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

HỆ ĐIỀU HÀNH (CO2018)

BÁO CÁO BÀI TẬP LỚN

Simple Operating System

GVHD: Nguyễn Minh Tâm
Lớp: L02
Nhóm: 6

Sinh viên thực hiện	MSSV	Hoàn thành
Lê Thế Lộc	2311950	100%
Đỗ Minh Sang	2312930	100%
Trần Doãn Hoàng Lâm	2311828	100%
Vương Quốc Khánh	2311542	100%
Nguyễn Đặng Trí Dũng	2310554	100%

TP Hồ Chí Minh, Tháng 12 Năm 2025

Mục lục

1 Lời mở đầu

Với mục tiêu tìm hiểu cơ chế vận hành bên trong của một hệ điều hành hiện đại, nhóm chúng em đã thực hiện đồ án xây dựng hệ điều hành mô phỏng theo yêu cầu của môn học. Đồ án tập trung vào việc thiết kế và cài đặt các module quan trọng nhất của Kernel, bao gồm: Bộ lập lịch tiến trình (CPU Scheduler) sử dụng giải thuật MLQ và Bộ quản lý bộ nhớ (Memory Management Unit) sử dụng cơ chế phân trang đa cấp.

Quá trình thực hiện đề tài đã giúp nhóm củng cố vững chắc kiến thức về cách hệ điều hành cấp phát tài nguyên, cơ chế dịch địa chỉ ảo sang vật lý và cách xử lý đồng bộ đa luồng. Báo cáo dưới đây sẽ mô tả chi tiết kiến trúc hệ thống, các hàm chức năng chính và kết quả kiểm thử trên các kịch bản mô phỏng khác nhau.

Nhóm em xin gửi lời cảm ơn đến thầy Nguyễn Minh Tâm đã tận tình hướng dẫn nhóm trong việc thực hiện đề tài và kính mong nhận được những ý kiến đóng góp từ thầy để hoàn thiện hệ thống hơn trong tương lai.

1.1 Đường dẫn đến mã nguồn bài tập lớn

Mã nguồn của bài tập lớn được nhóm em lưu trữ và quản lý tại GitHub:

- TẠI ĐÂY

2 Scheduler

2.1 Cơ sở Lý thuyết

2.1.1 Kiến trúc hàng đợi đa cấp(Multi-Level Queue - MLQ)

Trong bài tập lớn này sử dụng thuật toán lập lịch Multi-Level Queue (MLQ). Đây là một thuật toán được thiết kế cho các hệ thống cần phân loại các tiến trình vào các nhóm khác nhau dựa trên mức độ ưu tiên (priority).

Nguyên lý hoạt động : Hệ thống sẽ duy trì read-queue với mỗi hàng đợi tương ứng với một mức ưu tiên cố định. Ở đây ta có tổng cộng là **MAX_PRIO** mức ưu tiên(đối với src code này là 140) với mức 0 là mức ưu tiên cao nhất và mức 139 là thấp nhất. Mỗi process khi được tạo ra sẽ được gán vào một hàng đợi tương ứng với độ ưu tiên của nó và không thay đổi trong suốt quá trình thực thi (Fixed Priority).

2.1.2 Cơ chế Slot và Chống Đói tài nguyên (Starvation Mitigation)

Một vấn đề thường gặp của các giải thuật dựa trên độ ưu tiên là Starvation (đói tài nguyên), nơi các tiến trình ưu tiên thấp có thể không bao giờ được chạy nếu luôn có các tiến trình ưu tiên cao trong hệ thống. Để giải quyết vấn đề này, hệ thống áp dụng cơ chế Slot (Định mức).

- Định nghĩa Slot: Mỗi hàng đợi ưu tiên được cấp một lượng "thời gian sử dụng CPU" gọi là slot. Số lượng slot tỉ lệ thuận với độ ưu tiên của hàng đợi đó.
- Công thức tính: **slot= MAX_PRIO - prio**
  Ví dụ: Hàng đợi mức 0 (cao nhất) có 140 slot, trong khi hàng đợi mức 139 (thấp nhất) chỉ có 1 slot.
- **Cơ chế vận hành:** Scheduler sẽ ưu tiên tiến trình từ hàng đợi cao nhất. Tuy nhiên mỗi lần một tiến trình từ hàng đợi prio được chọn, số slot của hàng đợi đó sẽ giảm đi. Khi slot giảm về 0, bộ lập lịch buộc phải nhường quyền (preempt) và chuyển sang xét duyệt các hàng đợi có độ ưu tiên thấp hơn kế tiếp, đồng thời nạp lại (reset) số slot cho hàng đợi vừa hết.

2.1.3 Đa xử lý và Đồng bộ hóa(Multi-Processing Synchronization)

Vì yêu cầu chạy trên phần cứng multi-cpu nên đặt ra các vấn đề liên quan đến đồng bộ hóa.

- Tài nguyên chia sẻ: Các hàng đợi ưu tiên (mlq_ready_queue) là tài nguyên chung mà tất cả các CPU đều có thể truy cập để lấy tiến trình (dequeue) hoặc trả tiến trình về (enqueue).
- Cơ chế khóa (Locking): Để đảm bảo tính toàn vẹn dữ liệu và tránh xung đột (race condition), các thao tác truy xuất vào hàng đợi phải được bảo vệ bằng cơ chế loại trừ tương hỗ (Mutual Exclusion - Mutex). Chỉ một CPU được phép thao tác trên hàng đợi tại một thời điểm.

2.2 Thiết kế và Hiện thực (Implementation)

2.2.1 Cấu trúc dữ liệu (Data Structures)

Hệ thống sử dụng các cấu trúc dữ liệu chính sau trong `src/sched.c` và `src/queue.c`:

- **Hàng đợi ưu tiên (`mlq_ready_queue`):** Mảng gồm `MAX_PRIO` (140) hàng đợi. Mỗi mức ưu tiên có một hàng đợi riêng biệt để chứa các tiến trình (PCB).

```
static struct queue_t mlq_ready_queue[MAX_PRIO];
```

- **Mảng định mức (`slot`):** Mảng số nguyên lưu trữ số lượng "khe thời gian" (quantum) còn lại mà một hàng đợi được phép sử dụng trước khi phải nhường quyền.

```
static int slot[MAX_PRIO];
```

- **Đồng bộ hóa (`queue_lock`):** Sử dụng `pthread_mutex_t` để bảo vệ tài nguyên chia sẻ, đảm bảo an toàn khi nhiều CPU cùng truy cập vào hàng đợi.

2.2.2 Giải thuật cho queue

Được thực hiện tại file `src/queue.c`

2.2.2.1.enqueue: thêm vào hàm đợi

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

Thuật toán hoạt động:

- **Cơ chế hoạt động:** Hàm thực hiện thêm một tiến trình vào cuối hàng đợi theo nguyên tắc FIFO (First-In-First-Out).
- **Kiểm tra an toàn:**
  - Đầu tiên sẽ kiểm tra tính hợp lệ của con trỏ để tránh lỗi truy cập bộ nhớ (Segmentation Fault).
  - Tiếp theo sẽ kiểm tra xem giới hạn của hàng đợi (‘MAX_QUEUE_SIZE‘). Đây là bước quan trọng để ngăn chặn lỗi tràn bộ đệm (Buffer Overflow) trong trường hợp mảng tĩnh được sử dụng để cài đặt hàng đợi.
- **Các bước:** Tiến trình mới được gán vào chỉ số ‘q->size‘. Vì mảng trong C bắt đầu từ 0, nếu ‘size‘ hiện tại là 5 (có các phần tử 0-4), phần tử mới sẽ nằm ở vị trí 5. Sau đó, biến đếm ‘size‘ được tăng lên để phản ánh trạng thái mới.

