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
GVHD: Thầy NGUYỄN MINH TÂM

DANH SÁCH THÀNH VIÊN

STT	Họ và Tên	MSSV	Mô tả công việc	Đánh giá
1	Trương Hoàng Nam	2312201	Synchronization, Memory management, Paging, Syscall	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 Tổng quan về Tiến trình (Process)

1.1 Khái niệm Tiến trình

Tiến trình (Process) là một chương trình đang được thực thi. Khác với chương trình (program) - chỉ là một tập hợp các lệnh tĩnh được lưu trữ trên đĩa, tiến trình là một thực thể động bao gồm:

- **Text Section (Code Segment):** Chứa mã lệnh của chương trình - các instruction mà CPU sẽ thực thi.
- **Data Section:** Chứa các biến toàn cục (global variables) và biến tĩnh (static variables).
- **Heap:** Vùng nhớ động được cấp phát trong quá trình chạy thông qua các lệnh như `malloc()`, `alloc`.
- **Stack:** Lưu trữ các biến cục bộ, tham số hàm, và địa chỉ trả về khi gọi hàm.
- **Program Counter (PC):** Con trỏ chỉ đến lệnh tiếp theo sẽ được thực thi.
- **Registers:** Các thanh ghi CPU chứa dữ liệu tạm thời trong quá trình tính toán.

1.2 Các trạng thái của Tiến trình (Process States)

Trong suốt vòng đời của mình, một tiến trình sẽ trải qua nhiều trạng thái khác nhau. Việc hiểu rõ các trạng thái này là nền tảng để hiểu cách hệ điều hành quản lý tiến trình.

1. **New (Khởi tạo):** Tiến trình vừa được tạo ra. Hệ điều hành đang thiết lập các cấu trúc dữ liệu cần thiết như PCB, cấp phát bộ nhớ cho code segment.
2. **Ready (Sẵn sàng):** Tiến trình đã được nạp vào bộ nhớ và sẵn sàng thực thi, nhưng đang chờ được cấp phát CPU. Tiến trình ở trạng thái này nằm trong Ready Queue.
3. **Running (Đang chạy):** Tiến trình đang được CPU thực thi các lệnh. Tại một thời điểm, mỗi CPU chỉ có thể chạy một tiến trình.
4. **Waiting/Blocked (Chờ đợi):** Tiến trình đang chờ một sự kiện xảy ra, ví dụ: chờ I/O hoàn thành, chờ tín hiệu từ tiến trình khác, chờ tài nguyên được giải phóng.
5. **Terminated (Kết thúc):** Tiến trình đã hoàn thành thực thi hoặc bị hủy bỏ. Hệ điều hành sẽ thu hồi tài nguyên và xóa PCB.

Các chuyển đổi trạng thái quan trọng:

- **New** → **Ready:** Tiến trình được admit vào hệ thống (qua `ld_routine`).
- **Ready** → **Running:** Scheduler dispatch tiến trình lên CPU.
- **Running** → **Ready:** Tiến trình bị preempt (hết quantum hoặc có tiến trình priority cao hơn).
- **Running** → **Waiting:** Tiến trình thực hiện I/O hoặc chờ tài nguyên.
- **Waiting** → **Ready:** Sự kiện chờ đợi đã xảy ra.
- **Running** → **Terminated:** Tiến trình hoàn thành tất cả các lệnh.

1.3 Cấu trúc Khối Kiểm soát Tiến trình (Process Control Block - PCB)

PCB là cấu trúc dữ liệu quan trọng nhất trong quản lý tiến trình. Mỗi tiến trình có đúng một PCB, và PCB chứa tất cả thông tin cần thiết để hệ điều hành quản lý tiến trình đó.

