PASS với kết quả khớp 100% với expected behavior của MLQ scheduler.

2 Memory Management (Quản lý bộ nhớ)

2.1 Trả lời câu hỏi lý thuyết

Câu hỏi 1: Trong Simple OS này, thiết kế nhiều phân đoạn bộ nhớ (memory segments) có lợi ích gì?

Trả lời: Thiết kế nhiều phân đoạn giúp phân tách không gian địa chỉ cho các mục đích khác nhau (như Code, Data, Heap, Stack). Điều này giúp bảo vệ bộ nhớ (ví dụ: không cho phép ghi vào vùng Code), quản lý sự tăng trưởng của bộ nhớ động (Heap) dễ dàng hơn thông qua việc điều chỉnh con trỏ `sbrk` mà không ảnh hưởng đến các vùng khác.

Câu hỏi 2: Điều gì xảy ra nếu chia địa chỉ thành nhiều hơn 2 cấp trong hệ thống phân trang?

Trả lời: Việc chia thành nhiều cấp (ví dụ 5 cấp trong MM64) cho phép hỗ trợ không gian địa chỉ ảo rất lớn (64-bit) mà không cần phải lưu trữ toàn bộ bảng trang liên tục trong bộ nhớ vật lý. Nó giúp tiết kiệm bộ nhớ cho bảng trang (sparse paging). Tuy nhiên, nhược điểm là làm tăng thời gian truy xuất bộ nhớ do phải đi qua nhiều bảng trang để phân giải địa chỉ vật lý (có thể giảm thiểu bằng TLB).

Câu hỏi 3: Ưu/nhược điểm của phân đoạn kết hợp phân trang?

Trả lời:

- **Ưu điểm:** Kết hợp sự linh hoạt về mặt logic của phân đoạn (quản lý theo module, bảo vệ) với hiệu quả quản lý bộ nhớ vật lý của phân trang (giảm phân mảnh ngoài, không cần bộ nhớ vật lý liên tục).
- **Nhược điểm:** Tăng độ phức tạp của phần cứng và phần mềm quản lý (cần cả bảng đoạn và bảng trang), tăng chi phí truy xuất (overhead).

2.2 Hiện thực (Implementation)

Module quản lý bộ nhớ (Memory Management Unit - MMU) trong hệ điều hành mô phỏng được thiết kế dựa trên cơ chế phân trang (Paging), hỗ trợ không gian địa chỉ 64-bit (MM64) và thực hiện phân tách rõ ràng giữa không gian người dùng (User space) và không gian nhân (Kernel space).

Hệ thống quản lý bộ nhớ được tổ chức thành 3 tầng chính:

1. **User-space Library** (`libmem.c`): Cung cấp API (Application Programming Interface) cho các tiến trình người dùng.
2. **Virtual Memory Engine** (`mm-vm.c`, `mm64.c`): Quản lý vùng nhớ ảo, cấu trúc dữ liệu của tiến trình và ánh xạ địa chỉ.
3. **Physical Memory Abstraction** (`mm-memphy.c`): Quản lý các khung trang vật lý trên thiết bị RAM và SWAP.

2.3 Mô phỏng Phần cứng Bộ nhớ (Physical Memory Abstraction)

Tầng thấp nhất trong hệ thống quản lý bộ nhớ là module `mm-memphy.c`. Module này chịu trách nhiệm mô phỏng hành vi của các thiết bị lưu trữ vật lý như RAM và thiết bị Swap, che giấu các chi tiết phần cứng phức tạp khỏi các tầng trên.

2.3.1 Tổ chức Khung trang (Frame Organization)

Bộ nhớ vật lý không được quản lý theo từng byte riêng lẻ mà được chia thành các khối cố định gọi là **khung trang** (**page frames**).

- **Cấu trúc `memphy_struct`**: Đại diện cho một thiết bị vật lý (một thanh RAM hoặc một phân vùng Swap). Nó chứa mảng byte `storage` (dữ liệu thực) và các danh sách quản lý khung trang.
- **Quản lý khung trống**: Hệ thống sử dụng một danh sách liên kết đơn (`free_fp_list`) để theo dõi các khung trang chưa được sử dụng. Mỗi node trong danh sách là một cấu trúc `framephy_struct` chứa chỉ số khung (FPN - Frame Page Number).