2.2.2.2.dequeue: Lấy ra khỏi hàng đợi

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

```

```
8       q->size--;
9       return p;
10  }
```

Giải thích cách hoạt động:

- **Nguyên tắc ưu tiên:** Trong cấu trúc hàng đợi, phần tử ở vị trí đầu tiên ('index 0') là phần tử được đưa vào sớm nhất, do đó nó sẽ được lấy ra đầu tiên để xử lý.
- **Thao tác dồn mảng (Shifting):** Do sử dụng mảng tĩnh, khi phần tử đầu tiên bị lấy đi, khoảng trống tại 'index 0' cần được lấp đầy. Vòng lặp 'for' (dòng 9-11) thực hiện di chuyển tất cả các phần tử còn lại sang trái một vị trí ('index i' chuyển về 'index i-1').
- **Độ phức tạp:** Thao tác này có độ phức tạp $O(N)$ do phải duyệt qua mảng để di chuyển phần tử.

2.2.3 Giải thuật Lập lịch (Scheduling Algorithm)

Logic chính được hiện thực trong hàm `get_mlq_proc()` tại file `src/sched.c`.

2.2.3.1 Khởi tạo Slot Mỗi hàng đợi ưu tiên `prio` được cấp số lượng slot ban đầu theo công thức:

$$\text{slot}[prio] = \text{MAX\_PRIO} - \text{prio}$$

Điều này đảm bảo các tiến trình có độ ưu tiên cao (giá trị `prio` nhỏ) sẽ nhận được nhiều thời gian CPU hơn so với các tiến trình ưu tiên thấp.

2.2.3.2.Cơ chế chọn tiến trình

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

Thuật toán hoạt động theo các bước sau:

1. Duyệt tuần tự các hàng đợi từ ưu tiên cao nhất (0) đến thấp nhất (`MAX_PRIO - 1`).
2. Kiểm tra nếu hàng đợi không rỗng (`!empty`):
   - **Nếu còn slot (slot > 0):** Hệ thống lấy tiến trình ra khỏi hàng đợi (`dequeue`), giảm slot đi 1, và trả về tiến trình đó ngay lập tức cho CPU.
   - **Nếu hết slot (slot ≤ 0):** Hệ thống nạp lại slot (`reset`) nhưng **không** lấy tiến trình ở hàng đợi này. Thay vào đó, nó tiếp tục vòng lặp để xét hàng đợi có độ ưu tiên thấp hơn kế tiếp.

2.3 Trả lời câu hỏi (Questionnaire)

Câu hỏi: *What are the advantages of using the scheduling algorithm described in this assignment compared to other scheduling algorithms you have learned?* (Các ưu điểm của thuật toán lập lịch trong bài tập này so với các thuật toán khác?)
Trả lời: Thuật toán MLQ (với cơ chế Slot và không Feedback) được hiện thực trong bài tập này mang lại các ưu điểm sau:

1. **Đảm bảo tính ưu tiên (Priority Enforcement):** Scheduler luôn xem các hàng đợi có độ ưu tiên cao trước. Điều này rất quan trọng đối với các hệ thống cần phản hồi nhanh cho các tác vụ quan trọng hoặc tương tác người dùng.
2. **Chống Đói tài nguyên (Starvation Mitigation):** Khác với thuật toán lập lịch ưu tiên tuyệt đối (nơi tiến trình thấp có thể bị chặn vĩnh viễn), cơ chế `slot` buộc hàng đợi ưu tiên cao phải "nhường quyền" sau khi dùng hết định mức. Điều này đảm bảo các tiến trình ưu tiên thấp (như priority 139) vẫn có cơ hội được CPU phục vụ sau một khoảng thời gian nhất định. Tuy vậy, cơ chế chống đói này chỉ là cơ chế chống độc chiếm, khả năng chống đói còn cực kỳ hạn chế khi chỉ chống được việc 1 tiến trình làm treo hệ thống, nhưng không chống được việc 1 nhóm tiến trình ưu tiên cao cô lập nhóm ưu tiên thấp.
3. **Chi phí vận hành thấp (Low Overhead):** Do không cài đặt cơ chế phản hồi (di chuyển tiến trình giữa các hàng đợi hay tính toán lại độ ưu tiên động như trong MLFQ), thuật toán đơn giản hơn và tốn ít tài nguyên CPU cho việc quản lý hàng đợi.

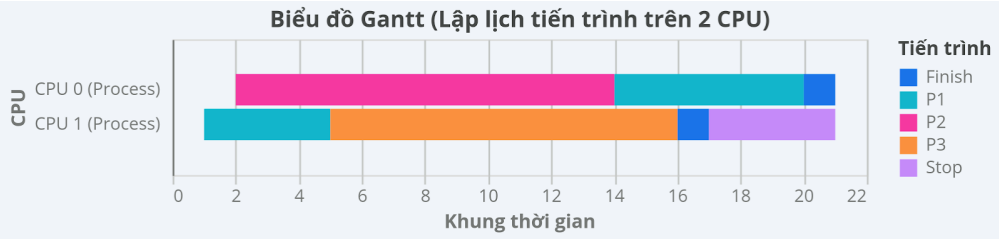
2.4 Kết quả Kiểm thử (Test Interpretation)

2.4.1 Testcase sched

2.4.1.1 Kịch bản kiểm thử (Scenario)

- **Cấu hình:** 2 CPU, Time Slice = 4.
- **Danh sách tiến trình:**
  - **p1s** (PID 1): Priority 1 (Thấp), đến tại T=0. Code size = 10.
  - **p2s** (PID 2): Priority 0 (Cao), đến tại T=1. Code size = 12.
  - **p3s** (PID 3): Priority 0 (Cao), đến tại T=2. Code size = 11.

2.4.1.2 Biểu đồ Gantt Dựa trên log kết quả chạy (`sched.output`), dưới đây là biểu đồ Gantt mô tả quá trình thực thi:



Từ biểu đồ và log hệ thống, chúng em rút ra các nhận xét sau:

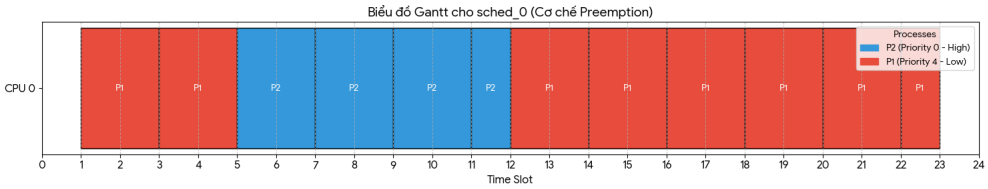
1. **Đa xử lý hiệu quả:** Cả hai CPU đều hoạt động song song từ $T = 2$ đến $T = 13$.
2. **Cơ chế MLQ hoạt động đúng:**
   - Tại thời điểm $T = 0$, `p1s` (Prio 1) được chạy vì hệ thống đang rảnh.
   - Tại thời điểm $T = 5$, `p1s` hết time slice và quay lại hàng đợi. Lúc này, `p3s` (Prio 0) cũng đang chờ.
   - Scheduler đã chọn `p3s` để chạy tiếp trên CPU 1 thay vì tiếp tục chạy `p1s`. Điều này chứng tỏ hàng đợi mức 0 được ưu tiên tuyệt đối hơn hàng đợi mức 1.
3. **Hiện tượng Starvation tạm thời:** Tiến trình `p1s` phải chờ từ $T = 5$ đến tận $T = 14$ (khi các tiến trình ưu tiên cao P2, P3 hoàn thành hoặc nhả CPU) mới được cấp phát lại tài nguyên. Đây là hành vi của thuật toán MLQ.

2.4.2 Testcase sched0

2.4.2.1 Kịch bản kiểm thử(Scenario)

- **Cấu hình :**
  - Time Slice (Lượng thời gian): 2 .
  - Số CPU: 1 (Single CPU).
- **Danh sách Tiến trình:**
  - P1 (s0 - PID 1):Thời điểm đến (Arrival): $T = 0$.Độ ưu tiên (Priority): 4 (Thấp). Số lượng lệnh: 15 lệnh (theo input/proc/s0).
  - P2 (s1 - PID 2):Thời điểm đến (Arrival): $T = 4$.Độ ưu tiên (Priority): 0 (Cao nhất).Số lượng lệnh: 7 lệnh .

2.4.2.2 Biểu đồ Gantt Dựa trên log kết quả chạy (`sched0.output`), dưới đây là biểu đồ Gantt mô tả quá trình thực thi:



Từ biểu đồ và log hệ thống, chúng em rút ra các nhận xét sau:

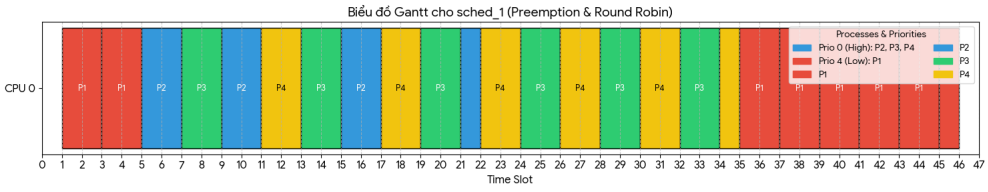
1. Test case sched_0 này là minh chứng rõ ràng nhất cho tính năng Fixed Priority Preemptive của bộ lập lịch
2. Priority Driven: P2 được ưu tiên tuyệt đối so với P1.
3. Preemption: P1 bị dừng ngay khi hết time slice để nhường cho P2 (tại T=5).
4. No Starvation (trong ngữ cảnh này): P1 chỉ bị hoãn tạm thời, sau khi P2 xong nó được chạy tiếp ngay.

2.4.3 Testcase sched1

2.4.3.1 Kịch bản kiểm thử(Scenario)

- **Cấu hình :**
  - 1 CPU, Time Slice = 2.
- **Danh sách Tiến trình:**
  - P1 (s0): Priority 4 (Thấp), đến T=0
  - P2 (s1): Priority 0 (Cao), đến T=4.
  - P3 (s2): Priority 0 (Cao), đến T=6.
  - P4 (s3): Priority 0 (Cao), đến T=7.

2.4.3.2 Biểu đồ Gantt Dựa trên log kết quả chạy (`sched1.output`), dưới đây là biểu đồ Gantt mô tả quá trình thực thi:



Từ biểu đồ và log hệ thống, chúng em rút ra các nhận xét sau:

1. **Cơ chế Round Robin:** Khi có nhiều tiến trình cùng độ ưu tiên cao nhất (P2, P3, P4 cùng Prio 0), Scheduler đã phân chia thời gian công bằng. Không có tiến trình nào được chạy liên tục quá time_slice (2 nhịp).
2. **Cơ chế Starvation (Tạm thời):** P1 (Priority 4) hoàn toàn không được cấp CPU trong suốt khoảng thời gian nhóm Prio 0 đang hoạt động (từ T=5 đến T=35). Đây là logic đúng của thuật toán MLQ ưu tiên cố định (Fixed Priority). P1 chỉ được chạy khi hệ thống "rảnh" (không còn tiến trình ưu tiên cao).

3 Tổng quan system call

System call là cơ chế cho phép chương trình người dùng yêu cầu dịch vụ từ nhân hệ điều hành. Mọi system call đều được định danh bằng một số hiệu (syscall number) và được xử lý hoàn toàn trong kernel space.

3.1 Cài đặt system call `sys_getprocinfo`

System call mới có số hiệu **441**, với tên là `getprocinfo`. Chức năng của system call này là truy vấn thông tin của tiến trình dựa trên PID được cung cấp từ chương trình người dùng.
Đăng ký system call System call được đăng ký trong file `syscall.tbl` và `syscalltbl.lst`:

```
441     getprocinfo sys_getprocinfo
```

3.1.1 Cài đặt trong kernel

Hàm `__sys_getprocinfo` được cài đặt trong file `sys_getprocinfo.c`. Hàm nhận PID thông qua thanh ghi `a1`, sau đó tìm kiếm tiến trình tương ứng trong các hàng đợi của kernel như ready queue, running list và MLQ queue (nếu có).
Việc truy cập PCB được thực hiện hoàn toàn trong kernel space, đảm bảo chương trình người dùng không thể truy cập trực tiếp vào cấu trúc dữ liệu nội bộ của hệ điều hành.
Nếu tiến trình tồn tại, độ ưu tiên của tiến trình sẽ được trả về thông qua thanh ghi `a1`. Ngược lại, nếu không tìm thấy, giá trị `-1` sẽ được trả về.

3.2 Chương trình kiểm thử

Ba chương trình người dùng được sử dụng để kiểm thử hệ thống:

3.2.1 Kiểm thử listsyscall (sc1)

Chương trình `sc1` gọi system call số 0 để liệt kê danh sách các system call hiện có trong hệ thống.

```
$ ./os os_syscall_list
Time slot   0
ld_routine
Time slot   1
Time slot   2
Time slot   3
Time slot   4
Time slot   5
Time slot   6
Time slot   7
Time slot   8
Time slot   9
        Loaded a process at input/proc/sc1, PID: 1 PRIO: 15
Time slot  10
        CPU 0: Dispatched process  1
0-sys_listsyscall
17-sys_memmap
441-sys_getprocinfo
Time slot  11
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Giải thích: Tại time slot 10, tiến trình `sc1` được CPU dispatch và thực hiện system call số 0. Kernel chuyển quyền điều khiển từ user space sang kernel space và gọi hàm `__sys_listsyscall`. Hàm này duyệt qua bảng `sys_call_table` và in ra tất cả các system call đã được đăng ký. Dòng `441-sys_getprocinfo` xuất hiện trong output chứng tỏ system call mới đã được thêm thành công vào bảng syscall và kernel nhận diện đúng.
Sau khi thực hiện xong system call, tiến trình kết thúc bình thường và CPU dừng hoạt động.

3.2.2 Kiểm thử memmap (sc2)

Chương trình `sc2` thực hiện các thao tác cấp phát bộ nhớ và ghi dữ liệu, nhằm kiểm tra system call quản lý bộ nhớ.

```
$ ./os os_syscall
Time slot   0
ld_routine
Time slot   1
Time slot   2
Time slot   3
Time slot   4
Time slot   5
Time slot   6
Time slot   7
Time slot   8
Time slot   9
        Loaded a process at input/proc/sc2, PID: 1 PRIO: 15
Time slot  10
        CPU 0: Dispatched process  1
liballoc:261
print_pgtbl:
 PDG=000077b520000bb0 P4g=000077b520001bc0 PUD=000077b520002bd0 PMD
    =000077b520003be0
Time slot  11
libwrite:490
print_pgtbl:
 PDG=000077b520000bb0 P4g=000077b520001bc0 PUD=000077b520002bd0 PMD
    =000077b520003be0
Time slot  12
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
libwrite:490
print_pgtbl:
 PDG=000077b520000bb0 P4g=000077b520001bc0 PUD=000077b520002bd0 PMD
    =000077b520003be0
Time slot  13
libwrite:490
print_pgtbl:
 PDG=000077b520000bb0 P4g=000077b520001bc0 PUD=000077b520002bd0 PMD
    =000077b520003be0
Time slot  14
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  15
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Giải thích: Sau khi được dispatch tại time slot 10, tiến trình `sc2` thực hiện các lời gọi system call liên quan đến quản lý bộ nhớ. Thông báo `liballoc` cho thấy kernel đã cấp phát vùng nhớ cho tiến trình.
Các dòng `print_pgtbl` hiển thị thông tin bảng trang (page table), bao gồm các cấp PDG, P4G, PUD và PMD. Điều này chứng minh rằng kernel đã thiết lập ánh xạ bộ nhớ ảo sang bộ nhớ vật lý thành công.
Trong các time slot tiếp theo, tiến trình thực hiện nhiều thao tác ghi dữ liệu (`libwrite`), dẫn đến việc bị đưa về hàng đợi sẵn sàng và được CPU dispatch lại. Việc này thể hiện cơ chế lập lịch hoạt động bình thường.

Cuối cùng, tiến trình kết thúc mà không phát sinh lỗi, cho thấy system call mới không ảnh hưởng đến các system call quản lý bộ nhớ hiện có.

3.2.3 Kiểm thử getprocinfo (sc3)

Chương trình `sc3` gọi system call số 440/441 để kích hoạt `sys_getprocinfo`. System call này không in kết quả ra màn hình mà trả giá trị thông qua thanh ghi, do đó chương trình kết thúc bình thường mà không có output hiển thị.

3.3 Kết quả thực nghiệm

Kết quả chạy các file cấu hình `os_syscall_list`, `os_syscall` và `os_sc` cho thấy:

- System call mới được đăng ký thành công
- Kernel xử lý đúng syscall number
- Không xảy ra lỗi hay treo hệ thống

```
$ ./os os_sc
Time slot   0
ld_routine
Time slot   1
Time slot   2
Time slot   3
Time slot   4
Time slot   5
Time slot   6
Time slot   7
Time slot   8
Time slot   9
        Loaded a process at input/proc/sc3, PID: 1 PRIO: 15
Time slot  10
        CPU 0: Dispatched process  1
Time slot  11
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Giải thích: Tại time slot 10, tiến trình `sc3` được CPU dispatch và thực hiện system call `sys_getprocinfo` với tham số PID truyền qua thanh ghi `a1`.
Kernel tiếp nhận yêu cầu và tìm kiếm tiến trình tương ứng trong các cấu trúc quản lý tiến trình như ready queue và running list. Nếu tiến trình tồn tại, thông tin độ ưu tiên sẽ được ghi lại vào thanh ghi `a1`.
Do chương trình người dùng không thực hiện lệnh xuất giá trị thanh ghi ra màn hình, nên kết quả không được hiển thị trong output. Tuy nhiên, việc tiến trình kết thúc bình thường chứng tỏ system call đã được thực thi thành công trong kernel space.

