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

Report Assignment

Nhóm: 6

Operation System

Topic: LamiaAtrium

DANH SÁCH THÀNH VIÊN

STT	Họ và Tên	MSSV	Mô tả công việc	Đánh giá
1	Trương Hoàng Nam	2312201	Synchronization, Memory management, Paging	100%
2	Nguyễn Minh Kiên	2252395	Schedular	100%
3	Nguyễn Minh Nhật	2212397		0%
4	Nguyễn Công Vũ Hoàng	2211087		0%
5	Nguyễn Hoàng Anh	2310099		0%

Thành phố Hồ Chí Minh, Tháng 12

I Mở đầu

Trong bài tập lớn này, chúng em triển khai và mô phỏng một hệ điều hành đơn giản dựa trên bộ mã nguồn được cung cấp. Mục tiêu của bài làm là hiểu và xây dựng lại ba thành phần quan trọng của một hệ điều hành thực thụ:

1. **Scheduler** (Bộ lập lịch tiến trình)
2. **Synchronization** (Đồng bộ hoá trong môi trường đa CPU)
3. **Memory Management** – cơ chế cấp phát bộ nhớ và phân trang 32/64-bit (Paging)

Mặc dù đây không phải là một hệ điều hành đầy đủ, nhưng bài tập giúp chúng em hiểu được vai trò cốt lõi của OS:

- Quản lý **CPU ảo**, quyết định tiến trình nào được chạy.
- Quản lý **RAM ảo**, ánh xạ địa chỉ ảo sang địa chỉ vật lý.
- Đảm bảo các tiến trình không can thiệp lẫn nhau, ngay cả khi chạy trên nhiều CPU song song.

1 Mục tiêu phần Scheduler

Dựa theo mô tả trong tài liệu, hệ thống sử dụng cơ chế **MLQ** – **Multi-Level Queue**, trong đó mỗi mức ưu tiên có một hàng đợi riêng. Nhiệm vụ của nhóm bao gồm:

- Cài đặt `enqueue()` và `dequeue()` trong `queue.c`.
- Cài đặt `get_proc()` trong `sched.c` theo đúng chính sách MLQ.
- Kiểm tra hoạt động bằng Gantt chart và so sánh với output mẫu.

2 Mục tiêu phần Memory Management

Theo chương trình mô tả trong tài liệu, chúng em thực hiện:

- Xây dựng mô hình virtual memory dựa trên `vm_area` và `region`.
- Ánh xạ trang (paging) từ địa chỉ ảo sang địa chỉ vật lý.
- Xử lý các yêu cầu `alloc/free/read/write` theo cơ chế paging.
- Thực thi cơ chế swapping RAM $\leftrightarrow$ SWAP.
- Hỗ trợ **multi-level paging** khi bật chế độ MM64.

Hệ thống mô phỏng đầy đủ luồng xử lý như trong sơ đồ tổng quát: từ `libmem` $\rightarrow$ `mm-vm` $\rightarrow$ `mm` $\rightarrow$ `memphy`.

3 Mục tiêu phần System Call

Dựa theo chương trình mô tả trong PDF, nhóm thực hiện:

- Viết system call mới và thêm vào bảng syscall (`syscall.tbl`).
- Hiện thực hàm syscall handler trong kernel-mode.
- Kiểm thử bằng chương trình user-mode thông qua giao diện syscall.

Qua đó, chúng em hiểu rõ hơn cơ chế giao tiếp **user–kernel** và lý do tại sao tiến trình người dùng không thể truy cập trực tiếp vào cấu trúc PCB bên trong nhân hệ điều hành.

4 Kỳ vọng đạt được

Sau khi hoàn thành bài tập, sinh viên có khả năng:

- Hiểu rõ cách OS lập lịch tiến trình theo MLQ.
- Hiểu và triển khai mô hình bộ nhớ phân trang hiện đại.
- Nắm được sự khác biệt giữa user-space và kernel-space.
- Hiểu cơ chế gọi và xử lý system call.
- Nhận biết và xử lý các vấn đề đồng bộ như race-condition trong môi trường đa CPU.

Đây là những kỹ năng nền tảng quan trọng cho các môn học và lĩnh vực nâng cao như Operating System Internals, Virtualization, Kernel Programming và Computer Architecture.

II Cơ sở Lý thuyết và Kiến trúc Hệ thống

Hệ điều hành (Operating System - OS) đóng vai trò là phần mềm nền tảng quản lý tài nguyên phần cứng và cung cấp các dịch vụ trừu tượng hóa cho các chương trình ứng dụng. Trong đồ án này, chúng em được mô phỏng một hệ điều hành đơn giản (Simple OS) dựa trên kiến trúc nguyên khối (Monolithic Kernel), tập trung vào ba phân hệ quan trọng nhất: Lập lịch tiến trình (Scheduler), Đồng bộ hóa (Synchronization) và Quản lý bộ nhớ (Memory Management).

1 Hệ thống Lập lịch Tiến trình (CPU Scheduler)

Bộ lập lịch là thành phần quyết định tiến trình nào sẽ được cấp phát tài nguyên CPU để thực thi tại một thời điểm nhất định.

1.1 Cấu trúc Khối kiểm soát tiến trình (PCB)

Mỗi tiến trình trong hệ thống được đại diện bởi một cấu trúc dữ liệu gọi là PCB (*Process Control Block*). PCB chứa các thông tin thiết yếu:

- **PID (Process ID):** Định danh duy nhất của tiến trình.
- **Priority (Độ ưu tiên):** Giá trị số nguyên xác định tầm quan trọng của tiến trình. Trong hệ thống này, giá trị càng nhỏ, độ ưu tiên càng cao.
- **Code Segment & Program Counter (PC):** Lưu trữ mã lệnh và vị trí lệnh hiện tại đang thực thi.
- **Registers:** Tập hợp các thanh ghi ảo (10 thanh ghi) dùng để lưu trữ dữ liệu tạm thời trong quá trình tính toán.