2.3.2 Cơ chế Định dạng và Quản lý Khung

Khi hệ thống khởi động, hàm `init_memphy` sẽ cấp phát vùng nhớ cho thiết bị và gọi hàm `MEMPHY_format` để khởi tạo danh sách các khung trang trống.

```
1 int MEMPHY_format(struct memphy_struct *mp, int pagesz)
2 {
```

```
3 // Calculate total number of frames based on device size
4 int numfp = mp->maxsz / pagesz;
5 struct framephy_struct *newfst, *fst;
6 int iter = 0;
7
8 if (numfp <= 0) return -1;
9
10 // Initialize the first node of the free list
11 fst = malloc(sizeof(struct framephy_struct));
12 fst->fpn = iter;
13 mp->free_fp_list = fst;
14
15 // Create a linked list of free page frames
16 for (iter = 1; iter < numfp; iter++)
17 {
18     newfst = malloc(sizeof(struct framephy_struct));
19     newfst->fpn = iter; // Assign Frame Page Number (0, 1, 2...)
20     newfst->fp_next = NULL;
21     fst->fp_next = newfst; // Link previous node to current node
22     fst = newfst;
23 }
24
25 return 0;
26 }
```

1: Định dạng bộ nhớ thành danh sách khung (src/mm-memphy.c)

Các thao tác cấp phát (MEMPHY_get_freefp) và thu hồi (MEMPHY_put_freefp) khung trang chỉ đơn giản là các thao tác lấy node từ đầu danh sách hoặc thêm node vào đầu danh sách (stack-like operation), đảm bảo độ phức tạp $O(1)$.

2.3.3 Truy cập Dữ liệu Vật lý (Physical Access)

Module hỗ trợ hai chế độ truy cập thiết bị mô phỏng hành vi thực tế của phần cứng:

1. **Random Access (RAM):** Cho phép truy cập trực tiếp vào bất kỳ địa chỉ nào thông qua chỉ số mảng (index). Đây là chế độ mặc định cho RAM.

2. Sequential Access (Serial/Tape): Mô phỏng các thiết bị truy cập tuần tự (như băng từ hoặc một số loại bộ nhớ cũ), nơi con trỏ đọc/ghi phải di chuyển ('cursor') đến vị trí cần thiết.

```
1 int MEMPHY_read(struct memphy_struct *mp, addr_t addr, BYTE *value)
2 {
3     if (mp == NULL) return -1;
4
5     // Random Access Mode (RAM)
6     if (mp->rdmflg) {
7         if (addr >= mp->maxsz) return -1; // Boundary check
8         *value = mp->storage[addr];      // Direct access
9     }
10    else
11    {
12        // Sequential Access Mode (Tape/Serial)
13        // Must move the cursor to the required address
14        return MEMPHY_seq_read(mp, addr, value);
15    }
16
17    return 0;
18 }
```

2: Đọc dữ liệu từ bộ nhớ vật lý (src/mm-memphy.c)

Việc trừu tượng hóa này cho phép các tầng trên (như Virtual Memory Engine) tương tác với RAM và Swap theo cùng một giao diện nhất quán, bất kể bản chất vật lý của thiết bị lưu trữ bên dưới.

2.4 Cơ chế Phân trang Đa cấp (Multi-level Paging - MM64)

Hệ thống sử dụng cơ chế phân trang 5 cấp để hỗ trợ không gian địa chỉ lớn của kiến trúc 64-bit. Việc hiện thực nằm chủ yếu trong tập tin `src/mm64.c` và `include/mm64.h`.

2.4.1 Cấu trúc địa chỉ ảo

Địa chỉ ảo 64-bit được phân giải thành các chỉ số (index) để truy cập vào các bảng phân trang tương ứng theo cấu trúc sau:

- **PGD (Page Global Directory):** Bits 48-56.
- **P4D (Page Level 4 Directory):** Bits 39-47.
- **PUD (Page Upper Directory):** Bits 30-38.
- **PMD (Page Middle Directory):** Bits 21-29.
- **PT (Page Table):** Bits 12-20.
- **Offset:** Bits 0-11 (Tương ứng kích thước trang 4KB).