3.4 Kết luận

System call `sys_getprocinfo` đã được cài đặt thành công, đảm bảo nguyên tắc tách biệt User Space và Kernel Space. Việc kiểm thử thông qua các chương trình người dùng chứng minh system call hoạt động ổn định và đúng yêu cầu đề bài.

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

Do số lượng thanh ghi dùng để truyền tham số cho system call là có hạn, nên để truyền các tham số phức tạp (ví dụ như cấu trúc dữ liệu hoặc nhiều tham số cùng lúc), chương trình người dùng thường sử dụng cơ chế truyền con trỏ đến vùng nhớ.
Cụ thể, chương trình người dùng sẽ cấp phát một vùng nhớ trong user space, lưu toàn bộ dữ liệu cần truyền vào vùng nhớ này, sau đó truyền địa chỉ của vùng nhớ đó thông qua một thanh ghi khi gọi system call. Kernel sau khi nhận được địa chỉ sẽ sao chép dữ liệu từ user space sang kernel space để xử lý.
Cơ chế này giúp system call có thể xử lý các tham số phức tạp mà không bị giới hạn bởi số lượng thanh ghi, đồng thời vẫn đảm bảo nguyên tắc tách biệt giữa user space và kernel space.

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

Nếu việc thực thi một system call mất quá nhiều thời gian, tiến trình gọi system call sẽ bị giữ trong kernel space cho đến khi system call hoàn thành. Trong nhiều hệ điều hành, system call được thực thi theo cơ chế không bị ngắt (non-preemptive), do đó CPU sẽ bị chiếm giữ trong suốt thời gian xử lý.
Điều này có thể làm giảm khả năng đáp ứng của hệ thống. Vì vậy, trong thiết kế hệ điều hành, các system call thường được cài đặt ngắn gọn và tránh các thao tác tốn nhiều thời gian, hoặc được tách thành các cơ chế xử lý bất đồng bộ khi cần thiết.

4 Memory Management

4.1 Cơ sở lý thuyết

4.1.1 Ánh xạ bộ nhớ ảo của từng quá trình

1. **Tổng quan về Ánh xạ bộ nhớ (Memory mapping)**

   Trong kiến trúc của hệ điều hành mô phỏng, mỗi quá trình (process) sở hữu một không gian bộ nhớ ảo riêng biệt. Toàn bộ không gian bộ nhớ ảo này được quản lý bằng cấu trúc Memory mapping (được đại diện bởi struct mm_struct) thông qua các Khối điều khiển quá trình (PCB). Mỗi khối PCB (struct pcb_t) chứa một con trỏ trỏ đến struct mm_struct nhằm mục đích cốt lõi là định nghĩa không gian bộ nhớ riêng và quản lý các vùng nhớ ảo:

   - **Định nghĩa không gian bộ nhớ riêng:** Mỗi quá trình trong hệ thống đều có một không gian địa chỉ ảo độc lập (Con trỏ mm trỏ đến một cấu trúc mm_struct duy nhất của quá trình đó). Điều này đảm bảo bảng trang (pgd) và danh sách vùng nhớ (mmap) của các quá trình hoàn toàn tách biệt với nhau, ngăn chặn việc truy cập trái phép hoặc xung đột bộ nhớ giữa các quá trình.
   - **Quản lý các vùng nhớ ảo:** Thông qua mm_struct, quá trình quản lý danh sách các vùng nhớ ảo bằng con trỏ mmap. Con trỏ mmap là đầu mối của các danh sách liên kết vm_area_struct, nó giúp hệ thống xác định các phân đoạn của quá trình (heap, stack, code) và các giới hạn địa chỉ (vm_start, vm_end, sbrk) để kiểm tra segmentation fault trước khi thực hiện ánh xạ vật lý.

2. **Cấu trúc và quản lý vùng nhớ ảo (Virtual memory area)**

Trong hệ thống quản lý bộ nhớ, mỗi phân đoạn được quản lý bởi cấu trúc vm_area_struct. Cấu trúc này phân chia không gian địa chỉ thành 2 trạng thái riêng biệt thông qua các biến: vm_start, vm_end và sbrk.

- **Trạng thái giới hạn tĩnh:** quy định phạm vi tối đa của một vùng nhớ có thể có.
  - **vm_start:** Địa chỉ bắt đầu của một vùng nhớ ảo.
  - **vm_end:** Địa chỉ kết thúc của một vùng nhớ ảo. vm_end giúp xác định kích thước tối đa mà vùng nhớ này có thể mở rộng tới, ngăn chặn một quá trình truy cập hoặc cấp phát vùng nhớ vượt qua không gian địa chỉ cho phép.
- **Trạng thái giới hạn động:** quyết định trạng thái thực tế của bộ nhớ.
  - **Vùng khả dụng ([vm_start, sbrk]):** là vùng nhớ đã được cấp phát khung trang vật lý và sẵn sàng sử dụng. Mọi truy cập đọc/ghi vào vùng nhớ này đều hợp lệ.
  - **Vùng dự trữ ([sbrk, vm_end]):** vùng này thuộc không gian địa chỉ của quá trình nhưng chưa được sử dụng. Bất kỳ truy cập nào vào vùng này (mà chưa nới rộng sbrk) đều sẽ bị hệ thống coi là lỗi truy cập trái phép.

3. **Tổ chức bên trong vùng khả dụng**

Tong không gian bộ nhớ khả dụng, hệ thống sử dụng 2 thành phần chính để tối ưu hóa việc sử dụng lại bộ nhớ: Vùng nhớ cấp phát và Danh sách quản lý khoảng trống.

- **Vùng nhớ cấp phát (Memory region)**
  - **Cấu trúc dữ liệu:** Mỗi vùng nhớ cấp phát được biểu diễn bởi struct vm_rg_struct bao gồm:
    + **rg_start:** Địa chỉ ảo bắt đầu của khối nhớ.
    + **rg_end:** Địa chỉ ảo kết thúc của khối nhớ.
    + **rg_next:** Con trỏ liên kết, dùng để nối các vùng nhớ lại với nhau trong danh sách quản lý.
  - **Cơ chế quản lý:** Hệ thống sử dụng một mảng cố định `symrgtbl` với mỗi chỉ mục index của mảng tương ứng với các thanh ghi của quá trình ảo. Khi lệnh **ALLOC** được gọi với một chỉ số thanh ghi cụ thể, thông tin về vùng nhớ mới (rg_start, rg_end) được lưu trực tiếp vào vị trí tương ứng trong `symrgtbl`. Điều này cho phép truy xuất nhanh với thời gian O(1) khi cần đọc/ghi hoặc giải phóng vùng nhớ đó.
- **Danh sách quản lý khoảng trống**
  - Hệ thống duy trì một danh sách liên kết đơn (vm_freerg_list) chứa các vm_rg_struct đại diện cho các vùng nhớ không được sử dụng nằm xen kẽ giữa các vùng đang sử dụng .
  - Các vùng nhớ không được sử dụng này xuất hiện khi người dùng gọi lệnh **FREE** để trả lại bộ nhớ hoặc do việc cấp phát dư thừa khi nới rộng sbrk.

4. **Cơ chế định địa chỉ CPU**

- **Cơ chế định địa chỉ CPU 32-bit:** Là địa chỉ do CPU tạo ra để truy cập vào một vị trí bộ nhớ cụ thể. Hệ thống phân trang mô phỏng một CPU Bus 22-bit gồm cấu trúc địa chỉ được chia thành:

- **Page number (p) - 14-bit:** Được sử dụng làm chỉ mục để tra cứu trong bảng trang, bảng này lưu trữ địa chỉ cơ sở cho mỗi trang trong bộ nhớ vật lý.
- **Page offset (d) - 8 bit:** Được kết hợp với địa chỉ cơ sở để xác định địa chỉ bộ nhớ vật lý được gửi đến Memory Management Unit.

- **Cơ chế định địa chỉ CPU 64-bit:** Là mô hình mô phỏng kiến trúc x86-64 với phần mở rộng 5-level paging cho phép quản lý không gian địa chỉ ảo khổng lồ lên đến 128 petabytes. Trong đó, địa chỉ ảo được chia thành các phần như sau:
  - **Page Global Directory - Level 5 (PGD):** Bit 56-48.
  - **Page Level 4 Directory - Level 4 (P4D):** Bit 47-39.
  - **Page Upper Directory - Level 3 (PUD):** Bit 38-30.
  - **Page Middle Directory - Level 2 (PMD):** Bit 21-29.
  - **Page Table - Level 1 (PT):** Bit 20-12.
  - **Page Offset:** Bit 11-0.

4.1.2 Bộ nhớ vật lý của hệ thống

1. **Tổng quan về kiến trúc bộ nhớ vật lý Mô hình "Singleton Physical":** Hệ thống áp dụng mô hình quản lý phần cứng tập trung. Mặc dù mỗi quá trình trong hệ thống có một không gian địa chỉ (memory mapping) riêng biệt nhưng tất cả các ánh xạ này đều trỏ về một thiết bị vật lý duy nhất được cài đặt cho toàn bộ hệ thống. Điều này mô phỏng chính xác tính chất chia sẻ tài nguyên và nguyên lý cơ bản của quản lý bộ nhớ: Ánh xạ nhiều không gian địa chỉ vào một không gian địa chỉ vật lý chia sẻ duy nhất.

2. **Phân loại thiết bị nhớ (Memory device)**

   Hệ thống phân chia bộ nhờ thành 2 loại thiết bị: RAM và SWAP. Các thiết bị này đều có thể được triển khai trên cùng một thiết bị vật lý với các thiết lập khác nhau như trong mm-memphy.c.

   - **Bộ nhớ chính - RAM:** Đặc điểm quan kỹ thuật quan trọng nhất của RAM là khả năng truy cập ngẫu nhiên và được kết nối trực tiếp với bus địa chỉ của CPU. Nhớ kiến trúc này, CPU có thể thực hiện các lệnh máy để đọc/ghi dữ liệu trực tiếp lên RAM ngay lập tức mà không cần qua các bước đệm trung gian.
   - **Bộ nhớ thứ cấp - SWAP:** Do đặc tính truy cập tuần tự và không có khả năng kết nối trực tiếp với CPU, mọi thao tác đọc/ghi trên SWAP phải thông qua cơ chế chuyển dữ liệu vào RAM trước khi vào CPU để thực hiện.

   Để quản lý bộ nhớ vật lý (RAM và SWAP), hệ thống sử dụng hai cấu trúc chính là `struct framephy_struct` và `struct memphy_struct`. Trong đó, cấu trúc `framephy_struct` được dùng để lưu thông tin số hiệu khung trang (frame number - fpn), còn cấu trúc `memphy_struct` quản lý vùng lưu trữ (storage), kích thước bộ nhớ (maxsz), trường truy cập (rdmflg bao gồm 2 chế độ: chế độ truy cập ngẫu nhiên và chế độ truy cập tuần tự) và danh sách quản lý các khung trang (free_fp_list và used_fp_list).