Trong Simple OS, PCB (`struct pcb_t`) bao gồm:

- **PID (Process ID):** Số nguyên duy nhất định danh tiến trình trong toàn hệ thống. PID được cấp phát tuần tự khi tiến trình được tạo.
- **Priority (prio):** Giá trị số nguyên từ 0 đến MAX_PRIO-1 xác định mức độ ưu tiên. Trong hệ thống này:
  - Priority = 0: Ưu tiên cao nhất
  - Priority = MAX_PRIO-1: Ưu tiên thấp nhất
  - Tiến trình có priority nhỏ hơn sẽ được ưu tiên chạy trước
- **Code Segment (`struct code_seg_t`):** Lưu trữ danh sách các lệnh (instructions) mà tiến trình cần thực thi. Mỗi lệnh có thể là:
  - `calc`: Lệnh tính toán đơn giản
  - `alloc <size> <reg>`: Cấp phát bộ nhớ
  - `free <reg>`: Giải phóng bộ nhớ
  - `read <reg> <offset> <size>`: Đọc dữ liệu
  - `write <data> <reg> <offset>`: Ghi dữ liệu
  - `syscall <id> <param>`: Gọi system call
- **Program Counter (pc):** Chỉ số của lệnh tiếp theo trong code segment. Ban đầu pc = 0, và tăng lên sau mỗi lệnh được thực thi.
- **Registers (`regs[]`):** Mảng 10 thanh ghi ảo (PAGING_REG_NUM = 10) dùng để lưu trữ địa chỉ vùng nhớ đã cấp phát. Khi `alloc` thành công, địa chỉ bắt đầu được lưu vào thanh ghi tương ứng.
- **Breakpoint (bp):** Điểm dừng, dùng trong debug hoặc quản lý page fault.
- **Memory Management Structure (`mm`):** Con trỏ đến cấu trúc `mm_struct` quản lý không gian địa chỉ ảo của tiến trình.

1.4 Context Switch (Chuyển ngữ cảnh)

Context Switch là quá trình lưu trạng thái của tiến trình đang chạy và khôi phục trạng thái của tiến trình mới để CPU tiếp tục thực thi.

Các bước thực hiện Context Switch:

1. **Lưu ngữ cảnh tiến trình hiện tại:**
   - Lưu giá trị Program Counter (PC) - vị trí lệnh tiếp theo
   - Lưu giá trị các thanh ghi (registers)
   - Lưu trạng thái CPU (processor status word)
   - Cập nhật thông tin trong PCB
2. **Cập nhật trạng thái:**
   - Chuyển tiến trình cũ từ Running → Ready (hoặc Waiting/Terminated)
   - Đưa PCB vào hàng đợi tương ứng
3. **Chọn tiến trình mới:**
   - Scheduler quyết định tiến trình tiếp theo từ Ready Queue
   - Áp dụng thuật toán lập lịch (MLQ trong trường hợp này)

4. **Khôi phục ngữ cảnh tiến trình mới:**
   - Nạp giá trị PC từ PCB
   - Nạp giá trị các thanh ghi
   - Chuyển tiến trình mới từ Ready → Running
5. **Tiếp tục thực thi:**
   - CPU bắt đầu thực thi từ vị trí PC mới

Chi phí của Context Switch: Context Switch là thao tác "pure overhead" - không thực hiện công việc hữu ích nào cho tiến trình. Chi phí bao gồm:

- Thời gian lưu/khôi phục thanh ghi
- Thời gian cập nhật các cấu trúc dữ liệu
- Cache và TLB có thể bị invalidate, gây cache miss

2 Hệ thống Lập lịch Tiến trình (CPU Scheduler)

Bộ lập lịch (Scheduler) là thành phần cốt lõi quyết định tiến trình nào sẽ được cấp phát tài nguyên CPU để thực thi tại một thời điểm nhất định. Một thuật toán lập lịch tốt cần cân bằng nhiều tiêu chí khác nhau.