1.2 Giải thuật Multi-Level Queue (MLQ)

Hệ thống sử dụng giải thuật Hàng đợi đa cấp (MLQ) để quản lý các tiến trình ở trạng thái sẵn sàng (Ready State).

- **Tổ chức:** Hệ thống duy trì một mảng các hàng đợi `mlq_ready_queue[MAX_PRIO]`. Mỗi phần tử là một hàng đợi FIFO.
- **Nguyên lý hoạt động:**
  1. Khi một tiến trình được nạp hoặc bị ngắt nhường CPU, nó được đưa vào hàng đợi tương ứng với độ ưu tiên của nó (Enqueue).
  2. Khi CPU nhàn rỗi, bộ lập lịch quét các hàng đợi từ mức ưu tiên cao nhất (0) đến thấp nhất (MAX_PRIO).
  3. Tiến trình ở đầu hàng đợi có độ ưu tiên cao nhất khác rỗng sẽ được chọn (Dequeue).
- **Tính chất Chiếm quyền (Preemptive):** Đây là đặc điểm quan trọng. Nếu một tiến trình có độ ưu tiên cao hơn xuất hiện (ví dụ: vừa được nạp từ Input), nó sẽ chiếm quyền CPU ngay tại Time Slot tiếp theo, đẩy tiến trình đang chạy (có độ ưu tiên thấp hơn) về lại hàng đợi.

2 Đồng bộ hóa trong Môi trường Đa xử lý

Hệ thống mô phỏng hoạt động trên cơ chế đa luồng (Multi-threading) của C để giả lập môi trường đa vi xử lý (Multi-core CPU). Điều này dẫn đến thách thức về tính toàn vẹn dữ liệu.

2.1 Vấn đề Tranh chấp (Race Condition)

Tranh chấp xảy ra khi nhiều CPU cùng truy cập và thay đổi một vùng nhớ chia sẻ mà không có cơ chế kiểm soát. Trong Simple OS, các vùng găng (Critical Sections) chính bao gồm:

- **Hàng đợi lập lịch:** Việc `enqueue` và `dequeue` làm thay đổi cấu trúc danh sách liên kết/mảng. Nếu hai CPU cùng thực hiện, danh sách có thể bị đứt gãy hoặc mất phần tử.
- **Danh sách khung trang (Free Frame List):** Khi cấp phát bộ nhớ, việc lấy một khung trang ra khỏi danh sách trống cần phải là nguyên tử (atomic).

2.2 Cơ chế Mutual Exclusion (Mutex)

Hệ thống sử dụng `pthread_mutex_t` để đảm bảo tính loại trừ tương hỗ.

- **Nguyên tắc:** Trước khi vào vùng găng, luồng phải gọi `lock()`. Nếu khóa đang đóng, luồng sẽ bị chặn (block) cho đến khi khóa mở. Sau khi thao tác xong, luồng gọi `unlock()`.
- **Áp dụng:** Biến `queue_lock` bảo vệ các thao tác trên Ready Queue. Biến `mmvm_lock` và `mem_lock` bảo vệ các thao tác trên bộ nhớ ảo và vật lý.

3 Quản lý Bộ nhớ và Cơ chế Phân trang

Hệ thống áp dụng kỹ thuật Phân trang (Paging) hiện đại, hỗ trợ không gian địa chỉ ảo 64-bit, mô phỏng theo kiến trúc của các CPU x86-64 ngày nay.

3.1 Hệ thống Phân trang 5 cấp (5-Level Paging)

Thay vì sử dụng một bảng trang phẳng khổng lồ (điều bất khả thi với không gian 2^{64}), hệ thống tổ chức bảng trang theo cấu trúc cây 5 tầng.

- **Cấu trúc:**
$$PGD \rightarrow P4D \rightarrow PUD \rightarrow PMD \rightarrow PT \rightarrow PhysicalFrame$$
- **Cơ chế dịch địa chỉ (Address Translation):** Địa chỉ ảo 64-bit được chia thành các trường (fields). 9 bit đầu làm chỉ số cho PGD, 9 bit tiếp theo cho P4D, v.v... và 12 bit cuối cùng là Offset (độ lệch) trong trang.
- **Lợi ích:**
  - **Tiết kiệm bộ nhớ:** Chỉ cấp phát các bảng con (sub-tables) khi vùng nhớ đó thực sự được sử dụng (Demand Paging).
  - **Hỗ trợ không gian lớn:** Cho phép quản lý không gian địa chỉ ảo lên tới 128 Petabytes.

3.2 Cấu trúc Bộ nhớ Ảo (VMA)

Không gian nhớ của mỗi tiến trình (‘mm_struct‘) được chia thành các khu vực ảo (‘vm_area_struct‘ - VMA).

- Mỗi VMA đại diện cho một đoạn logic như Heap, Stack, Code.
- Bên trong VMA, các biến được cấp phát dưới dạng Region (‘vm_rg_struct‘).
- Thiết kế này giúp hệ điều hành quản lý quyền truy cập (Read/Write/Execute) và thu hồi bộ nhớ một cách hiệu quả theo từng vùng.