4.1.3 Multi-level paging

Việc áp dụng thiết kế **phân trang đa cấp** (cụ thể là 5 cấp trong hệ thống mô phỏng này) mang lại những lợi ích thiết yếu sau so với phương pháp phân trang đơn cấp:

- **Tối ưu hóa đáng kể dung lượng lưu trữ bảng trang**: Đây là lợi ích quan trọng nhất. Thay vì phải cấp phát một bảng trang khổng lồ chứa tất cả các mục entry ngay từ đầu, hệ thống chỉ cần duy trì bảng cấp cao nhất PGD. Các bảng cấp thấp hơn P4D, PUD, PMD, PT chỉ được cấp phát khi và chỉ khi tiến trình thực sự sử dụng vùng địa chỉ đó. Ta có thể thực hiện phép tính đơn giản sau, dựa trên kết quả thống kê, tổng dung lượng bảng trang của toàn hệ thống chỉ khoảng 80KB. Nếu sử dụng phân trang đơn cấp cho không gian địa chỉ 64-bit, kích thước bảng trang sẽ lớn đến mức không thể lưu trữ nổi, cỡ vào hàng Petabytes. Thiết kế này giúp hệ thống tiết kiệm tài nguyên triệt để, đặc biệt với các tiến trình có không gian địa chỉ thưa.
- **Giải quyết vấn đề phân mảnh bộ nhớ**: Bảng trang đơn cấp yêu cầu một vùng nhớ vật lý liên tục rất lớn để lưu trữ. Việc tìm kiếm một vùng trống lớn liên tục trong RAM là rất khó khăn khi bộ nhớ bị phân mảnh. Phân trang đa cấp chia nhỏ bảng trang thành các bảng con có kích thước bằng đúng một khung trang. Hệ điều hành có thể lưu trữ các bảng con này rải rác ở bất kỳ đâu trong bộ nhớ vật lý, giúp tận dụng tối đa các "lỗ hổng" bộ nhớ và không yêu cầu vùng nhớ liên tục.
- **Khả năng mở rộng cho hệ thống 64-bit** Thiết kế đa cấp là giải pháp duy nhất khả thi để hỗ trợ không gian địa chỉ 64-bit. Với không gian địa chỉ ảo cực lớn (2^{64}), việc phân chia thành 5 cấp, như thiết kế trong file `src/mm64.c` đảm bảo rằng mỗi bảng trang thành phần luôn giữ kích thước nhỏ gọn (4KB), giúp việc quản lý, cấp phát và thu hồi bộ nhớ cho bảng trang trở nên linh hoạt và khả thi về mặt vật lý.

4.1.4 Cơ chế dịch địa chỉ 5 cấp

Trong hệ thống 64-bit hiện đại, không gian địa chỉ ảo quá lớn để có thể quản lý bằng bảng trang đơn cấp hoặc 2 cấp. Do đó, hệ thống sử dụng cơ chế **phân trang 5 cấp** (5-level paging) là tối ưu nhất. Một địa chỉ ảo 64-bit (thực tế sử dụng 57 bit thấp) được phân chia logic thành các phần như sau:

- **Bits 56-48 (9 bits):** Chỉ số **PGD** (Page Global Directory) - Cấp 5.
- **Bits 47-39 (9 bits):** Chỉ số **P4D** (Page Level-4 Directory) - Cấp 4.
- **Bits 38-30 (9 bits):** Chỉ số **PUD** (Page Upper Directory) - Cấp 3.
- **Bits 29-21 (9 bits):** Chỉ số **PMD** (Page Middle Directory) - Cấp 2.
- **Bits 20-12 (9 bits):** Chỉ số **PT** (Page Table) - Cấp 1.
- **Bits 11-0 (12 bits):** **Offset** (Độ dời) trong trang 4KB.

Quy trình dịch địa chỉ từ địa chỉ ảo sang địa chỉ vật lý được thực hiện bởi bộ quản lý bộ nhớ thông qua việc duyệt cây bảng trang như sau:

1. **CR3 → PGD:** Thanh ghi nền trỏ đến bảng PGD. Dùng 9 bit PGD Index để tìm entry trỏ đến bảng P4D.
2. **PGD → P4D:** Từ bảng P4D, dùng 9 bit P4D Index để tìm entry trỏ đến bảng PUD.
3. **P4D → PUD:** Từ bảng PUD, dùng 9 bit PUD Index để tìm entry trỏ đến bảng PMD.
4. **PUD → PMD:** Từ bảng PMD, dùng 9 bit PMD Index để tìm entry trỏ đến bảng PT.

5. **PMD → PT:** Từ bảng PT, dùng 9 bit PT Index để tìm entry chứa số khung trang vật lý PFN.
6. **Tạo PA:** Kết hợp PFN vừa tìm được với 12 bit Offset để tạo thành địa chỉ vật lý cuối cùng.

Mô hình minh họa quá trình dịch địa chỉ:

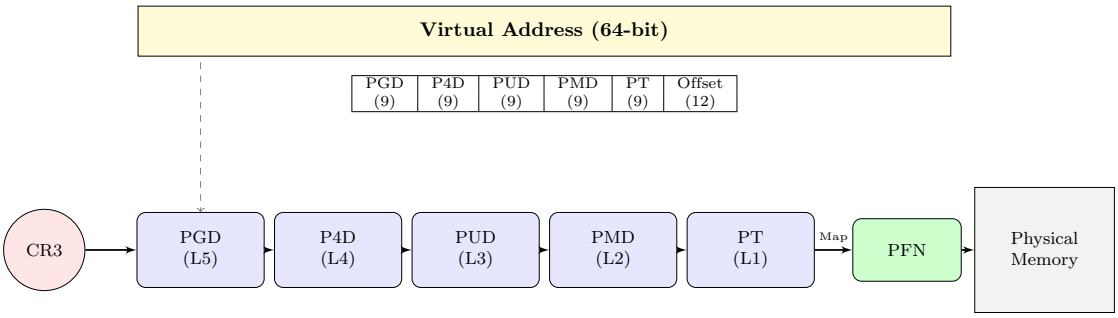


Figure 1: Sơ đồ dịch địa chỉ phân trang 5 cấp

4.2 Thiết kế và hiện thực hệ thống

Hệ thống quản lý bộ nhớ được xây dựng dựa trên kiến trúc phân tầng, tách biệt giữa quản lý bộ nhớ ảo và quản lý bảng trang vật lý. Dưới đây là phân tích chi tiết các thành phần cốt lõi.

4.2.1 Cấu trúc dữ liệu nền tảng

File `include/os-mm.h` đóng vai trò định nghĩa các cấu trúc dữ liệu xương sống cho hệ thống.

4.2.1.1 Cấu trúc quản lý bộ nhớ (`mm_struct`) Cấu trúc `mm_struct` được mở rộng để hỗ trợ chế độ 64-bit (`MM64`). Thay vì sử dụng mảng tĩnh, nhóm chúng em sử dụng các con trỏ để liên kết động các cấp bảng trang, hỗ trợ cơ chế cấp phát thưa.

Listing 1: Cấu trúc mm_struct trong os-mm.h

```
struct mm_struct {
#ifdef MM64
    /* Con tro den cac cap bang trang */
    uint64_t *pgd; // Level 5: Page Global Directory
    uint64_t *p4d; // Level 4: Page Level-4 Directory
    uint64_t *pud; // Level 3: Page Upper Directory
    uint64_t *pmd; // Level 2: Page Middle Directory
    uint64_t *pt;  // Level 1: Page Table

    /* Mang danh dau de thong ke dung luong su dung */
    uint8_t used_pgd[512]; // Danh dau cac entry PGD dang su dung
    uint8_t used_p4d[512];
    uint8_t used_pud[512];
    uint8_t used_pmd[512];
    uint8_t used_pt[512];

    /* Cac bien thong ke hieu nang */
```

```
    uint64_t access_counter; // Dem tong so lan truy cap
    uint64_t byte_counter;   // Dem tong dung luong bo nho danh cho bang
        trang
#endif
    // ...
};
```

Giải thích:

- Các con trỏ `pgd, p4d, pud, pmd, pt` đại diện cho địa chỉ cơ sở của các bảng trang tại mỗi cấp. Trong hiện thực, chúng trỏ đến các vùng nhớ được cấp phát động bởi `malloc`.
- Các mảng `used_*` và biến `byte_counter` được thêm vào sử dụng để thống kê và theo dõi dung lượng bộ nhớ. Mỗi khi một bảng trang mới được tạo, biến đếm sẽ tăng lên.

4.2.2 Hiện thực cơ chế phân trang 5 cấp

Toàn bộ logic xử lý phân trang 5 cấp nằm trong file source code `src/mm64.c`. Đây là module quan trọng nhất của hệ điều hành đơn giản này, nơi diễn ra mọi thao tác từ cấp thấp (thao tác bit trên PTE) đến cấp cao (ánh xạ vùng nhớ).

4.2.2.1 Cơ chế cấp phát - hàm `pteTravelling` Hàm `pteTravelling` đóng vai trò là thành phần cốt lõi trong phân hệ quản lý bộ nhớ ảo, chịu trách nhiệm điều phối toàn bộ quy trình duyệt cây phân trang. Nó chịu trách nhiệm duyệt qua 5 tầng bảng trang để tìm đến ô chứa dữ liệu PTE. Đặc biệt, hàm này hiện thực giải thuật **cấp phát theo nhu cầu**: chỉ cấp phát bộ nhớ cho các bảng trang con khi chúng thực sự cần thiết.

Listing 2: Chi tiết hàm pteTravelling và Thống kê

```
addr_t* pteTravelling(struct mm_struct *mm, addr_t pgn){
    addr_t pgd=0, p4d=0, pud=0, pmd=0, pt=0;
    addr_t *pgd_entry, *p4d_entry, *pud_entry, pmd_entry, *pt_entry;

    /* 1. Phan chia dia chi ao */
    get_pd_from_address(pgn, &pgd, &p4d, &pud, &pmd, &pt);

    /* 2. Duyet tu cap cao nhat PGD */
    addr_t *pgd_entry = (addr_t *)mm->pgd;

    /* 3. Logic cap phat cho P4D */
    if(pgd_entry[pgd] == 0){
      // Neu nhanh chua ton tai -> Cap phat dong bang P4D
      addr_t *p4d_new = malloc(sizeof(addr_t) * TABLE_SIZE_64);

      //Khoi tao bang moi bang 0 de tranh rac
      memset(p4d_new, 0, sizeof(addr_t) * TABLE_SIZE_64);

      // Lien ket PGD -> P4D
      pgd_entry[pgd] = (addr_t)p4d_new;

      // Thong ke: Tang dung luong bo nho su dung
      mm->byte_counter += sizeof(addr_t) * TABLE_SIZE_64;
    }

```

```
    // ... (Lap lai logic tuong tu cho PUD, PMD, PT) ...

    /* 4. Tra ve dia chi cua PTE cho cap cuoi cung */
    addr_t *pt_entry = (addr_t *)pmd_entry[pmd];
    return &pt_entry[pt];
}
```

Phân tích giải thuật:

- **Cấp phát động (`malloc`):** Cho phép các bảng trang nằm rải rác trong bộ nhớ, không cần một vùng nhớ liên tục khổng lồ.
- **Khởi tạo (`memset`):** Đảm bảo các entry mới sạch sẽ, tránh việc trỏ nhầm đến các vùng nhớ rác.
- **Thống kê (`byte_counter`):** Được cập nhật ngay khi `malloc` thành công, đảm bảo số liệu chính xác về dung lượng RAM tiêu tốn cho việc quản lý bảng trang.

4.2.3 Xử lý bảng trang và ánh xạ

Sau khi có cấu trúc bảng trang, hệ thống cần các hàm để thiết lập ánh xạ giữa trang ảo và khung vật lý.

4.2.3.1 Thiết lập PTE (`pte_set_fpn`) Hàm này được gọi khi hệ thống muốn gán một khung vật lý cho một trang ảo. Nó sử dụng `pteTravelling` để lấy vị trí ghi, sau đó sử dụng các macro thao tác bit để ghi thông tin.

Listing 3: Hàm pte_set_fpn

```
int pte_set_fpn(struct pcb_t *caller, addr_t pgn, addr_t fpn) {
  addr_t *pte;
  // Tim vi tri PTE tuong ung trong bang trang 5 cap
  pte = pteTravelling(caller->mm, pgn << PAGING64_ADDR_PT_SHIFT);

  // Ghi cac bit trang thai: PRESENT=1, SWAPPED=0
  SETBIT(*pte, PAGING_PTE_PRESENT_MASK);
  CLRBIT(*pte, PAGING_PTE_SWAPPED_MASK);

  // Ghi so khung vat ly FPN vao cac bit quy dinh (bit 12-51)
  SETVAL(*pte, fpn, PAGING_PTE_FPN_MASK, PAGING_PTE_FPN_LOBIT);

  return 0;
}
```

4.2.3.2 Ánh xạ dải trang (`vmap_page_range`) Đây là hàm cấp cao hơn, được sử dụng trong quá trình cấp phát bộ nhớ (`ALLOC`). Nó thực hiện vòng lặp để ánh xạ một danh sách các khung vật lý vào một dải các trang ảo liên tiếp.