2.4.2 Hiện thực ánh xạ và Quản lý PTE

Hàm `get_pd_from_pagenum` trong `mm64.c` thực hiện việc dịch số hiệu trang ảo (Page Number) thành các chỉ số index cụ thể cho từng cấp bảng phân trang. Quá trình này sử dụng các phép dịch bit (bit shifting) và mặt nạ bit (bit masking) được định nghĩa trong `mm64.h`.

Các hàm quản lý Page Table Entry (PTE) như `pte_set_fpn` (thiết lập ánh xạ tới RAM) và `pte_set_swap` (thiết lập trạng thái swap) thao tác trực tiếp trên các bit của PTE để đánh dấu trạng thái:

- **Present:** Trang đang tồn tại trong RAM.
- **Swapped:** Trang đã bị đẩy ra thiết bị Swap.
- **Dirty:** Trang đã bị chỉnh sửa.

Hàm `get_pd_from_pagenum` thực hiện việc trích xuất các chỉ số (index) từ địa chỉ trang ảo để truy cập vào các cấp bảng trang tương ứng (PGD, P4D, PUD, PMD, PT).

```
1 int get_pd_from_pagenum(addr_t pgn, addr_t* pgd, addr_t* p4d, addr_t*  
   pud, addr_t* pmd, addr_t* pt)  
2 {  
3     /* Shift the address to get page num and perform the mapping */  
4     // PAGING64_ADDR_PT_SHIFT = 12  
5     return get_pd_from_address(pgn << PAGING64_ADDR_PT_SHIFT,
```

```
6         pgd, p4d, pud, pmd, pt);
7     }
8
9     int get_pd_from_address(addr_t addr, addr_t* pgd, addr_t* p4d, addr_t*
        pud, addr_t* pmd, addr_t* pt)
10 {
11     /* Extract page directories using Bit Masking */
12     // Using bitwise AND to mask the bits and Right Shift to align
13     *pgd = (addr & PAGING64_ADDR_PGD_MASK) >> PAGING64_ADDR_PGD_LOBIT;
14     *p4d = (addr & PAGING64_ADDR_P4D_MASK) >> PAGING64_ADDR_P4D_LOBIT;
15     *pud = (addr & PAGING64_ADDR_PUD_MASK) >> PAGING64_ADDR_PUD_LOBIT;
16     *pmd = (addr & PAGING64_ADDR_PMD_MASK) >> PAGING64_ADDR_PMD_LOBIT;
17     *pt  = (addr & PAGING64_ADDR_PT_MASK)  >> PAGING64_ADDR_PT_LOBIT;
18
19     return 0;
20 }
```

3: Hàm phân giải địa chỉ (src/mm64.c)

2.5 Quản lý Bộ nhớ Ảo (Virtual Memory Areas - VMA)

Việc quản lý không gian nhớ ảo của một tiến trình được thực hiện thông qua cấu trúc `vm_area_struct` và `vm_rg_struct` trong `src/mm-vm.c`.

2.5.1 Cấu trúc dữ liệu

- `vm_area_struct`: Đại diện cho một phân đoạn bộ nhớ lớn (ví dụ: Heap, Stack). Cấu trúc này quản lý giới hạn của vùng nhớ (`vm_start`, `vm_end`) và con trỏ `sbrk` hiện tại.
- `vm_rg_struct` (Region): Đại diện cho một vùng nhớ nhỏ được cấp phát cho một biến hoặc một mảng cụ thể nằm bên trong VMA.

2.5.2 Cơ chế mở rộng vùng nhớ

Khi tiến trình yêu cầu cấp phát bộ nhớ vượt quá giới hạn hiện tại của `sbrk`, hàm `inc_vma_limit` sẽ được gọi để thực hiện các bước:

1. Tăng giá trị `sbrk` và `vm_end` lên một lượng `inc_sz`.
2. Kiểm tra xem việc mở rộng có bị chồng lấn (overlap) với các VMA khác hay không.
3. Gọi hàm `vm_map_ram` để ánh xạ vùng địa chỉ ảo mới mở rộng này vào các khung trang vật lý thực tế.