3.3 Bộ nhớ Vật lý và Swapping

- **RAM:** Được chia thành các Frame có kích thước cố định (ví dụ 256 bytes). Trạng thái của Frame (rảnh/bận) được quản lý bởi danh sách liên kết.
- **Swap Device:** Khi RAM đầy, hệ thống thực hiện cơ chế hoán đổi (Swapping). Một trang ít sử dụng (Victim Page) sẽ được xác định, nội dung của nó được ghi ra thiết bị Swap (Backing Store) để giải phóng Frame cho trang mới cần dùng. Hệ thống sử dụng chiến lược FIFO (First-In, First-Out) đơn giản để chọn trang nạn nhân.

III Trả lời câu hỏi

1. **Cơ chế nào để truyền một tham số phức tạp cho system call khi số lượng thanh ghi có hạn?**
   Phân tích: Các kiến trúc CPU (như x86, ARM) chỉ dành một số lượng nhỏ thanh ghi (thường là 4-6) để truyền tham số cho hàm nhằm tối ưu tốc độ. Với các cấu trúc dữ liệu lớn (như một mảng 1MB hay một struct phức tạp), việc sao chép vào thanh ghi là bất khả thi.
   Giải pháp: Cơ chế **Truyền tham chiếu (Pass by Reference/Pointer)** được sử dụng.

   - **Bước 1 (User Space):** Chương trình cấp phát một vùng nhớ đệm (buffer) liên tục trong không gian của nó để chứa dữ liệu phức tạp.
   - **Bước 2 (Gọi Syscall):** Chương trình chỉ đặt **địa chỉ bắt đầu** (con trỏ) của vùng nhớ đệm đó vào một thanh ghi quy ước (ví dụ 'RDI' hoặc 'EBX').
   - **Bước 3 (Kernel Space):** Hệ điều hành đọc địa chỉ từ thanh ghi, sau đó sử dụng các hàm an toàn để truy cập dữ liệu tại địa chỉ đó.

2. **Điều gì xảy ra nếu việc thực thi system call tốn quá nhiều thời gian?**
   Phân tích: System call chạy trong chế độ đặc quyền (Kernel Mode).
   Hệ quả:

   - **Chặn CPU (CPU Blocking):** Nếu kernel không được thiết kế theo dạng Preemptive (có thể bị ngắt), tác vụ system call sẽ giữ CPU cho đến khi hoàn tất. Các tiến trình khác (bao gồm cả các tiến trình thời gian thực hoặc giao diện người dùng) sẽ bị "đóng băng" (starvation), dẫn đến hệ thống bị lag, không phản hồi.
   - **Ví dụ thực tế:** Một lệnh đọc đĩa (Disk I/O) bị lỗi phần cứng và kẹt trong vòng lặp vô hạn trong driver sẽ khiến toàn bộ hệ thống treo cứng.

3. **Ưu điểm của thuật toán lập lịch MLQ trong bài tập này so với các thuật toán khác?**
   So sánh chi tiết:

   - **So với FCFS (First-Come, First-Served):** MLQ giải quyết được vấn đề "Convey Effect" (Hiệu ứng đoàn tàu). Một tiến trình quan trọng không phải chờ đợi các tiến trình không quan trọng đến trước nó.
   - **So với Round Robin thuần túy:** Round Robin đối xử công bằng tuyệt đối, nhưng không tối ưu cho hệ thống cần độ phản hồi nhanh cho các tác vụ tương tác. MLQ cho phép phân lớp: lớp tương tác (Interactive) ưu tiên cao chạy nhanh, lớp tính toán nền (Batch) ưu tiên thấp chạy sau.
   - **Tính linh hoạt:** Cấu trúc dữ liệu mảng các hàng đợi giúp thao tác chọn tiến trình ($O(1)$ với số lượng mức ưu tiên cố định) nhanh hơn việc phải sắp xếp lại một hàng đợi duy nhất mỗi khi có tiến trình mới ($O(N \log N)$).

4. **Ưu điểm của thiết kế đa phân đoạn bộ nhớ (Multiple Segments)?**
   Phân tích thiết kế: Việc chia bộ nhớ thành `vm_area_struct` mang lại lợi ích về:

   - **Ngữ nghĩa (Semantics):** Phản ánh đúng cấu trúc chương trình. Ví dụ: Code Segment (chứa lệnh), Data Segment (biến toàn cục), Stack (biến cục bộ), Heap (cấp phát động).
   - **Bảo mật (Protection):** Hệ điều hành có thể gán quyền (Read/Write/Execute) cho từng vùng. Vùng Code thường là Read-Only/Execute để ngăn chặn tấn công Self-modifying code.
   - **Tối ưu hóa:** Có thể Swap-out toàn bộ một Segment không dùng đến một cách dễ dàng, hoặc chia sẻ Code Segment giữa nhiều tiến trình chạy cùng một chương trình (Shared Libraries).

5. **Tại sao cần 5 cấp trong phân trang? Điều gì xảy ra nếu chia nhiều hơn 2 cấp?**
   Bối cảnh: Không gian địa chỉ 64-bit thực tế thường dùng 48-bit hoặc 57-bit để định địa chỉ ảo.
   Phân tích:

   - **Vấn đề của 2 cấp:** Với không gian 64-bit, nếu chỉ dùng 2 cấp, mỗi bảng trang sẽ phải quản lý một vùng quá lớn, dẫn đến kích thước bảng trang cấp 1 khổng lồ, không thể chứa vừa trong RAM.