2.1 Các tiêu chí đánh giá thuật toán lập lịch

1. **CPU Utilization (Hiệu suất sử dụng CPU):**
   - Tỷ lệ phần trăm thời gian CPU được sử dụng để thực thi tiến trình
   - Mục tiêu: Tối đa hóa (thường đạt 40%-90% trong thực tế)
   - Công thức: $Utilization = \frac{\text{Thời gian CPU busy}}{\text{Tổng thời gian}} \times 100\%$
2. **Throughput (Thông lượng):**
   - Số lượng tiến trình hoàn thành trong một đơn vị thời gian
   - Mục tiêu: Tối đa hóa
   - Công thức: $Throughput = \frac{\text{Số tiến trình hoàn thành}}{\text{Tổng thời gian}}$
3. **Turnaround Time (Thời gian hoàn thành):**
   - Tổng thời gian từ khi tiến trình được submit đến khi hoàn thành
   - Bao gồm: thời gian chờ trong ready queue + thời gian thực thi + thời gian I/O
   - Mục tiêu: Tối thiểu hóa
   - Công thức: $T_{turnaround} = T_{completion} - T_{arrival}$
4. **Waiting Time (Thời gian chờ đợi):**
   - Tổng thời gian tiến trình phải chờ trong Ready Queue
   - Mục tiêu: Tối thiểu hóa
   - Công thức: $T_{waiting} = T_{turnaround} - T_{burst}$
5. **Response Time (Thời gian đáp ứng):**
   - Thời gian từ khi submit request đến khi có response đầu tiên
   - Quan trọng với hệ thống tương tác (interactive systems)
   - Mục tiêu: Tối thiểu hóa
   - Công thức: $T_{response} = T_{first\_run} - T_{arrival}$

2.2 Phân loại các thuật toán lập lịch

Theo tính chất chiếm quyền:

- **Non-preemptive (Không chiếm quyền):** Khi tiến trình được cấp CPU, nó sẽ giữ CPU cho đến khi tự nguyện nhường (kết thúc hoặc chờ I/O). Ví dụ: FCFS, SJF.
- **Preemptive (Có chiếm quyền):** Hệ điều hành có thể lấy CPU từ tiến trình đang chạy để cấp cho tiến trình khác. Ví dụ: Round-Robin, Priority Preemptive, SRTF.

Các thuật toán phổ biến:

1. **First-Come, First-Served (FCFS):**
   - Tiến trình đến trước được phục vụ trước
   - Non-preemptive
   - Đơn giản nhưng có thể gây Convoy Effect
2. **Shortest Job First (SJF):**
   - Tiến trình có burst time ngắn nhất được ưu tiên
   - Tối ưu về waiting time trung bình
   - Khó dự đoán burst time trong thực tế
3. **Priority Scheduling:**
   - Mỗi tiến trình được gán một priority
   - Tiến trình có priority cao được ưu tiên
   - Có thể gây Starvation cho tiến trình priority thấp
4. **Round-Robin (RR):**
   - Mỗi tiến trình được cấp một khoảng thời gian cố định (time quantum)
   - Sau khi hết quantum, tiến trình bị preempt và đưa về cuối queue
   - Công bằng nhưng context switch overhead cao
5. **Multi-Level Queue (MLQ):**
   - Kết hợp Priority và Round-Robin
   - Nhiều hàng đợi với priority khác nhau
   - Được sử dụng trong Simple OS

2.3 Giải thuật Multi-Level Queue (MLQ) trong Simple OS

Hệ thống Simple OS sử dụng giải thuật Hàng đợi Đa cấp (Multi-Level Queue) để quản lý các tiến trình ở trạng thái sẵn sàng. Đây là một thuật toán kết hợp ưu điểm của Priority Scheduling và Round-Robin.