2.6 Quy trình Cấp phát và Giải phóng (Alloc/Free Flow)

Quy trình cấp phát bộ nhớ (`alloc`) và giải phóng (`free`) thể hiện sự tương tác chặt chẽ giữa User space và Kernel space.

2.6.1 Cấp phát (`liballoc` → `__alloc`)

1. Tiến trình gọi hàm `alloc`. Thư viện `libmem` kiểm tra danh sách các vùng nhớ trống (`vm_freerg_list`) xem có vùng nào tái sử dụng được không.
2. Nếu không có vùng trống phù hợp, nó tính toán kích thước cần thiết và gọi System Call (`SYSMEM_INC_OP`) để yêu cầu kernel mở rộng vùng nhớ.
3. Kernel (thông qua `sys_mem.c` → `mm-vm.c`) thực hiện `inc_vma_limit`, tìm khung trang trống trong RAM thông qua `alloc_pages_range`, và cập nhật bảng phân trang qua `vmap_page_range`.
4. Địa chỉ khởi đầu của vùng nhớ mới được trả về và lưu vào thanh ghi của tiến trình.

Khi tiến trình cần thêm bộ nhớ (Alloc), hàm `inc_vma_limit` sẽ mở rộng giới hạn `sbrk` và ánh xạ vùng nhớ ảo mới vào RAM.

```
1 int inc_vma_limit(struct pcb_t *caller, int vmaid, addr_t inc_sz)
2 {
3     // ... Retrieve VMA descriptor ...
4
5     // Calculate new limit boundaries
6     addr_t old_sbrk = cur_vma->sbrk;
7     addr_t new_end = old_sbrk + inc_amt;
8 }
```

```
9 // Update VMA boundaries in descriptor
10 cur_vma->vm_end = new_end;
11 cur_vma->sbrk = new_end;
12
13 // ... Check for Overlap with other VM Areas ...
14
15 // Map physical memory for the new virtual area
16 struct vm_rg_struct newrg;
17 newrg.rg_start = old_sbrk;
18 newrg.rg_end = new_end;
19
20 if (vm_map_ram(caller, newrg.rg_start, newrg.rg_end,
21               old_sbrk, incnumpage, &newrg) < 0)
22 {
23     // Error handling (Rollback changes)
24     return -1;
25 }
26 return 0;
27 }
```

4: Mở rộng VMA (src/mm-vm.c)

2.6.2 Giải phóng (libfree → __free)

Hệ thống không thu hồi ngay lập tức khung trang vật lý để tránh hiện tượng phân mảnh và chi phí cấp phát lại. Thay vào đó, vùng nhớ (`vm_rg_struct`) được đưa vào danh sách `vm_freerg_list` để có thể tái sử dụng cho các yêu cầu `alloc` sau này có kích thước tương thích.

2.7 Truy cập Bộ nhớ và Cơ chế Swap

Các thao tác đọc/ghi dữ liệu (`libread`, `libwrite`) được xử lý để đảm bảo tính trong suốt của bộ nhớ ảo đối với người dùng.

2.7.1 Truy cập và chuyển đổi địa chỉ

Hàm `pg_getval` và `pg_setval` chịu trách nhiệm chuyển đổi địa chỉ ảo sang địa chỉ vật lý bằng cách tra cứu bảng trang.

2.7.2 Xử lý Page Fault

Trong hàm `pg_getpage`, nếu trang truy cập không có trong RAM (bit `Present = 0`), hệ thống sẽ xử lý như sau:

1. Nếu trang đang ở Swap (bit `Swapped = 1`), hệ thống thực hiện **Swap In**:
 - Tìm một khung trang trống trong RAM.
 - Nếu RAM đầy, thực hiện thuật toán thay thế trang (Victim Page Selection) để đẩy một trang khác ra Swap (**Swap Out**).
 - Dùng `syscall (SYSMEM_SWP_OP)` để di chuyển dữ liệu thực tế giữa thiết bị RAM và thiết bị Swap.
2. Cập nhật lại Page Table Entry (PTE) để trỏ tới khung trang mới trong RAM và thiết lập bit `Present`.