Listing 4: Hàm vmap_page_range trong mm64.c

```
addr_t vmap_page_range(struct pcb_t *caller, addr_t addr, int pgnum,
                       struct framephy_struct *frames, struct
                           vm_rg_struct *ret_rg)
```

```
{
  struct framephy_struct *fpit = frames;
  addr_t pgn = addr >> PAGING64_ADDR_PT_SHIFT; // Lay so trang bat dau

  /* Duyet qua cac trang can anh xa */
  for (int i = 0; i < pgnum && fpit != NULL; i++) {
        // Thiet lap anh xa cho tung trang don le
        pte_set_fpn(caller, pgn, fpit->fpn);

        // Thong ke truy cap
        MultiLevelHelper(caller->mm, pgn);

        // Dua vao danh sach FIFO de quan ly thay the trang sau nay
        enlist_pgn_node(&caller->mm->fifo_pgn, pgn);

        pgn++;
        fpit = fpit->fp_next; // Chuyen sang khung tiep theo
  }
  return 0;
}
```

4.2.4 Truy xuất bộ nhớ ảo

File `src/mm-vm.c` chứa các hàm giao tiếp giữa CPU và hệ thống nhớ. Các hàm này mô phỏng việc CPU đọc/ghi dữ liệu thông qua địa chỉ ảo.

4.2.4.1 Đọc/Ghi dữ liệu (`pg_getval` / `pg_setval`) Khi CPU thực hiện lệnh `READ` hoặc `WRITE`, nó không truy cập trực tiếp RAM. Thay vào đó, nó gọi các hàm này để dịch địa chỉ ảo sang vật lý.

Listing 5: Quy trình truy xuất trong pg_getval

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

   // 1. Dich dia chi: Tim FPN tuong ung voi PGN
   // Ham pg_getpage se goi pteTravelling de tim trong bang trang
   if (pg_getpage(mm, pgn, &fpn, caller) != 0)
       return -1; // Loi: Trang khong ton tai hoac chua cap phat

   // 2. Tinh dia chi vat ly: (FPN * PageSize) + Offset
   int phyaddr = (fpn << PAGING_ADDR_FPN_LOBIT) + off;

   // 3. Truy cap RAM vat ly
   MEMPHY_read(caller->mram, phyaddr, data);
   return 0;
}
```

4.2.5 Thống kê và debug

Để kiểm chứng hệ thống hoạt động đúng và đo lường hiệu quả, hàm `print_pgtbl` trong `src/mm64.c` được sử dụng để in ra cấu trúc cây bảng trang hiện tại.

Listing 6: Hàm print_pgtbl

```
int print_pgtbl(struct pcb_t *caller, addr_t start, addr_t end)
{
  // Duyet qua toan bo khong gian dia chi da su dung
  for(addr_t addr = start; addr < end; addr += PAGING64_PAGESZ)
  {
      // In ra dia chi cua cac bang con (PGD, P4D...) tai moi muc
      printHelper(caller->mm, addr, printed_cache);
  }
  return 0;
}
```

Kết quả của hàm này trong các Testcase (ví dụ: `PDG=b52f...` `P4g=b52f...`) là bằng chứng xác thực nhất cho việc cấu trúc phân trang 5 cấp đã được hình thành và liên kết chính xác trong bộ nhớ.

4.2.5.1 Hàm thống kê: `MultiLevelHelper` Mỗi khi có một truy cập vào bảng trang (trong hàm `vmap_page_range`), hàm này được gọi để cập nhật số liệu thống kê.

Listing 7: Hàm MultiLevelHelper

```
void MultiLevelHelper(struct mm_struct *mm, addr_t pgn){
    addr_t pgd, p4d, pud, pmd, pt;
    get_pd_from_pagenum(pgn, &pgd, &p4d, &pud, &pmd, &pt);

    // Kiem tra va tinh dung luong cho tung cap bang trang
    // Neu bang tai index nay chua duoc tinh (used == 0)
    if(mm->used_pgd[pgd] == 0){
        mm->used_pgd[pgd] = 1;
        mm->byte_counter += sizeof(uint64_t); // Tang kich thuoc
    }
    // ... Tuong tu cho ca capp P4D, PUD, PMD, PT ...

    // Moi lan truy cap trang ton 5 lan truy cap bo nho
    mm->access_counter += 5;
}
```

Hàm này cho phép ta đo lường chính xác chi phí quản lý của bảng trang đa cấp. Nó chứng minh rằng mặc dù không gian địa chỉ rất lớn, lượng RAM thực tế tiêu tốn cho bảng trang là rất nhỏ nhờ cơ chế cấp phát thưa.

4.2.5.2 Hàm helper in bảng trang: `printHelper` Để debug, chúng ta cần nhìn thấy cấu trúc cây. Hàm này in ra địa chỉ của các bảng con tại một địa chỉ ảo cụ thể.

```
void printHelper(struct mm_struct* mm , addr_t addr, short* printed)
{
  // ... Trich xuat cac index ...

```

```
    // Duyet cay thu cong
    pgd_table = (addr_t *)mm->pgd;
    if(pgd_table[pgd_idx] == 0) return; // Nhanh cut, dung in

    p4d_table = (addr_t *)pgd_table[pgd_idx];
    // ... tiep tuc di xuong ...

    // In ra dia chi cua cac bang de kiem tra tinh lien ket
    printf(" PGD=%016llx P4D=%016llx PUD=%016llx PMD=%016llx\n",
           (uint64_t)pgd_table[pgd_idx], (uint64_t)p4d_table[p4d_idx], ...)
               ;
}
```

4.3 Trả lời câu hỏi

Câu hỏi: Trong hệ điều hành đơn giản này, chúng ta hiện thực thiết kế đa phân đoạn bộ nhớ hoặc các vùng bộ nhớ trong khai báo mã nguồn. Việc thiết kế đa phân đoạn có ưu điểm gì?

Trả lời: Trong kiến trúc của hệ điều hành mô phỏng, việc thiết kế và quản lý bộ nhớ theo mô hình đa phân đoạn hay quản lý theo các vùng nhớ ảo mang lại những ưu điểm như:

1. **Tăng cường bảo vệ bộ nhớ và kiểm soát quyền truy cập:** Việc chia không gian địa chỉ thành các phân đoạn riêng biệt như Text, Data, Heap, Stack cho phép hệ điều hành cài đặt các quyền hạn khác nhau cho từng vùng:
    - **Phân đoạn mã lệnh (Text segment):** Được thiết lập quyền Read-Only và Execute. Điều này ngăn chặn việc quá trình vô ý hoặc cố ý ghi đè dữ liệu lên mã lệnh, bảo vệ tính toàn vẹn của chương trình.
    - **Phân đoạn dữ liệu (Data/Heap/Stack):** Được thiết lập quyền Read-Write nhưng không được cấp quyền Execute. Điều này ngăn chặn việc Tràn bộ đệm và ngăn chặn việc chèn và chạy mã độc từ vùng nhớ dữ liệu.
2. **Tối ưu hóa quản lý không gian địa chỉ:** Thiết kế đa phân đoạn cho phép quản lý các vùng nhớ một cách linh hoạt và độc lập:
    - Việc tách rời các vùng không gian địa chỉ giúp tận dụng tối đa khoảng trống ở giữa không gian địa chỉ ảo, hạn chế hiện tượng phân mảnh và xung đột vùng nhớ.
    - Việc định nghĩa rõ ràng `vm_start` và `vm_end` cho từng vùng cho phép hệ điều hành dễ dàng phát hiện các lỗi truy cập trái phép khi con trỏ vượt quá giới hạn của phân đoạn cho phép.
3. **Hỗ trợ chia sẻ tài nguyên:** Mô hình phân đoạn tạo điều kiện thuận lợi cho việc chia sẻ bộ nhớ giữa các quá trình. Các quá trình chạy cùng một chương trình có thể chia sẻ chung phân đoạn mã lệnh trên bộ nhớ vật lý trong khi vẫn duy trì các phân đoạn dữ liệu (Data/Stack) riêng biệt.

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

Trả lời:

1. **Khả năng quản lý không gian địa chỉ khổng lồ**
   - Hệ thống 64-bit cung cấp một không gian địa chỉ ảo khổng lồ lên đến 2^64 địa chỉ. Nếu áp dụng phân trang 2 cấp cho hệ 64-bit, bảng trang cấp cao nhất (Outer Page Table) sẽ phải chứa hàng tỷ mục để bao quát không gian địa chỉ này. Điều này đòi hỏi hệ điều hành phải cấp phát một lượng lớn RAM liên tục, vượt quá khả năng đáp ứng của phần cứng thực tế ngay cả khi quá trình chỉ sử dụng vài KB bộ nhớ.
   - Việc mở rộng bảng trang thành nhiều cấp (cụ thể là 5 cấp trong `mm64.c`) chia nhỏ chuỗi địa chỉ ảo dài 64-bit thành các đoạn chỉ số ngắn hơn. Điều này giúp kích thước của mỗi bảng trang con được giữ ở mức nhỏ gọn, giúp việc định vị và ánh xạ trang trở nên khả thi.
2. **Tiết kiệm bộ nhớ lưu trữ bảng trang**
   - Không gian địa chỉ ảo của quá trình thương rất rộng nhưng dữ liệu thực tế lại nằm rải rác. Trong mô hình bảng trang phẳng, hệ thống phải tốn bộ nhớ để lưu trữ các entry cho cả những vùng địa chỉ chưa sử dụng.
   - Mô hình đa cấp khắc phục điều này bằng cấu trúc cây: nếu một nhánh địa chỉ chưa được dùng, con trỏ cấp cha sẽ trỏ về NULL. Hệ điều hành chỉ cần cấp phát RAM vật lý để lưu trữ các bảng trang con cho các vùng nhớ đang thực sự hoạt động.
3. **Thời gian truy cập bộ nhớ tăng lên**
   - Để đổi lấy sự tiết kiệm không gian, hệ thống phải hi sinh hiệu năng về thời gian. Để truy xuất một byte dữ liệu, phần cứng MMU không thể đi thẳng đến được khung trang (frame). Nó buộc phải thực hiện việc truy cập qua 5 cấp bảng trang một cách tuần tự từ PGD $\rightarrow$ P4D $\rightarrow$ PUD $\rightarrow$ PMD $\rightarrow$ PT rồi mới đến được bảng trang.
   - Với mô hình 5 cấp, mỗi lần truy xuất dữ liệu tốn 6 lần truy cập RAM (5 lần đọc bảng trang + 1 lần đọc dữ liệu). Hệ quả là độ phức tạp thời gian luôn là $O(k)$ với k = 5 cho mỗi trang dẫn đến tổng chi phí là $O(N)$ cho N trang truy cập.

Câu hỏi: *Những ưu điểm và nhược điểm của phân đoạn kết hợp phân trang là gì?*

Trả lời: Trong hệ điều hành mô phỏng này, kỹ thuật **segmentation with paging** (phân đoạn kết hợp phân trang) được triển khai nhằm tận dụng ưu điểm của cả hai phương pháp quản lý bộ nhớ. Phương pháp này chia không gian địa chỉ logic của một tiến trình thành các đoạn có độ rộng khác nhau, và gán từng trang của các đoạn đó vào các khung trang tương ứng trong bộ nhớ vật lý. Mô hình này được phản ánh qua cấu trúc `vm_area_struct` và `mm_struct` với bảng trang riêng cho từng tiến trình.

Ưu điểm:

- So với kỹ thuật phân trang thuần túy, phân đoạn kết hợp phân trang mang lại sự linh hoạt cao hơn trong việc kiểm soát cấp phát bộ nhớ. Mỗi chương trình trong hệ thống này có thể gồm nhiều đoạn với kích thước khác nhau. Kỹ thuật này cho phép chia chương trình thành các đơn vị logic như mã lệnh, dữ liệu, ngăn xếp, phù hợp với tư duy lập trình tự nhiên. Điều này giúp việc quản lý, gỡ lỗi dễ dàng hơn và mang lại sự linh hoạt cao trong kiểm soát cấp phát, giảm lãng phí tài nguyên so với cấp phát tĩnh.

- Việc phân trang bên trong mỗi phân đoạn giúp loại bỏ hiện tượng phân mảnh ngoài - một vấn đề lớn của phân đoạn thuần túy. Ngoài ra, kỹ thuật này hỗ trợ không gian địa chỉ thưa: chỉ những trang thực sự cần thiết mới được nạp vào RAM (swapping), cho phép hệ điều hành chạy các ứng dụng lớn hơn dung lượng vật lý thực tế.
- Phân đoạn kết hợp phân trang giúp bảo vệ bộ nhớ của từng chương trình khỏi sự can thiệp không mong muốn. Mỗi phân đoạn có thể được thiết lập các thuộc tính bảo vệ riêng biệt (như chỉ đọc, chỉ thực thi). Sự kết hợp này tạo ra lớp bảo vệ kép: kiểm soát ở cấp độ phân đoạn và cô lập chi tiết ở cấp độ trang, giúp tăng cường tính bảo mật và độ tin cậy của hệ thống, ngăn chặn các truy cập trái phép.
- Phân đoạn kết hợp phân trang cho phép các ứng dụng chia sẻ bộ nhớ. Việc cho phép nhiều chương trình cùng sử dụng một đoạn nhất định giúp chia sẻ dữ liệu hoặc mã nguồn, từ đó giảm sự dư thừa và tiết kiệm không gian bộ nhớ. Ngoài ra, cơ chế này còn hỗ trợ cấp phát bộ nhớ động, cho phép cấp phát hoặc thu hồi bộ nhớ khi cần thiết, từ đó nâng cao hiệu quả sử dụng.
- Phân đoạn kết hợp phân trang thường được sử dụng trong các hệ thống bộ nhớ ảo, cho phép hệ điều hành sử dụng nhiều bộ nhớ hơn so với dung lượng vật lý thực tế của thiết bị. Để thực hiện điều này, các đoạn dữ liệu và mã ít được sử dụng sẽ được lưu tạm thời trên ổ đĩa cứng và chỉ được nạp vào bộ nhớ khi thật sự cần.

Nhược điểm:

- Phân đoạn kết hợp phân trang có thể gây ra phân mảnh bộ nhớ, do không gian trống giữa các đoạn có thể phát sinh khi kích thước các đoạn khác nhau. Điều này dẫn đến việc sử dụng bộ nhớ không liền mạch và gây lãng phí tài nguyên. Mặc dù loại bỏ được phân mảnh ngoài, nhưng phân mảnh trong vẫn có thể xảy ra ở trang cuối cùng của mỗi phân đoạn (do kích thước đoạn không chia hết cho kích thước trang)
- So với phân trang hoặc phân đoạn thuần túy, quá trình dịch địa chỉ trở nên phức tạp hơn do phải đi qua hai bước trung gian: từ địa chỉ logic sang số hiệu đoạn và offset (qua bảng phân đoạn), rồi từ địa chỉ đó ánh xạ sang địa chỉ vật lý (qua bảng trang). Hệ thống bắt buộc phải sử dụng phần cứng chuyên dụng như TLB (Translation Lookaside Buffer) để giảm thiểu độ trễ này.
- Việc kết hợp cả hai cơ chế đòi hỏi hệ điều hành phải quản lý đồng thời cả bảng phân đoạn và nhiều bảng phân trang. Điều này làm tăng độ phức tạp trong thiết kế phần mềm và yêu cầu sự hỗ trợ phức tạp và tăng thêm chi phí từ phần cứng. Việc này có thể làm giảm hiệu suất hệ thống và tăng tổng chi phí vận hành.
- Phân đoạn kết hợp phân trang cũng có thể hạn chế lượng không gian địa chỉ mà một tiến trình có thể truy cập. Kích thước của bảng đoạn quyết định tổng bộ nhớ mà tiến trình có thể sử dụng, từ đó làm giới hạn kích thước tối đa của từng đoạn bộ nhớ.
- Việc duy trì các cấu trúc dữ liệu quản lý tiêu tốn đáng kể bộ nhớ. Mỗi tiến trình cần một bảng phân đoạn và có thể cần rất nhiều bảng phân trang, làm tăng lượng bộ nhớ bị chiếm dụng cho mục đích quản lý thay vì lưu trữ dữ liệu thực.

4.4 Kết quả hiện thực

Để kiểm chứng hệ thống phân trang đa cấp hoạt động đúng, nhóm chúng em sẽ phân tích chi tiết testcase `os_1_mlq_paging`.

4.4.1 Input: input/os_1_mlq_paging