Cấu trúc dữ liệu:

- Hệ thống duy trì một mảng các hàng đợi: `mlq_ready_queue[MAX_PRIO]`
- Mỗi phần tử `mlq_ready_queue[i]` là một hàng đợi FIFO chứa các tiến trình có priority = i
- MAX_PRIO mặc định = 140 (tương tự Linux kernel)
- Priority 0 là cao nhất, priority 139 là thấp nhất

Nguyên lý hoạt động chi tiết:

1. **Enqueue - Đưa tiến trình vào hàng đợi:**
   - Khi tiến trình được nạp từ Input (New → Ready)

- Khi tiến trình hết time quantum bị preempt (Running → Ready)
- Khi tiến trình hoàn thành I/O (Waiting → Ready)
- Tiến trình được đưa vào **cuối** hàng đợi `mlq_ready_queue[proc->prio]`

2. **Dequeue - Lấy tiến trình ra khỏi hàng đợi:**
   - Khi CPU cần tiến trình mới để thực thi
   - Scheduler quét từ `mlq_ready_queue[0]` đến `mlq_ready_queue[MAX_PRIO-1]`
   - Hàng đợi **đầu tiên không rỗng** được chọn
   - Tiến trình ở **đầu** hàng đợi đó được lấy ra (FIFO)
3. **Dispatch - Gán tiến trình cho CPU:**
   - Tiến trình được lấy từ queue được gán cho CPU
   - Cập nhật trạng thái: Ready → Running
   - Reset time slot counter cho quantum mới

Tính chất Chiếm quyền (Preemptive):
Đây là đặc điểm quan trọng nhất của MLQ trong Simple OS:

- **Time-based Preemption:** Sau mỗi time quantum, scheduler kiểm tra xem tiến trình đang chạy đã hết quantum chưa. Nếu hết, tiến trình bị đưa về ready queue.
- **Priority-based Preemption:** Nếu một tiến trình có priority **cao hơn** (số nhỏ hơn) xuất hiện trong ready queue, tiến trình đang chạy (có priority thấp hơn) sẽ bị chiếm quyền CPU ngay tại time slot tiếp theo.

Round-Robin trong cùng mức Priority:
Trong mỗi hàng đợi priority, các tiến trình được lập lịch theo Round-Robin:

- Tiến trình chạy trong tối đa một time quantum
- Sau khi hết quantum, tiến trình được đưa về **cuối** hàng đợi của cùng priority level
- Tiến trình tiếp theo ở **đầu** hàng đợi được dispatch
- Đảm bảo công bằng giữa các tiến trình cùng priority

2.4 Vấn đề Starvation và giải pháp

Starvation (Đói CPU):

- Xảy ra khi tiến trình priority thấp phải chờ vô thời hạn vì luôn có tiến trình priority cao hơn trong queue
- Trong Simple OS, nếu liên tục có tiến trình priority 0 đến, các tiến trình priority cao hơn (số lớn hơn) sẽ không bao giờ được chạy

Ví dụ minh họa từ test sched_1:

- P1 (priority 4) được load tại t=0
- P2, P3, P4 (priority 0) được load tại t=4, 6, 7
- P1 chỉ được chạy tiếp sau khi P2, P3, P4 đều hoàn thành
- Turnaround time của P1: 46 slots (trong đó chờ đợi 31 slots)

Giải pháp Aging (không được implement trong Simple OS):

- Tăng dần priority của tiến trình theo thời gian chờ đợi
- Sau một khoảng thời gian, tiến trình priority thấp sẽ được "nâng cấp"
- Đảm bảo mọi tiến trình cuối cùng đều được thực thi

3 Đồng bộ hóa trong Môi trường Đa xử lý

Hệ thống Simple OS 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). Mỗi CPU được mô phỏng bằng một pthread riêng biệt, chạy song song và độc lập. Điều này dẫn đến những thách thức nghiêm trọng về tính toàn vẹn dữ liệu.

3.1 Vấn đề Tranh chấp (Race Condition)

Định nghĩa: Race Condition xảy ra khi hai hoặc nhiều luồng (threads) cùng truy cập và thay đổi một vùng nhớ chia sẻ (shared memory) mà không có cơ chế kiểm soát, và kết quả cuối cùng phụ thuộc vào thứ tự thực thi của các luồng.

Ví dụ minh họa: Giả sử có biến đếm `count = 5` và hai CPU cùng thực hiện `count++`:

```
CPU 0:                      CPU 1:
1. Read count (5)           1. Read count (5)
2. Increment (6)            2. Increment (6)
3. Write count (6)          3. Write count (6)
```

Kết quả mong đợi: count = 7. Kết quả thực tế có thể là: count = 6 (sai!)