- **Giải pháp 5 cấp:** Chia nhỏ việc quản lý. Mỗi cấp chỉ quản lý một phần nhỏ (9 bit index = 512 mục).
- **Hệ quả:**
  - (+) **Tiết kiệm bộ nhớ:** Chỉ cấp phát các nhánh cây cần thiết (Sparse Address Space).
  - (-) **Hiệu năng:** CPU phải thực hiện 5 lần truy cập bộ nhớ (Memory Access) để lấy địa chỉ vật lý cho 1 lần truy cập dữ liệu. Điều này làm chậm tốc độ xử lý đáng kể. Giải pháp bắt buộc là phải có **TLB (Translation Lookaside Buffer)** để cache kết quả dịch.

6. **Minh họa vấn đề khi không có đồng bộ hóa (Synchronization)?**
   Ví dụ cụ thể trong bài tập:
   Xét hàm `MEMPHY_get_freefp` (Lấy khung trang trống):

```
int MEMPHY_get_freefp(struct memphy_struct *mp, addr_t *retfpn) {
    struct framephy_struct *fp = mp->free_fp_list;
    // ... (Race condition here)
    mp->free_fp_list = fp->fp_next;
    *retfpn = fp->fpn;
    return 0;
}
```

   Nếu 2 CPU cùng chạy dòng 1, cùng trỏ `fp` vào cùng một Frame (ví dụ Frame #5). Cả hai cùng thực hiện dòng cập nhật danh sách.
   ⇒ **Hậu quả:** Cả Process A và Process B đều nghĩ mình sở hữu Frame #5. Process A ghi dữ liệu của nó, Process B ghi đè lên. Dữ liệu của A bị mất vĩnh viễn, dẫn đến tính toán sai hoặc crash chương trình.

IV Phân tích Kết quả Thực nghiệm

Phần này phân tích chi tiết log hoạt động của hệ thống qua các kịch bản kiểm thử (Test Cases), đối chiếu giữa Input đầu vào và Output nhận được để kiểm chứng tính đúng đắn của các module.

1 Phân tích Lập lịch (Scheduler Analysis)

Hệ thống lập lịch được kiểm thử qua 3 kịch bản khác nhau để chứng minh các đặc tính cốt lõi của giải thuật Multi-Level Queue (MLQ). Dưới đây là phân tích chi tiết dựa trên log thực thi thực tế.

1.1 Kịch bản 1: Cơ chế Chiếm quyền (Preemptive Priority) - `sched_0`

Mục tiêu: Kiểm chứng khả năng ngắt một tiến trình đang chạy khi có tiến trình khác quan trọng hơn xuất hiện.

```
namtrhcmut@hoangnamtruong:/mnt/c/BKL/Nam3_HK1/HDH/ossim_lamiaatrium/ossim_lamiaatrium$ ./os sched_0
Time slot   0
ld_routine
        Loaded a process at input/proc/s0, PID: 1 PRIO: 4
Time slot   1
        CPU 0: Dispatched process  1
Time slot   2
Time slot   3
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   4
        Loaded a process at input/proc/s1, PID: 2 PRIO: 0
Time slot   5
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
Time slot   6
Time slot   7
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot   8
Time slot   9
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot  10
Time slot  11
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot  12
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  1
Time slot  13
Time slot  14
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  15
Time slot  16
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  17
Time slot  18
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  19
Time slot  20
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  21
Time slot  22
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  23
        CPU 0: Processed  1 has finished
        CPU 0 stopped

MEMORY CONTENT:
```

Figure 1: *Log thực thi kịch bản sched_0 (Preemption)*

Phân tích chi tiết (Hình 1):

- **Giai đoạn 1 ($t = 0 \to 3$):** Hệ thống chỉ có tiến trình `s0` (PID 1) với độ ưu tiên thấp (Prio 4). CPU 0 thực thi tiến trình này bình thường.
- **Điểm gãy ($t = 4 \to 5$):** Tại Time slot 4, tiến trình `s1` (PID 2) được nạp vào với độ ưu tiên cao nhất (Prio 0). Ngay lập tức tại Time slot 5, log ghi nhận: *"CPU 0: Put process 1 to run queue"* và

"CPU 0: Dispatched process 2".
→ **Nhận xét:** Bộ lập lịch đã phát hiện sự xuất hiện của tiến trình ưu tiên cao hơn. Nó không đợi `s0` hết thời gian (time slice) mà thực hiện **cưỡng chế thu hồi CPU** (Forced Preemption) để cấp cho `s1`.

- **Giai đoạn 2 ($t = 5 \rightarrow 12$):** `s1` chạy liên tục cho đến khi hoàn thành (*Processed 2 has finished*).
- **Giai đoạn 3 ($t = 13$):** Sau khi `s1` kết thúc, CPU mới quay lại lấy `s0` từ hàng đợi để tiếp tục xử lý.

1.2 Kịch bản 2: Cơ chế Luân phiên (Round Robin) - `sched_1`

Mục tiêu: Kiểm chứng tính công bằng khi nhiều tiến trình có cùng độ ưu tiên cao nhất cùng tồn tại.