```
2 4 8
1048576 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
```

Phân tích Input:

- **Cấu hình hệ thống:**
  - **Time slice:** 2 (Mỗi process được chạy tối đa 2 đơn vị thời gian trước khi bị ngắt).
  - **CPU:** 4 (Hệ thống mô phỏng 4 bộ xử lý đồng thời).
  - **Số lượng process:** 8 process sẽ được nạp.
- **Cấu hình bộ nhớ:**
  - **RAM:** 1,048,576 bytes (1 MB).
  - **Swap:** 16,777,216 bytes (16 MB).
- **Lịch trình nạp tiến trình:**

Time	Process name	PID	Priority
1	p0s	1	130
2	s3	2	39
4	m1s	3	15
6	s2	4	120
7	m0s	5	120
9	p1s	6	15
11	s0	7	38
16	s1	8	0

4.4.2 Giải thích kết quả

Time slot 0 - 2: Khởi động và nạp process

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

- **Time slot 0:** Hệ thống khởi động `ld_routine` để nạp các tiến trình từ file đầu vào.
- Process **PID 1** (`p0s`) và **PID 2** (`s3`) được nạp vào hàng đợi sẵn sàng.
- **Time slot 1:** Load process 1 từ `input/proc/p0s`. CPU 3 được dispatch để thực thi process PID 1.
- **Time slot 3:** Load process 2 từ `input/proc/s3` .CPU 2 được dispatch để thực thi process PID 2.

Time slot 3: Cấp phát bộ nhớ lần đầu cho PID 1

```
Time slot   3
liballoc:261
print_pgtbl:
PDG=00007f82f0000bb0 P4g=00007f82f0001bc0
PUD=00007f82f0002bd0 PMD=00007f82f0003be0
```

- **Sự kiện:** Process PID 1 thực hiện lệnh `alloc 300 0` (Cấp phát 300 bytes).
- **Hiện thực:** Hàm `liballoc` gọi `vm_map_ram`, kích hoạt quy trình ánh xạ bộ nhớ. Hàm `pteTravelling` trong `mm64.c` được gọi để duyệt cây bảng trang 5 cấp.
- **Kết quả Log:**

```
print_pgtbl:
 PDG=...0bb0 P4g=...1bc0 PUD=...2bd0 PMD=...3be0
```

- **Giải thích:** Đây là minh chứng cho cơ chế *cấp phát theo nhu cầu*. Ban đầu bảng trang rỗng. Khi cần cấp phát trang đầu tiên, hệ thống đã tự động cấp phát các bảng con từ pool bộ nhớ:
  - **PGD** (Cấp 5) tại `...0bb0` trỏ đến **P4D** (Cấp 4) tại `...1bc0`.
  - **P4D** trỏ đến **PUD** (Cấp 3) tại `...2bd0`.
  - **PUD** trỏ đến **PMD** (Cấp 2) tại `...3be0`.

  Việc các địa chỉ này khác nhau và liên kết với nhau chứng tỏ cấu trúc cây phân trang 5 cấp đã được hình thành chính xác trong bộ nhớ.

Time slot 4: Điều phối PID 1 và nạp PID 3

```
Time slot   4
CPU 3: Put process  1 to run queue
CPU 3: Dispatched process  1
liballoc:261
print_pgtbl:
PDG=00007f82f0000bb0 P4g=00007f82f0001bc0
PUD=00007f82f0002bd0 PMD=00007f82f0003be0
Loaded a process at input/proc/m1s, PID: 3 PRIO: 15
CPU 2: Put process  2 to run queue
CPU 2: Dispatched process  2
```

- Process PID 1 tiếp tục thực hiện `alloc 300 4`. Hàm `print_pgtbl` tiếp tục in ra cấu trúc bảng trang của process PID 1 (địa chỉ `PDG=...0bb0` không đổi), xác nhận tính ổn định của không gian địa chỉ.

- Process **PID 3** (`m1s`) được load vào hệ thống từ địa chỉ `input/proc/m1s`.

Time slot 5: Minh chứng cô lập bộ nhớ

```
Time slot   5
libfree:291
print_pgtbl:
PDG=00007f82f0000bb0 P4g=00007f82f0001bc0
PUD=00007f82f0002bd0 PMD=00007f82f0003be0
CPU 1: Dispatched process  3
liballoc:261
print_pgtbl:
PDG=00007f82f4000bb0 P4g=00007f82f4001bc0
PUD=00007f82f4002bd0 PMD=00007f82f4003be0
CPU 3: Put process  1 to run queue
CPU 3: Dispatched process  1
liballoc:261
print_pgtbl:
PDG=00007f82f0000bb0 P4g=00007f82f0001bc0
PUD=00007f82f0002bd0 PMD=00007f82f0003be0
liballoc:261
print_pgtbl:
PDG=00007f82f4000bb0 P4g=00007f82f4001bc0
PUD=00007f82f4002bd0 PMD=00007f82f4003be0
Loaded a process at input/proc/s2, PID: 4 PRIO: 120
```

- **Sự kiện:** Process PID 1 thực hiện `free 0`, process PID 3 được dispatch và thực hiện `alloc`.
- **Kết quả Log:**
  - Process PID 1 (trong lệnh free): `PDG=...f0000bb0`
  - Process PID 3 (trong lệnh alloc): `PDG=...f4000bb0` (theo log thực tế là `7f82f4000bb0`).
- **Giải thích:** Quan sát kỹ log, tại dòng lệnh của PID 3, địa chỉ PGD là `...f4000bb0`, khác hoàn toàn với `...f0000bb0` của process PID 1. Điều này chứng minh rằng hàm `loader` đã cấp phát một cấu trúc `mm_struct` và bảng trang vật lý riêng biệt cho mỗi tiến trình. Các tiến trình hoạt động độc lập và không can thiệp vào không gian nhớ của nhau.

Time slot 6: Truy xuất bộ nhớ

```
Time slot   6
libwrite:490
print_pgtbl:
PDG=00007f82f0000bb0 P4g=00007f82f0001bc0
PUD=00007f82f0002bd0 PMD=00007f82f0003be0
CPU 0: Dispatched process  4
CPU 2: Put process  2 to run queue
CPU 2: Dispatched process  2
```

- **Sự kiện:** Process PID 1 thực hiện lệnh `write 100 1 20`.
- **Hiện thực:** Hệ thống gọi hàm `pg_setval`. Để ghi dữ liệu, CPU cần dịch địa chỉ ảo sang địa chỉ vật lý.

- **Giải thích:** Hàm `print_pgtbl` xuất hiện xác nhận hệ thống đang duyệt lại cây bảng trang của PID 1 (`PDG=...0bb0`) để tìm khung trang vật lý. Việc ghi thành công cho thấy ánh xạ từ địa chỉ ảo sang địa chỉ vật lý là chính xác.

Time slot 7 - 8: Mở rộng không gian với PID 5

```
Time slot   7
CPU 1: Put process  3 to run queue
CPU 1: Dispatched process  3
libfree:291
print_pgtbl:
PDG=00007f82f4000bb0 P4g=00007f82f4001bc0
PUD=00007f82f4002bd0 PMD=00007f82f4003be0
Loaded a process at input/proc/m0s, PID: 5 PRIO: 120
CPU 3: Put process  1 to run queue
CPU 3: Dispatched process  5
Time slot   8
liballoc:261
print_pgtbl:
PDG=00007f82f0004c50 P4g=00007f82f0005c60
PUD=00007f82f0006c70 PMD=00007f82f0007c80
liballoc:261
print_pgtbl:
PDG=00007f82f4000bb0 P4g=00007f82f4001bc0
PUD=00007f82f4002bd0 PMD=00007f82f4003be0
liballoc:261
print_pgtbl:
PDG=00007f82f0004c50 P4g=00007f82f0005c60
PUD=00007f82f0006c70 PMD=00007f82f0007c80
```

- **Sự kiện:** Process **PID 5** (`m0s`) được nạp và thực hiện `alloc`.
- **Kết quả log:** Xuất hiện một chuỗi địa chỉ bảng trang mới: `PDG=...4c50`, `P4g=...5c60`...
- **Giải thích:** Hệ thống tiếp tục cấp phát không gian nhớ thứ 3 riêng biệt cho PID 5. Biến thống kê bộ nhớ (như `byte_counter` trong `mm_struct`) sẽ tăng lên tương ứng với kích thước của các bảng trang mới được tạo này.

Time slot 9 - 12: Hoạt động đa nhiệm