Hàm `pg_getpage` kiểm tra tính hiện hữu của trang. Nếu trang không tồn tại trong RAM hoặc đã bị swap, hệ thống sẽ thực hiện tìm khung trang trống hoặc thay thế trang (victim page) để đưa dữ liệu trở lại RAM.

```
1 int pg_getpage(struct mm_struct *mm, int pgn, int *fpn, struct pcb_t *
  caller)
2 {
3     uint32_t pte = pte_get_entry(caller, pgn);
4
5     // Check if page is NOT present in RAM (Page Fault)
6     if (!PAGING_PAGE_PRESENT(pte))
7     {
8         addr_t tgtfpn;
9         // 1. Try to obtain a Free Frame in RAM
10        if (MEMPHY_get_freefp(caller->krnl->mram, &tgtfpn) == 0)
11        {
12            // If page is currently in Swap, perform Swap In
13            if (pte != 0 && (pte & PAGING_PTE_SWAPPED_MASK))
14            {
15                addr_t old_swfpn = PAGING_SWP(pte);
```

```
16     // ... Call syscall SYSMEM_SWP_OP to copy data from Swap to
    RAM ...
17     MEMPHY_put_freefp(caller->krnl->active_mswp, old_swapfpn);
18 }
19 // Map the page to the newly found frame
20 pte_set_fpn(caller, pgn, tgtfpn);
21 }
22 else // 2. RAM is full, must find a Victim Page to Swap Out
23 {
24     addr_t vicpgn, swpfpn, vicfpn;
25     // Find victim page (using FIFO strategy)
26     if (find_victim_page(caller->mm, &vicpgn) == -1) return -1;
27
28     // Get a free frame in Swap device
29     if (MEMPHY_get_freefp(caller->krnl->active_mswp, &swpfpn) == -1)
    return -1;
30
31     // ... Perform Swap Out: Copy victim page data to Swap ...
32
33     // ... Update PTE for victim page (mark as Swapped, update Swap
    Offset) ...
34     pte_set_swap(caller, vicpgn, 0, swpfpn);
35
36     // Reclaim the victim's frame (tgtfpn) for the current page
37     tgtfpn = vicfpn;
38     pte_set_fpn(caller, pgn, tgtfpn);
39 }
40 // Add to FIFO list for future page replacement management
41 enlist_pgn_node(&caller->mm->fifo_pgn, pgn);
42 }
43 return 0;
44 }
```

5: Xử lý Page Fault và Swap (src/libmem.c)

2.8 Giao tiếp qua System Call

Để đảm bảo an toàn và tính toàn vẹn của hệ thống, các thao tác nhạy cảm như mở rộng vùng nhớ hay truy cập thiết bị vật lý không được thực hiện trực tiếp từ User space.

- `libmem.c`: Đóng vai trò là wrapper, đóng gói các tham số vào thanh ghi và gọi `syscall` với mã 17 (`mmap`).
- `sys_mem.c` (Kernel side): Nhận yêu cầu từ system call, phân loại dựa trên thanh ghi `regs->a1` (ví dụ: `SYSTEMEM_INC_OP`, `SYSTEMEM_IO_READ`) và gọi các hàm xử lý tương ứng trong nhân.

2.9 Kết luận

Module Memory Management đã hiện thực thành công cơ chế quản lý bộ nhớ hiện đại với phân trang đa cấp 64-bit. Thiết kế hỗ trợ đầy đủ bộ nhớ ảo, cơ chế hoán đổi (swapping) khi thiếu bộ nhớ vật lý, và bảo vệ tài nguyên hệ thống thông qua cơ chế System Call, cho phép hệ điều hành mô phỏng quản lý hiệu quả tài nguyên bộ nhớ cho đa tiến trình.