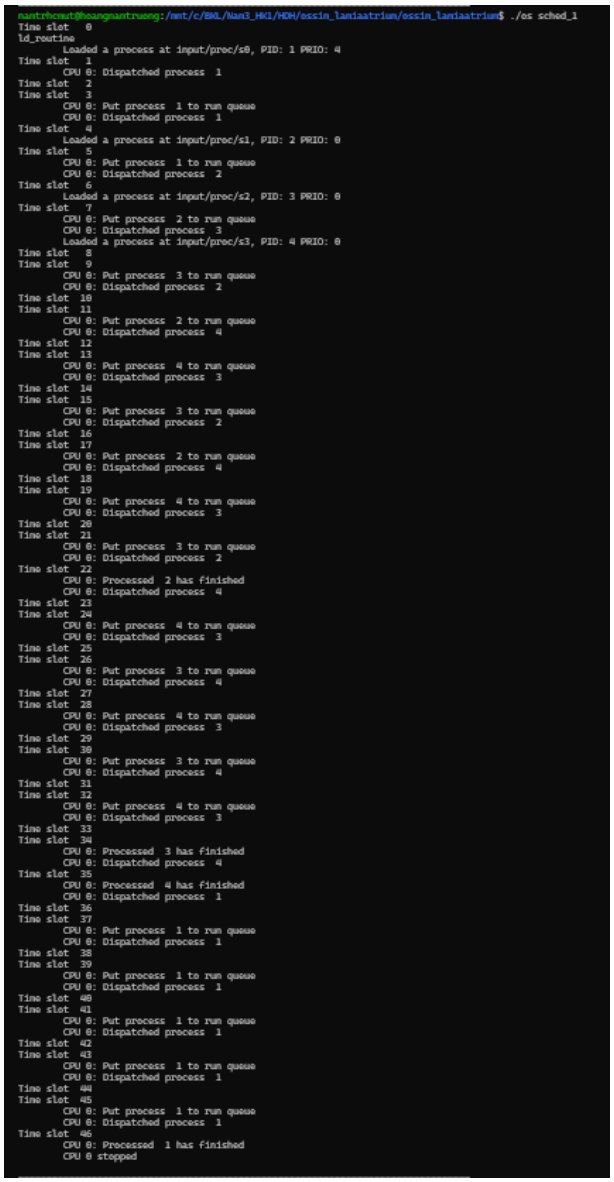

Figure 2: *Log thực thi kịch bản sched_1 (Round Robin)*

Phân tích chi tiết (Hình 2):

- **Bối cảnh:** Từ Time slot 7, trong hệ thống tồn tại đồng thời `s2` (PID 3) và `s3` (PID 4). Cả hai đều có độ ưu tiên cao nhất (Prio 0).
- **Diễn biến luân phiên:**

- Slot 9: *Dispatched process 2* (tức `s1` cũ/hoặc tiến trình ưu tiên trước đó).
- Slot 11: *Put process 2... Dispatched process 4* (CPU chuyển sang PID 4).
- Slot 13: *Put process 4... Dispatched process 3* (CPU chuyển sang PID 3).
- Slot 15: *Put process 3... Dispatched process 2* (Quay lại PID 2).

- **Nhận xét:** Hệ thống không cho phép bất kỳ tiến trình nào độc chiếm CPU. Mỗi tiến trình chỉ được chạy trong một khoảng *Time Slice* (trong cấu hình là 2 slot), sau đó bị đẩy xuống cuối hàng đợi (*Put process... to run queue*) để nhường lượt cho tiến trình tiếp theo. Đây chính là cơ chế **Round Robin** đảm bảo sự công bằng (Fairness).

1.3 Kịch bản 3: Đa xử lý song song (Parallelism) - `sched`

Mục tiêu: Kiểm chứng khả năng tận dụng đa lõi (2 CPU) của bộ lập lịch.

```
namtrhcmut@hoangnamtruong:/mnt/c/BKL/Nam3_HK1/HDH/ossim_lamiaatrium/ossim_lamiaatrium$ ./os sched
Time slot   0
ld_routine
        Loaded a process at input/proc/p1s, PID: 1 PRIO: 1
Time slot   1
        CPU 1: Dispatched process  1
        Loaded a process at input/proc/p2s, PID: 2 PRIO: 0
Time slot   2
        CPU 0: Dispatched process  2
        Loaded a process at input/proc/p3s, PID: 3 PRIO: 0
Time slot   3
Time slot   4
Time slot   5
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  3
Time slot   6
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot   7
Time slot   8
Time slot   9
        CPU 1: Put process  3 to run queue
        CPU 1: Dispatched process  3
Time slot  10
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot  11
Time slot  12
Time slot  13
        CPU 1: Put process  3 to run queue
        CPU 1: Dispatched process  3
Time slot  14
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  1
Time slot  15
Time slot  16
        CPU 1: Processed  3 has finished
        CPU 1 stopped
Time slot  17
Time slot  18
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  19
Time slot  20
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Figure 3: *Log thực thi kịch bản sched (Multi-CPU)*

Phân tích chi tiết (Hình 3):

- **Khởi tạo:**
  - Slot 1: CPU 1 xử lý Process 1 (`p1s`, Prio 1).
  - Slot 2: CPU 0 xử lý Process 2 (`p2s`, Prio 0).
- **Song song thực sự:** Tại **Time slot 5** và **6**, ta thấy cả hai dòng log xuất hiện cùng lúc:

```
CPU 1: Dispatched process 3
CPU 0: Dispatched process 2
```

- **Cạnh tranh tài nguyên:** Tại Slot 5, CPU 1 đang chạy Process 1 (Prio 1) thì Process 3 (Prio 0) xuất hiện. Do Process 3 có độ ưu tiên cao hơn, CPU 1 đã thực hiện Preemption: *"Put process 1 to run queue"* và chuyển sang chạy Process 3. Trong khi đó, CPU 0 vẫn tiếp tục chạy Process 2 (cũng là Prio 0).
- **Nhận xét:** Hệ thống đã phân phối tải hiệu quả trên cả 2 CPU. Các CPU hoạt động độc lập nhưng tuân thủ chung một chính sách lập lịch toàn cục (Global Scheduling Policy), đảm bảo các tiến trình quan trọng nhất (Prio 0) luôn được phục vụ bởi bất kỳ CPU nào đang rảnh hoặc đang chạy tác vụ kém quan trọng hơn.

2 Phân tích Quản lý Bộ nhớ (Memory Management Analysis)

Trong phần này, chúng ta phân tích hành vi của bộ nhớ ảo. Một điểm đặc biệt trong thiết kế của hệ thống mô phỏng này là việc áp dụng mô hình **Unified Kernel Memory Management**. Tất cả các tiến trình người dùng chia sẻ chung một cấu trúc quản lý bộ nhớ (‘mm_struct‘) và bảng trang gốc (‘pgd‘) của Kernel để tối ưu hóa tài nguyên mô phỏng. Do đó, trong các log bên dưới, địa chỉ cơ sở của bảng trang (`PDG`) là bất biến.

2.1 Kịch bản 1: Cấp phát và Phân trang cơ bản (`os_0_mlq_paging`)

Input: Các tiến trình thực hiện `alloc`, `free`, `read`, `write`.