```
Time slot   9
CPU 2: Put process  2 to run queue
CPU 2: Dispatched process  2
CPU 1: Put process  3 to run queue
CPU 1: Dispatched process  3
libfree:291
print_pgtbl:
PDG=00007f82f4000bb0 P4g=00007f82f4001bc0
PUD=00007f82f4002bd0 PMD=00007f82f4003be0
CPU 0: Put process  4 to run queue
CPU 0: Dispatched process  4
Loaded a process at input/proc/p1s, PID: 6 PRIO: 15
CPU 3: Put process  5 to run queue
CPU 3: Dispatched process  6
```

```
libfree:291
print_pgtbl:
PDG=00007f82f4000bb0 P4g=00007f82f4001bc0
PUD=00007f82f4002bd0 PMD=00007f82f4003be0
Time slot  10
CPU 2: Put process  2 to run queue
CPU 2: Dispatched process  2
Time slot  11
CPU 1: Processed  3 has finished
CPU 1: Dispatched process  5
CPU 0: Put process  4 to run queue
CPU 0: Dispatched process  4
Loaded a process at input/proc/s0, PID: 7 PRIO: 38
libfree:291
print_pgtbl:
PDG=00007f82f0004c50 P4g=00007f82f0005c60
PUD=00007f82f0006c70 PMD=00007f82f0007c80
CPU 3: Put process  6 to run queue
CPU 3: Dispatched process  6
Time slot  12
liballoc:261
print_pgtbl:
PDG=00007f82f0004c50 P4g=00007f82f0005c60
PUD=00007f82f0006c70 PMD=00007f82f0007c80
CPU 2: Put process  2 to run queue
CPU 2: Dispatched process  7
```

- Các process PID 2, 3, 4, 6 liên tục được điều phối vào/ra.
- **Time slot 11:** Process PID 3 kết thúc. Tài nguyên của nó sẽ được hệ điều hành thu hồi (mặc dù trong mô phỏng đơn giản này, ta tập trung vào việc giải phóng cấu trúc PCB).
- **Time slot 12:** Process PID 7 (`s0`) thực hiện `alloc`. Log in ra `PDG=...4c50`

Time slot 13 - 22: Ổn định và truy xuất dữ liệu

```
Time slot  13
CPU 1: Put process  5 to run queue
CPU 1: Dispatched process  2
CPU 0: Put process  4 to run queue
CPU 0: Dispatched process  5
libwrite:490
print_pgtbl:
PDG=00007f82f0004c50 P4g=00007f82f0005c60
PUD=00007f82f0006c70 PMD=00007f82f0007c80
CPU 3: Put process  6 to run queue
CPU 3: Dispatched process  6
Time slot  14
CPU 1: Processed  2 has finished
CPU 1: Dispatched process  4
libwrite:490
print_pgtbl:
PDG=00007f82f0004c50 P4g=00007f82f0005c60
```

```
PUD=00007f82f0006c70 PMD=00007f82f0007c80
CPU 2: Put process  7 to run queue
CPU 2: Dispatched process  7
CPU 0: Processed  5 has finished
CPU 0: Dispatched process  1
libread:443
Time slot  15
CPU 3: Put process  6 to run queue
CPU 3: Dispatched process  6
libwrite:490
print_pgtbl:
PDG=00007f82f0000bb0 P4g=00007f82f0001bc0
PUD=00007f82f0002bd0 PMD=00007f82f0003be0
Time slot  16
CPU 1: Put process  4 to run queue
CPU 1: Dispatched process  4
Loaded a process at input/proc/s1, PID: 8 PRIO: 0
Time slot  17
CPU 2: Put process  7 to run queue
CPU 0: Put process  1 to run queue
CPU 0: Dispatched process  8
CPU 2: Dispatched process  7
CPU 3: Put process  6 to run queue
CPU 3: Dispatched process  6
Time slot  18
CPU 1: Put process  4 to run queue
CPU 1: Dispatched process  4
CPU 2: Put process  7 to run queue
CPU 2: Dispatched process  7
CPU 0: Put process  8 to run queue
CPU 0: Dispatched process  8
Time slot  19
CPU 3: Processed  6 has finished
CPU 3: Dispatched process  1
libread:443
Time slot  20
CPU 1: Processed  4 has finished
CPU 1 stopped
libwrite:490
print_pgtbl:
PDG=00007f82f0000bb0 P4g=00007f82f0001bc0
PUD=00007f82f0002bd0 PMD=00007f82f0003be0
CPU 0: Put process  8 to run queue
CPU 0: Dispatched process  8
Time slot  21
CPU 2: Put process  7 to run queue
CPU 2: Dispatched process  7
CPU 3: Put process  1 to run queue
CPU 3: Dispatched process  1
libread:443
Time slot  22
```

- Các lệnh `libwrite` và `libread` xuất hiện dày đặc.

- **Giải thích:** Tại mỗi lần truy cập bộ nhớ, hệ thống đều phải thực hiện quy trình dịch địa chỉ qua 5 cấp. Sự xuất hiện liên tục của log `print_pgtbl` với các địa chỉ PGD tương ứng với từng PID đang nắm giữ CPU (ví dụ PID 1 luôn là `...0bb0`) khẳng định tính nhất quán của dữ liệu.

Time slot 23 - 28: Giải phóng và kết thúc

```
Time slot  23
CPU 2: Put process  7 to run queue
CPU 2: Dispatched process  7
CPU 0: Put process  8 to run queue
CPU 0: Dispatched process  8
CPU 3: Put process  1 to run queue
CPU 3: Dispatched process  1
libfree:291
print_pgtbl:
PDG=00007f82f0000bb0 P4g=00007f82f0001bc0
PUD=00007f82f0002bd0 PMD=00007f82f0003be0
Time slot  24
CPU 0: Processed  8 has finished
CPU 0 stopped
Time slot  25
CPU 2: Put process  7 to run queue
CPU 2: Dispatched process  7
CPU 3: Processed  1 has finished
CPU 3 stopped
Time slot  26
Time slot  27
CPU 2: Put process  7 to run queue
CPU 2: Dispatched process  7
Time slot  28
CPU 2: Processed  7 has finished
CPU 2 stopped
```

- **Time slot 23:** Process PID 1 thực hiện `free 4`.
- **Kết quả log:** `libfree:291` in ra `PDG=...0bb0`.
- **Giải thích:** Khi hàm `__free` được gọi, vùng nhớ được đánh dấu là tự do. Tuy nhiên, cấu trúc bảng trang (các bảng PGD, P4D...) vẫn được giữ lại trong bộ nhớ (không bị `free` ngay lập tức) để tái sử dụng cho các lần cấp phát sau hoặc cho đến khi tiến trình kết thúc hẳn. Điều này giải thích vì sao log vẫn in ra địa chỉ PGD cũ.
- **Time slot 28:** Hầu hết các tiến trình đã hoàn thành và dừng lại.

```
================================================================================

                    SYSTEM-WIDE MEMORY STATISTICS REPORT
================================================================================

 1. Total Memory Access Times (All Processes): 65 accesses
 2. Total Page Table Storage Size            : 32768 bytes (32.00 KB)
================================================================================
```

Đầu tiên, em kiểm tra thủ công số lần access, với test này, trong file input, em thấy chỉ có p0s gọi những hàm cần truy cập bộ nhớ, p1s chỉ bao gồm lệnh call. p0s gồm 14 lệnh cụ thể như sau :

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

Ta nhận thấy process này gồm 3 hàm `alloc`, 3 hàm `write` và 3 hàm `read`, 3 hàm `alloc`. Ở 3 hàm `alloc` vì tổng dung lượng chỉ 700 (nhỏ hơn 1 frame vật lý) nên số lượt `access` chỉ là số lần `access` của 1 `alloc` tức 5 lần access, với những hàm như read/write sẽ tốn 10 `access` với 5 lần để tìm và 5 lần nữa để cập nhật dữ liệu. Như vậy ta có thể tính được tổng số `access` là 65 (khớp với output trên).
Về mặt bộ nhớ, với `p0s` dùng `alloc` nên sẽ phải tạo cả 5 cấp bảng, tương ứng tốn $5 \times 512 \times 8 = 20480$ byte, ta còn lại 3 process `p1s`, tuy những process này chỉ thực hiện lệnh `calc` nhưng vẫn cần cấp phát bảng PGD cho chúng, do vậy ta cần thêm $3 \times 512 \times 8 = 12288$ byte, như vậy ta cần tổng 32768 bytes để tạo bảng cho toàn test này.

5 Synchronization

5.1 Cơ sở lý thuyết

Chương trình hệ điều hành đơn giản nhóm được giao được mô phỏng luồng (thread) song song với 3 nhóm luồng chính (Timer, Loader, các CPU). Kiến trúc này dẫn đến nguy cơ xung đột khi truy cập các tài nguyên chia sẻ (shared resources) như hàng đợi tiến trình (Ready Queue) hay bộ nhớ vật lý (RAM). Do đó, sau khi đã hoàn thiện hai thành phần cốt lõi là Bộ lập lịch (Scheduler) và Quản lý bộ nhớ (Memory Management), bước cuối cùng để xây dựng một hệ điều hành mô phỏng hoàn chỉnh là tích hợp chúng hoạt động đồng thời nhằm giải quyết thách thức lớn nhất trong môi trường đa xử lý (Multi-CPU): vấn đề đồng bộ hóa (Synchronization) và tranh chấp tài nguyên (Race Condition)."

5.2 Những sửa đổi đã thực hiện

1. **Timer**

   Cách thực hiện:

   Đảo thứ tự: thực hiện `_time++` và `printf("Time slot")` **trước**, sau đó mới gọi `pthread_cond_signal()` để đánh thức CPU.

   Nguyên nhân áp dụng:

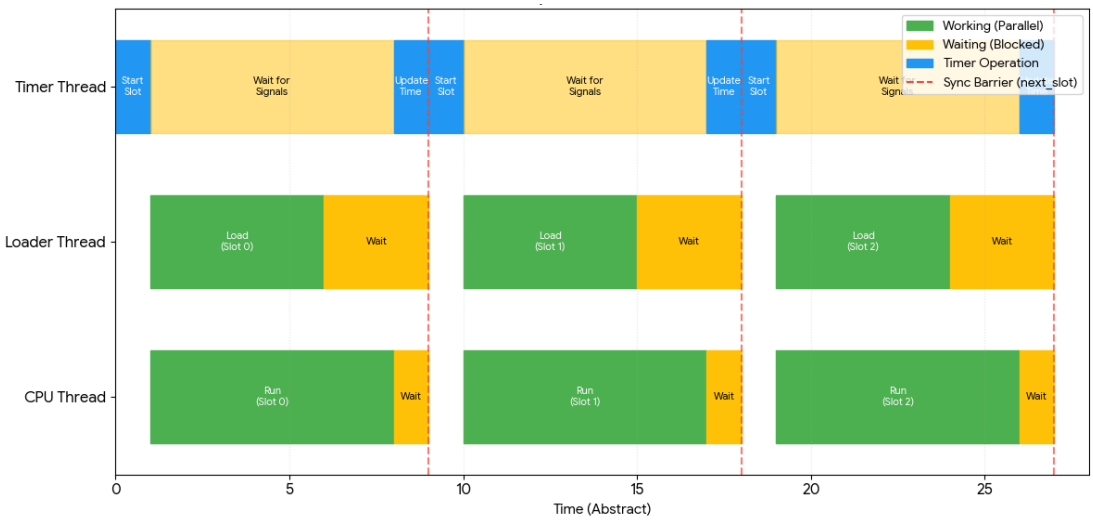


Figure 2: Mô hình hệ thống

- **Vấn đề gặp phải:** CPU chạy rất nhanh. Ngay khi Timer đánh thức CPU, CPU lập tức thức dậy và in ra kết quả (ví dụ: "Dispatched process 1"). Trong khi đó, Timer chưa kịp in dòng "Time slot 5". Kết quả là trên màn hình, dòng của CPU xuất hiện **trước** dòng Time slot.
- **Giải pháp:** Bắt buộc Timer phải in xong dòng "Time slot" trước khi đánh thức bất kỳ devices nào.