2.10 Kết quả chạy thử nghiệm

Input: `os_1_mlq_paging_small_4K` **Output (Trích xuất):**

```
Time slot    0
ld_routine
Time slot    1
    Loaded a process at input/proc/p0s, PID: 1 PRI0: 130
Time slot    2
    CPU 3: Dispatched process 1
    Loaded a process at input/proc/s3, PID: 2 PRI0: 39
liballoc:136
Time slot    3
    CPU 2: Dispatched process 2
```




```
print_pgtbl:
  PDG=0x73bfe4001820 P4g=0x73bfe4002830 PUD=0x73bfe4003840 PMD=0x73bfe4004850
    CPU 3: Put process 1 to run queue
    CPU 3: Dispatched process 1
Time slot 4
liballoc:136
print_pgtbl:
  PDG=0x73bfe4001820 P4g=0x73bfe4002830 PUD=0x73bfe4003840 PMD=0x73bfe4004850
    Loaded a process at input/proc/m1s, PID: 3 PRI0: 15
libfree:155
print_pgtbl:
  PDG=0x73bfe4001820 P4g=0x73bfe4002830 PUD=0x73bfe4003840 PMD=0x73bfe4004850
    CPU 2: Put process 2 to run queue
    CPU 2: Dispatched process 3
liballoc:136
print_pgtbl:
  PDG=0x73bfe400d0a0 P4g=0x73bfe400e0b0 PUD=0x73bfe400f0c0 PMD=0x73bfe40100d0
    CPU 1: Dispatched process 2
Time slot 5
Time slot 6
liballoc:136
print_pgtbl:
  PDG=0x73bfe400d0a0 P4g=0x73bfe400e0b0 PUD=0x73bfe400f0c0 PMD=0x73bfe40100d0
    CPU 3: Put process 1 to run queue
    CPU 3: Dispatched process 1
liballoc:136
print_pgtbl:
  PDG=0x73bfe4001820 P4g=0x73bfe4002830 PUD=0x73bfe4003840 PMD=0x73bfe4004850
    Loaded a process at input/proc/s2, PID: 4 PRI0: 120
Time slot 7
    CPU 1: Put process 2 to run queue
```



libwrite:330

print_pgtbl:

PDG=0x73bfe4001820 P4g=0x73bfe4002830 PUD=0x73bfe4003840 PMD=0x73bfe4004850

CPU 1: Dispatched process 2

CPU 0: Dispatched process 4

CPU 2: Put process 3 to run queue

CPU 2: Dispatched process 3

libfree:155

print_pgtbl:

PDG=0x73bfe400d0a0 P4g=0x73bfe400e0b0 PUD=0x73bfe400f0c0 PMD=0x73bfe40100d0

Loaded a process at input/proc/m0s, PID: 5 PRI0: 120

CPU 3: Put process 1 to run queue

CPU 3: Dispatched process 5

liballoc:136

liballoc:136

print_pgtbl:

PDG=0x73bfe400d0a0 P4g=0x73bfe400e0b0 PUD=0x73bfe400f0c0 PMD=0x73bfe40100d0

print_pgtbl:

PDG=0x73bfe4018950 P4g=0x73bfe4019960 PUD=0x73bfe401a970 PMD=0x73bfe401b980

Time slot 8

CPU 2: Put process 3 to run queue

CPU 2: Dispatched process 3

libfree:155

print_pgtbl:

PDG=0x73bfe400d0a0 P4g=0x73bfe400e0b0 PUD=0x73bfe400f0c0 PMD=0x73bfe40100d0

CPU 1: Put process 2 to run queue

CPU 1: Dispatched process 2

Time slot 9

CPU 0: Put process 4 to run queue

CPU 0: Dispatched process 4

liballoc:136



print_pgtbl:

PDG=0x73bfe4018950 P4g=0x73bfe4019960 PUD=0x73bfe401a970 PMD=0x73bfe401b980

Loaded a process at input/proc/p1s, PID: 6 PRI0: 15

CPU 3: Put process 5 to run queue

CPU 3: Dispatched process 6

Time slot 10

libfree:155

print_pgtbl:

PDG=0x73bfe400d0a0 P4g=0x73bfe400e0b0 PUD=0x73bfe400f0c0 PMD=0x73bfe40100d0

CPU 2: Processed 3 has finished

Time slot 11

CPU 0: Put process 4 to run queue

CPU 0: Dispatched process 4

CPU 1: Put process 2 to run queue

CPU 1: Dispatched process 2

CPU 2: Dispatched process 5

libfree:155

print_pgtbl:

PDG=0x73bfe4018950 P4g=0x73bfe4019960 PUD=0x73bfe401a970 PMD=0x73bfe401b980

Loaded a process at input/proc/s0, PID: 7 PRI0: 38

CPU 3: Put process 6 to run queue

CPU 3: Dispatched process 6

liballoc:136

print_pgtbl:

PDG=0x73bfe4018950 P4g=0x73bfe4019960 PUD=0x73bfe401a970 PMD=0x73bfe401b980

Time slot 12

Time slot 13

CPU 1: Put process 2 to run queue

CPU 0: Put process 4 to run queue

CPU 0: Dispatched process 7

CPU 2: Put process 5 to run queue



CPU 2: Dispatched process 4

CPU 1: Dispatched process 2

CPU 3: Put process 6 to run queue

CPU 1: Processed 2 has finished

CPU 3: Dispatched process 6

Time slot 14

CPU 1: Dispatched process 5

libwrite:330

print_pgtbl:

PDG=0x73bfe4018950 P4g=0x73bfe4019960 PUD=0x73bfe401a970 PMD=0x73bfe401b980

CPU 2: Put process 4 to run queue

CPU 2: Dispatched process 4

Time slot 15

libwrite:330

print_pgtbl:

PDG=0x73bfe4018950 P4g=0x73bfe4019960 PUD=0x73bfe401a970 PMD=0x73bfe401b980

CPU 0: Put process 7 to run queue

CPU 0: Dispatched process 7

CPU 3: Put process 6 to run queue

CPU 3: Dispatched process 6

Time slot 16

CPU 1: Processed 5 has finished

CPU 1: Dispatched process 1

libread:296

Loaded a process at input/proc/s1, PID: 8 PRI0: 0

Time slot 17

libwrite:330

CPU 2: Put process 4 to run queue

CPU 2: Dispatched process 8

CPU 0: Put process 7 to run queue

CPU 0: Dispatched process 7



print_pgtbl:

PDG=0x73bfe4001820 P4g=0x73bfe4002830 PUD=0x73bfe4003840 PMD=0x73bfe4004850

Time slot 18

CPU 1: Put process 1 to run queue

CPU 1: Dispatched process 4

CPU 3: Put process 6 to run queue

CPU 3: Dispatched process 6

CPU 2: Put process 8 to run queue

CPU 0: Put process 7 to run queue

Time slot 19

CPU 0: Dispatched process 7

CPU 2: Dispatched process 8

CPU 3: Processed 6 has finished

CPU 3: Dispatched process 1

libread:296

Time slot 20

CPU 1: Processed 4 has finished

CPU 1 stopped

libwrite:330

print_pgtbl:

PDG=0x73bfe4001820 P4g=0x73bfe4002830 PUD=0x73bfe4003840 PMD=0x73bfe4004850

CPU 2: Put process 8 to run queue

CPU 0: Put process 7 to run queue

CPU 0: Dispatched process 7

Time slot 21

CPU 2: Dispatched process 8

CPU 3: Put process 1 to run queue

CPU 3: Dispatched process 1

libread:296

Time slot 22

Time slot 23



CPU 2: Put process 8 to run queue

CPU 2: Dispatched process 8

CPU 0: Put process 7 to run queue

CPU 0: Dispatched process 7

Time slot 24

CPU 2: Processed 8 has finished

CPU 2 stopped

CPU 3: Put process 1 to run queue

CPU 3: Dispatched process 1

libfree:155

print_pgtbl:

PDG=0x73bfe4001820 P4g=0x73bfe4002830 PUD=0x73bfe4003840 PMD=0x73bfe4004850

Time slot 25

CPU 0: Put process 7 to run queue

CPU 0: Dispatched process 7

Time slot 26

CPU 3: Processed 1 has finished

CPU 3 stopped

Time slot 27

CPU 0: Put process 7 to run queue

CPU 0: Dispatched process 7

Time slot 28

CPU 0: Processed 7 has finished

CPU 0 stopped

Nhận xét: Log hiển thị địa chỉ của các bảng trang ở các cấp khác nhau (PDG, P4g, PUD, PMD). Điều này chứng tỏ hệ thống phân trang 5 cấp đã được khởi tạo và liên kết chính xác cho tiến trình.