```
namtrhcmut@hoangnamtruong:/mnt/c/BKL/Nam3_HK1/HDH/ossim_lamiaatrium/ossim_lamiaatrium$ ./os os_0_mlq_paging
Time slot   0
ld_routine
        Loaded a process at input/proc/p0s, PID: 1 PRIO: 0
Time slot   1
        CPU 0: Dispatched process  1
Time slot   2
liballoc:185
print_pgtbl:
 PDG=761b500030f0 P4g=761b50003000 PUD=761b50003010 PMD=761b50003020
        Loaded a process at input/proc/p1s, PID: 2 PRIO: 15
Time slot   3
        CPU 1: Dispatched process  2
liballoc:185
print_pgtbl:
 PDG=761b500030f0 P4g=761b50003000 PUD=761b50003010 PMD=761b50003020
Time slot   4
libfree:213
print_pgtbl:
 PDG=761b500030f0 P4g=761b50003000 PUD=761b50003010 PMD=761b50003020
        Loaded a process at input/proc/p1s, PID: 3 PRIO: 0
Time slot   5
liballoc:185
print_pgtbl:
 PDG=761b500030f0 P4g=761b50003000 PUD=761b50003010 PMD=761b50003020
Time slot   6
libwrite:431
print_pgtbl:
 PDG=761b500030f0 P4g=761b50003000 PUD=761b50003010 PMD=761b50003020
        Loaded a process at input/proc/p1s, PID: 4 PRIO: 0
Time slot   7
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  3
Time slot   8
Time slot   9
        CPU 1: Put process  2 to run queue
        CPU 1: Dispatched process  4
Time slot  10
Time slot  11
Time slot  12
Time slot  13
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  1
libread:378
print_pgtbl:
 PDG=761b500030f0 P4g=761b50003000 PUD=761b50003010 PMD=761b50003020
```

Figure 4: *Log hoạt động của testcase os_0_mlq_paging*

Phân tích Log (Hình 4):

- Tại Time slot 2, lệnh `liballoc` được gọi. Hệ thống in ra cấu trúc bảng trang:

```
PDG=761b500030f0 P4g=761b50003000 PUD=761b50003010 PMD=761b50003020
```

Các giá trị địa chỉ (cách nhau khoảng 16 bytes theo format in ấn) cho thấy hệ thống đã khởi tạo thành công cây bảng trang 5 cấp.

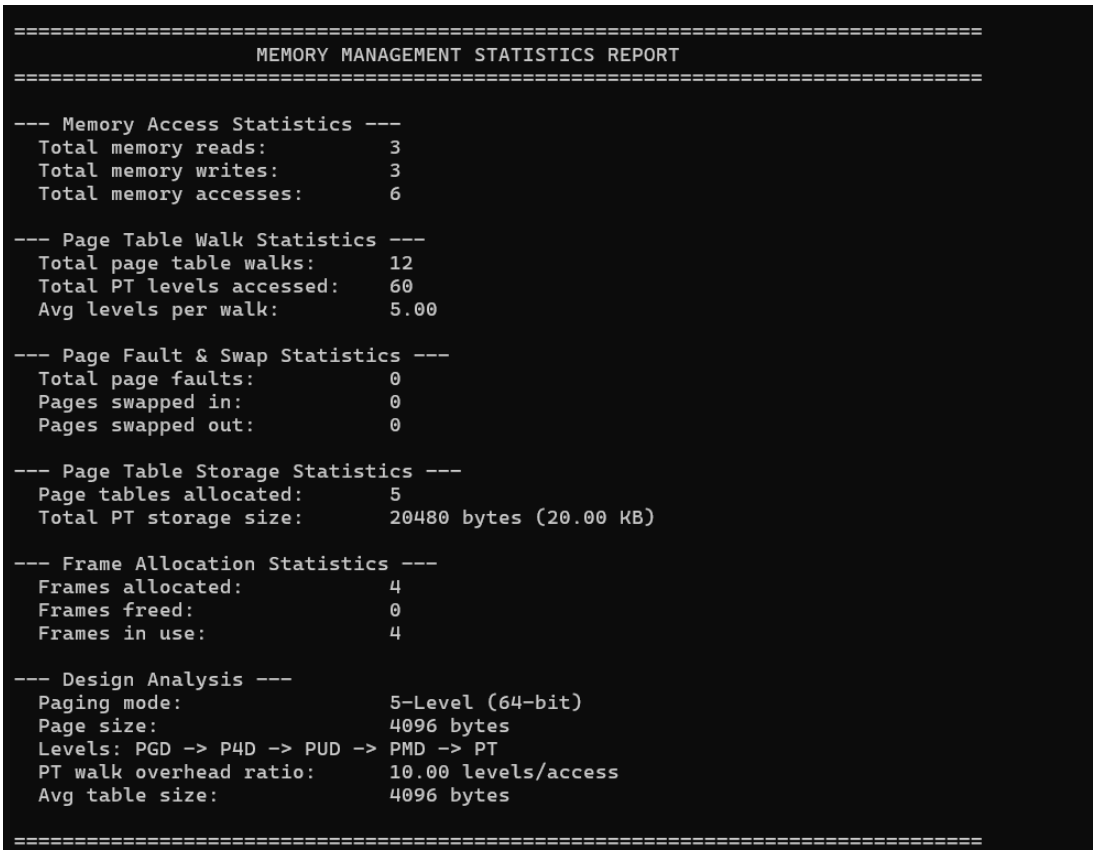

Figure 5: *Báo cáo thống kê (os_0_mlq_paging)*

Phân tích Thống kê (Hình 5):

- **Cơ chế 5 cấp:** Số liệu `Avg levels per walk: 5.00` xác nhận mọi truy cập bộ nhớ đều đi qua đúng 5 bước dịch địa chỉ ($PGD \rightarrow P4D \rightarrow PUD \rightarrow PMD \rightarrow PT$), tuân thủ đúng kiến trúc MM64.
- **Cấp phát thưa (Sparse Allocation):** Dù hỗ trợ không gian địa chỉ 64-bit khổng lồ, hệ thống chỉ tốn **20.00 KB** (tương ứng với 5 bảng trang con được cấp phát) để quản lý vùng nhớ cho tiến trình này. Đây là minh chứng cho hiệu quả của kỹ thuật cấp phát lười (Lazy Allocation).

2.2 Kịch bản 2: Swapping khi thiếu RAM (`os_1_mlq_paging_small_1K`)

Cấu hình: RAM cực nhỏ (2048 bytes = 8 frames). Tổng nhu cầu bộ nhớ vượt xa năng lực vật lý.