```
Time slot   8

    CPU 1: Put process  2 to run queue

    CPU 1: Dispatched process  4

Time slot   9

Time slot  10

Time slot  11

Time slot  12

Time slot  13
```

Ví dụ với test os_0_mlq_paging PID2 dispatched tại Slot 3, kết thúc: Put process tại Slot 8, hời gian chạy: Slot 3, 4, 5, 6, 7 → tổng cộng là 5 Slot. Time slice là 6, nhưng PID 2 mới chạy được 5 slot đã bị ngắt. Nó đáng lẽ phải chạy hết Slot 8 và bị ngắt ở Slot 9. Như vậy, Tại thời điểm chuyển giao sang Slot 9, Timer đánh thức CPU dậy để làm việc. CPU chạy quá nhanh, nó thức dậy và in dòng thông báo Put process (của Slot 9) mà lúc này Timer vẫn chưa kịp in dòng chữ Time slot 9.

Hậu quả: Dòng Put bị in chèn lên trước, nằm dưới Time slot 8, khiến bạn nhìn nhầm là Process bị ngắt ở Slot 8.

2. **Loader**

   Cách thực hiện:

   Trong file `src/os.c`, hàm `ld_routine`, di chuyển vòng lặp chờ thời gian lên **trước** lệnh load:

```
while (current_time() < start_time) // Chờ trước
    next_slot();
load(process);                      // Load sau khi đủ thời gian
```

   Nguyên nhân áp dụng:

   - **Vấn đề gặp phải:** Trong phiên bản code ban đầu, em thấy load(process) được gọi trước vòng while chờ đến time_slot, dẫn đến việc process sẽ được load mặc dù chưa đến thời điểm mà process đó cần được load.
   - **Giải pháp:** Bắt Loader phải chờ đến đúng thời điểm đã định (slot 5) rồi mới được load. Trong khi chờ, Loader gọi `next_slot()` và nhường CPU cho thread khác.

```
Time slot   0
ld_routine
    Loaded a process at input/proc/p0s, PID: 1 PRIO: 130
    CPU 3: Dispatched process  1
Time slot   1
liballoc:190
print_pgtbl:
 PDG=731860002220 P4g=731860003230 PUD=731860004240 PMD=731860005250
Time slot   2
    Loaded a process at input/proc/s3, PID: 2 PRIO: 39
    CPU 3: Put process  1 to run queue
    CPU 3: Dispatched process  1
liballoc:190
print_pgtbl:
 PDG=7318600078b0 P4g=7318600088c0 PUD=7318600098d0 PMD=73186000a8e0
    CPU 1: Dispatched process  2
```

   Ví dụ với test os_1_mlq_paging, ta nhận được output như trên, process 0 đúng ra phải được load ở slot1 nhưng lại load ở slot 0, và do được load sớm nên thứ tự thực hiện các hàm calc, alloc của process này cũng bị sai thứ tự.

3. **Scheduler và Memory**

   Cách thực hiện:

   Sử dụng khóa `pthread_mutex_t` để bảo vệ các vùng dữ liệu dùng chung:

   - **Tại Scheduler (`src/sched.c`):** Thêm biến `queue_lock`. Mỗi khi thao tác với hàng đợi ready queue (lấy process ra, đưa process vào), phải gọi `pthread_mutex_lock()` trước và `pthread_mutex_unlock()` sau.
   - **Tại Memory (`src/libmem.c`):** Thêm biến `mmvm_lock`. Mỗi khi cấp phát hoặc giải phóng bộ nhớ vật lý, phải khóa trước khi thao tác.

Nguyên nhân áp dụng:

- **Vấn đề gặp phải:** Hệ thống có nhiều CPU chạy song song. Giả sử tình huống: CPU 0 đang lấy process A ra khỏi hàng đợi, đúng lúc đó CPU 1 cũng đang lấy process. Nếu không có khóa, cả hai có thể lấy trùng một process.

  Tương tự với bộ nhớ: hai CPU cùng xin cấp phát RAM, có thể được cấp trùng một vùng nhớ, dẫn tới các thao tác sau này bị sai hành vi do ghi đè và double free. Khi một CPU đã giải phóng bộ nhớ của Process, CPU khác vẫn cố truy cập vào vùng nhớ đó để lấy PID, dẫn đến đọc ra dữ liệu rác hay khi nhiều CPU cùng chạy 1 Process, khi kết thúc, cả 3 đều gọi hàm free(). Lần free đầu tiên thành công, các lần sau sẽ gây crash vì vùng nhớ đó đã bị hủy rồi.

```
Time slot   0
ld_routine
Time slot   1
    Loaded a process at input/proc/p0s, PID: 1 PRIO: 130
Time slot   2
    CPU 2: Dispatched process  1
    CPU 3: Dispatched process  1 // both CPU 2 and 3 dispatch
        process 1
    ....
    Time slot   7
    CPU 3: Processed  3 has finished
    CPU 3: Dispatched process -283115515
    CPU 2: Processed  3 has finished
    CPU 1: Processed -283115515 has finished
    CPU 0: Processed -283115515 has finished
    CPU 0: Dispatched process -283115515
    CPU 1: Dispatched process -283115515
    CPU 2: Dispatched process -283115515
    Loaded a process at input/proc/m0s, PID: 5 PRIO: 120
Time slot   8
    CPU 2: Processed -283115515 has finished
    CPU 1: Processed -283115515 has finished
    CPU 3: Processed -283115515 has finished
wsl : free():

double free detected in tcache 2
At line:1 char:34
+ ... or ($i=101; $i -le 300; $i++) { wsl ./os os_1_mlq_paging
    2>&1 | Out-F ...
+
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          :
    NotSpecified: (free(): double free detected in
    tcache 2:String) [], RemoteException
    + FullyQualifiedErrorId : NativeCommandError

free(): double free detected in tcache 2
free(): double free detected in tcache 2
    CPU 0: Processed -283115515 has finished
free(): double free detected in tcache 2
```

4. (`src/libmem.c`)

   Cách thực hiện:

   Sử dụng biến khóa toàn cục `pthread_mutex_t mmvm_lock` (hoặc `mem_lock`) để bảo vệ tài nguyên vật lý dùng chung:

   - **Thực hiện:** Tại mỗi hàm giao tiếp chính (API) gồm: `alloc`, `free`, `read_proc`, `write_proc` gọi `pthread_mutex_lock(&mmvm_lock)` ở dòng đầu tiên để độc chiếm quyền truy cập. Sau khi thực hiện xong logic nghiệp vụ (gọi các hàm nội bộ như `__alloc`, `__free`), gọi `pthread_mutex_unlock(&mmvm_lock)` để trả tài nguyên trước khi `return`.

   Nguyên nhân áp dụng:

   - **Tranh chấp tài nguyên vật lý (Physical Memory Race):** RAM (`mram`) và Swap (`mswp`) là tài nguyên chia sẻ duy nhất. Nếu không có khóa, trường hợp hai CPU chạy song song cùng thực hiện `alloc` có thể dẫn đến việc cả hai cùng nhận được một chỉ số khung trang (frame index) giống nhau. Kết quả là Process A sẽ ghi đè dữ liệu lên vùng nhớ của Process B.
   - **Lỗi bộ nhớ (Memory Corruption):** Trong trường hợp giải phóng bộ nhớ (`free`), nếu thiếu đồng bộ, hai luồng có thể cùng thao tác lên danh sách liên kết các trang trống (`free_fp_list`), dẫn đến lỗi *Double Free* gây crash chương trình hoặc làm hỏng cấu trúc dữ liệu quản lý bộ nhớ.

5. **Đồng bộ hóa luồng xuất**

   Cách thực hiện:

   Thêm lệnh `fflush(stdout)` ngay sau mỗi lệnh `printf()` trong tất cả các file:

   Áp dụng cho: `timer.c`, `os.c` (cpu_routine, ld_routine), `libmem.c` (liballoc, libread, libwrite).

   Nguyên nhân áp dụng:

   - **Vấn đề gặp phải:** Hàm `printf()` không in ra màn hình ngay mà lưu vào bộ đệm (buffer) trước. Khi buffer đầy hoặc gặp ký tự xuống dòng mới in ra. Trong môi trường nhiều thread, thứ tự in ra màn hình có thể không đúng với thứ tự gọi `printf()`.

     Ví dụ: Thread A gọi printf trước, nhưng buffer chưa flush. Thread B gọi printf sau và flush luôn. Kết quả: output của B hiện trước A.
   - **Giải pháp:** Gọi `fflush(stdout)` ngay sau `printf()` để ép buffer in ra màn hình ngay lập tức, không chờ đợi.

5.3 Câu hỏi và trả lời

5.3.1 Câu hỏi

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.

5.3.2 Trả lời

Nếu hệ điều hành đơn giản không xử lý đồng bộ hóa, hệ thống sẽ rơi vào tình trạng không mong muốn race condition, data inconsistency hay deadlock. Điều này xảy ra khi nhiều luồng (CPU Threads, Timer, Loader) cùng truy cập và thay đổi tài nguyên chia sẻ (hàng đợi, bộ nhớ, biến thời gian) tại cùng một thời điểm mà không có sự kiểm soát, dẫn đến hành vi sai lệch, mất dữ liệu hoặc sập chương trình (Crash).

1. **Race Condition**
   - **Định nghĩa:** Race Condition là một tình huống lỗi xảy ra trong các hệ thống đa luồng (multithreaded) hoặc đa tiến trình, khi hai hoặc nhiều luồng cùng truy cập và thao tác trên một dữ liệu chia sẻ (shared resource) đồng thời mà không có cơ chế đồng bộ hóa phù hợp.
   - **Ví dụ thực tế (Scheduler):** CPU 0 và CPU 1 cùng kiểm tra hàng đợi vào cùng một thời điểm và đều thấy Process A. Do thiếu đồng bộ, cả hai cùng thực hiện lệnh `dequeue` đối với Process A.
   - **Hậu quả:** Lỗi **Double Dispatch** (hai CPU cùng chạy một process) dẫn đến lỗi nghiêm trọng **Double Free** khi process kết thúc (như log lỗi thực nghiệm: `free(): double free detected`).
2. **Data Inconsistency (Dữ liệu không nhất quán)**
   - **Định nghĩa :**Data Inconsistency là trạng thái mà dữ liệu được lưu trữ trong hệ thống bị sai lệch, mâu thuẫn hoặc bị hỏng (corrupted) so với logic thực tế.
   - **Ví dụ thực tế (Memory Allocation):** Tại `libmem`, hai process chạy song song cùng gọi hàm `alloc`. Do thiếu khóa tại `mram`, cả hai nhận được cùng một chỉ số khung trang vật lý (Physical Frame Index).
   - **Hậu quả:** Process đến sau ghi đè dữ liệu lên process đến trước. Biểu hiện trong thực nghiệm là cấu trúc PCB bị hỏng, khiến giá trị PID hiển thị thành số rác (ví dụ: `-283115515`).
3. **Deadlock (Khóa chết)**
   - **Định nghĩa :** Deadlock là tình trạng mà hai hay nhiều tiến trình trong hệ thống bị treo vĩnh viễn (blocked indefinitely), do mỗi tiến trình đều đang nắm giữ một tài nguyên và chờ đợi tài nguyên khác đang bị nắm giữ bởi tiến trình còn lại.
   - **Ví dụ lý thuyết (Tài nguyên chéo):** Process 1 giữ khóa quản lý RAM (`lock_A`) và chờ lấy khóa Swap (`lock_B`). Cùng lúc đó, Process 2 đang giữ khóa Swap (`lock_B`) và chờ lấy khóa RAM (`lock_A`).
   - **Hậu quả:** Tạo thành vòng tròn chờ đợi (Circular Wait), khiến cả hai process đứng yên vĩnh viễn và hệ thống ngừng hoạt động.