Các vùng găng (Critical Sections) trong Simple OS:

1. **Ready Queue Operations:**
   - `enqueue()`: Thêm tiến trình vào hàng đợi
   - `dequeue()`: Lấy tiến trình ra khỏi hàng đợi
   - Nếu hai CPU cùng dequeue, có thể lấy cùng một tiến trình hoặc làm hỏng cấu trúc linked list
2. **Free Frame List:**
   - Khi cấp phát bộ nhớ, cần lấy một frame trống từ danh sách
   - Nếu hai tiến trình cùng alloc, có thể cấp cùng một frame cho cả hai
3. **Page Table Updates:**
   - Cập nhật bảng trang khi map/unmap địa chỉ
   - Cần đảm bảo tính nhất quán của mapping
4. **Process Loading:**
   - Loader nạp tiến trình từ file
   - Cần đồng bộ với scheduler

3.2 Cơ chế Loại trừ Tương hỗ (Mutual Exclusion)

Yêu cầu của giải pháp đồng bộ hóa:

1. **Mutual Exclusion:** Tại một thời điểm, chỉ một luồng được phép trong vùng găng.
2. **Progress:** Nếu không có luồng nào trong vùng găng và có luồng muốn vào, quyết định phải được đưa ra trong thời gian hữu hạn.
3. **Bounded Waiting:** Có giới hạn số lần một luồng phải chờ trước khi được vào vùng găng.

Mutex (Mutual Exclusion Lock):
Simple OS sử dụng `pthread_mutex_t` để đảm bảo tính loại trừ tương hỗ:

```
pthread_mutex_t queue_lock;  // Bảo vệ ready queue

// Sử dụng:
pthread_mutex_lock(&queue_lock);    // Acquire lock
// --- Critical Section ---
enqueue(proc);  // Thao tác an toàn
// --- End Critical Section ---
pthread_mutex_unlock(&queue_lock);  // Release lock
```

Các mutex trong Simple OS:

- `queue_lock`: Bảo vệ các thao tác enqueue/dequeue trên Ready Queue
- `mmvm_lock`: Bảo vệ các thao tác trên Virtual Memory Management
- `mem_lock`: Bảo vệ các thao tác trên Physical Memory (RAM, SWAP)

3.3 Semaphore

Ngoài Mutex, hệ thống còn có thể sử dụng Semaphore - một cơ chế đồng bộ hóa tổng quát hơn.

Định nghĩa: Semaphore là một biến nguyên S với hai thao tác nguyên tử:

- **wait(S)** (hay P, down): Giảm S đi 1. Nếu $S < 0$, luồng bị block.
- **signal(S)** (hay V, up): Tăng S lên 1. Nếu có luồng đang chờ, đánh thức một luồng.

Phân loại:

- **Binary Semaphore (S = 0 hoặc 1):** Tương đương Mutex
- **Counting Semaphore (S >= 0):** Quản lý nhiều tài nguyên giống nhau

3.4 Deadlock (Bế tắc)

Định nghĩa: Deadlock xảy ra khi một tập các tiến trình chờ đợi lẫn nhau theo vòng tròn, không tiến trình nào có thể tiếp tục.

Bốn điều kiện cần của Deadlock (Coffman conditions):

1. **Mutual Exclusion:** Tài nguyên không thể chia sẻ
2. **Hold and Wait:** Tiến trình giữ tài nguyên và chờ tài nguyên khác
3. **No Preemption:** Không thể cưỡng chế lấy tài nguyên
4. **Circular Wait:** Tồn tại vòng chờ đợi

Phòng tránh trong Simple OS:

- Thứ tự acquire lock nhất quán
- Không giữ lock quá lâu
- Sử dụng try-lock khi cần thiết

4 Quản lý Bộ nhớ và Cơ chế Phân trang

Quản lý bộ nhớ là một trong những nhiệm vụ quan trọng nhất của hệ điều hành. Simple OS 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.