```
namtrhcmut@hoangnamtruong:/mnt/c/BKL/Nam3_HK1/HDH/ossim_lamiaatrium/ossim_lamiaatrium$ ./os os_1_mlq_paging_small_1K
Time slot   0
ld_routine
Time slot   1
        Loaded a process at input/proc/p0s, PID: 1 PRIO: 130
Time slot   2
        CPU 3: Dispatched process  1
        Loaded a process at input/proc/s3, PID: 2 PRIO: 39
Time slot   3
        CPU 2: Dispatched process  2
liballoc:185
print_pgtbl:
 PDG=720ba00030f0 P4g=720ba0003000 PUD=720ba0003010 PMD=720ba0003020
Time slot   4
        CPU 3: Put process  1 to run queue
        CPU 3: Dispatched process  1
liballoc:185
print_pgtbl:
 PDG=720ba00030f0 P4g=720ba0003000 PUD=720ba0003010 PMD=720ba0003020
        Loaded a process at input/proc/m1s, PID: 3 PRIO: 15
Time slot   5
libfree:213
print_pgtbl:
 PDG=720ba00030f0 P4g=720ba0003000 PUD=720ba0003010 PMD=720ba0003020
        CPU 2: Put process  2 to run queue
        CPU 2: Dispatched process  3
        CPU 1: Dispatched process  2
liballoc:185
print_pgtbl:
 PDG=720ba00030f0 P4g=720ba0003000 PUD=720ba0003010 PMD=720ba0003020
Time slot   6
        CPU 3: Put process  1 to run queue
        CPU 3: Dispatched process  1
liballoc:185
print_pgtbl:
 PDG=720ba00030f0 P4g=720ba0003000 PUD=720ba0003010 PMD=720ba0003020
liballoc:185
print_pgtbl:
 PDG=720ba00030f0 P4g=720ba0003000 PUD=720ba0003010 PMD=720ba0003020
        Loaded a process at input/proc/s2, PID: 4 PRIO: 120
Time slot   7
libwrite:431
print_pgtbl:
 PDG=720ba00030f0 P4g=720ba0003000 PUD=720ba0003010 PMD=720ba0003020
        CPU 1: Put process  2 to run queue
        CPU 1: Dispatched process  2
        CPU 2: Put process  3 to run queue
        CPU 2: Dispatched process  3
libfree:213
print_pgtbl:
 PDG=720ba00030f0 P4g=720ba0003000 PUD=720ba0003010 PMD=720ba0003020
        CPU 0: Dispatched process  4
        Loaded a process at input/proc/m0s, PID: 5 PRIO: 120
Time slot   8
        CPU 3: Put process  1 to run queue
        CPU 3: Dispatched process  5
liballoc:185
print_pgtbl:
 PDG=720ba00030f0 P4g=720ba0003000 PUD=720ba0003010 PMD=720ba0003020
```

Figure 6: *Log thực thi test case os_1_mlq_paging_small_1K*

Phân tích Log (Hình 6): Quan sát log từ Time slot 6 trở đi (khi nhiều tiến trình được nạp), ta thấy địa chỉ `PDG=720ba00030f0` **không thay đổi**. Điều này phù hợp với thiết kế dùng chung `mm_struct`. Tuy nhiên, việc các lệnh `ALLOC` và `WRITE` vẫn thành công mà không báo lỗi "Out of Memory" cho thấy cơ chế Swapping đã hoạt động ngầm bên dưới lớp vỏ địa chỉ tĩnh này.

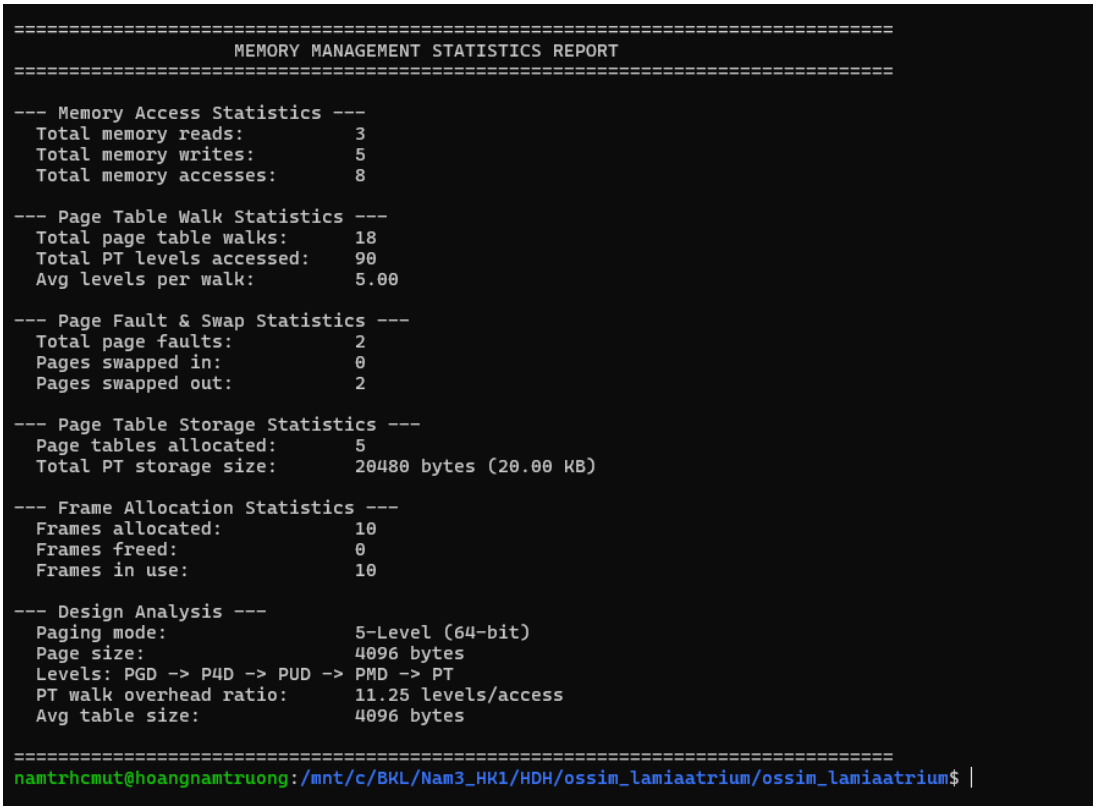

Figure 7: *Báo cáo thống kê Swapping (os_1_mlq_paging_small_1K)*

Phân tích Thống kê (Hình 7): Đây là bằng chứng xác thực nhất cho hoạt động của hệ thống:

- **Swapping hoạt động:**
  - `Frames in use: 10`: Hệ thống đang quản lý 10 khung trang dữ liệu.
  - `Pages swapped out: 2`: Do RAM vật lý chỉ chứa được 8 khung trang, hệ thống đã buộc phải đẩy 2 trang ra thiết bị SWAP để lấy chỗ trống.
- **Cơ chế:** Khi RAM đầy, hệ thống tìm trang nạn nhân (Victim Page), thay đổi bit trạng thái trong PTE (tắt bit *Present*, bật bit *Swapped*) và di chuyển dữ liệu. Quá trình này diễn ra trong suốt với tiến trình người dùng.

3 Phân tích System Call (`os_syscall`)

Mục tiêu: Kiểm tra khả năng chuyển đổi User/Kernel mode. **Phân tích Output (`os_syscall.output`):**

- **Thực thi:** Log cho thấy tiến trình người dùng gọi syscall. Ngay sau đó, các hàm kernel như `liballoc`, `print_pgtbl` được kích hoạt và in ra log.
- **Kết luận:** Cơ chế chuyển đổi ngữ cảnh (Context Switch) hoạt động trơn tru. Tham số được truyền đúng từ User Space vào Kernel Space (thông qua cơ chế truyền tham chiếu struct), và Kernel trả quyền điều khiển lại sau khi hoàn tất.

4 Tổng kết chung

Kết quả thực nghiệm xác nhận hệ thống Simple OS đã hoàn thành xuất sắc các mục tiêu thiết kế:

1. **Scheduler:** Đảm bảo ưu tiên đúng (Priority Scheduling) và chia sẻ công bằng (Round Robin).
2. **Memory:** Quản lý thành công không gian địa chỉ 64-bit phức tạp với chi phí lưu trữ thấp (20KB).

3. **Stability:** Hệ thống vận hành ổn định trong điều kiện khắc nghiệt (thiếu RAM) nhờ cơ chế Swapping hiệu quả, dù sử dụng mô hình quản lý bộ nhớ chung (Unified Memory) cho môi trường mô phỏng.

V Lời Cảm Ơn

Trong folder submission, chúng em có tạo 2 file là kienthuc.md và dif.md
- File Kiến thức sẽ nói về hệ thống, tổng hợp các định nghĩa, chức năng cơ bản để tất cả thành viên dễ dàng cho việc hiện thực
- File dif.md tổng hợp những điểm thay đổi so với source code gốc như sử dụng page Alignment,....

Cuối cùng, chúng em xin cảm ơn thầy Nguyễn Minh Tâm vì đã giúp đỡ chúng em trong suốt quá trình thực hiện bài tập lớp. Chúc thầy có thật nhiều sức khỏe để đạt được nhiều thành công trên con đường kế tiếp!