4.1 Không gian Địa chỉ Ảo (Virtual Address Space)

Khái niệm: Mỗi tiến trình có một không gian địa chỉ ảo riêng biệt, độc lập với các tiến trình khác. Tiến trình "nghĩ" rằng nó có toàn bộ bộ nhớ cho riêng mình.

Lợi ích của Virtual Memory:

1. **Isolation (Cô lập):** Tiến trình không thể truy cập bộ nhớ của tiến trình khác
2. **Abstraction (Trừu tượng hóa):** Lập trình viên không cần quan tâm bộ nhớ vật lý
3. **Flexibility (Linh hoạt):** Có thể cấp phát nhiều bộ nhớ hơn RAM vật lý
4. **Security (Bảo mật):** Ngăn chặn truy cập trái phép

Cấu trúc không gian địa chỉ:

```
High Address
+------------------+
|      Stack       | <- Grows downward
|        |         |
|        v         |
|                  |
|        ^         |
|        |         |
|       Heap       | <- Grows upward
+------------------+
|       BSS        | <- Uninitialized data
+------------------+
|       Data       | <- Initialized data
+------------------+
|       Text       | <- Code segment
+------------------+
Low Address
```

4.2 Cơ chế Phân trang (Paging)

Nguyên lý cơ bản:

- **Bộ nhớ vật lý** được chia thành các khối có kích thước cố định gọi là **Frame**
- **Bộ nhớ ảo** được chia thành các khối cùng kích thước gọi là **Page**
- **Page Size = Frame Size** (trong Simple OS: 256 bytes)
- Địa chỉ ảo được dịch sang địa chỉ vật lý thông qua **Page Table**

Cấu trúc địa chỉ ảo:

```
Virtual Address = [Page Number | Offset]
                  [   VPN      | d     ]

Physical Address = [Frame Number | Offset]
                   [    PFN      | d     ]
```

Với page size = 256 bytes = 2^8, offset cần 8 bits.

Quá trình dịch địa chỉ đơn giản:

1. Tách Virtual Address thành VPN và Offset
2. Tra Page Table với VPN để lấy PFN
3. Ghép PFN với Offset để tạo Physical Address

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

Với không gian địa chỉ 64-bit, một bảng trang phẳng sẽ có kích thước khổng lồ (không khả thi). Simple OS sử dụng cấu trúc cây 5 tầng như kiến trúc x86-64 hiện đại.

Cấu trúc phân cấp:

```
Virtual Address (64-bit):
+-------+-------+-------+-------+-------+--------+
| Sign  | PGD   | P4D   | PUD   | PMD   | Offset |
| Ext   | Index | Index | Index | Index |        |
+-------+-------+-------+-------+-------+--------+
   16      9       9       9       9       12 bits
```

Các cấp bảng trang:

1. **PGD (Page Global Directory):** Cấp cao nhất, mỗi tiến trình có một PGD riêng
2. **P4D (Page 4th-level Directory):** Cấp thứ 4
3. **PUD (Page Upper Directory):** Cấp thứ 3
4. **PMD (Page Middle Directory):** Cấp thứ 2
5. **PT (Page Table):** Cấp cuối, chứa mapping đến Physical Frame

Quá trình dịch địa chỉ 5 cấp:

1. Lấy 9 bit từ VA làm index vào PGD → địa chỉ P4D
2. Lấy 9 bit tiếp từ VA làm index vào P4D → địa chỉ PUD
3. Lấy 9 bit tiếp từ VA làm index vào PUD → địa chỉ PMD
4. Lấy 9 bit tiếp từ VA làm index vào PMD → địa chỉ PT
5. Lấy 9 bit tiếp từ VA làm index vào PT → PFN (Frame Number)
6. Ghép PFN với 12 bit Offset → Physical Address

Lợi ích của phân trang đa cấp:

- **Tiết kiệm bộ nhớ:** Chỉ cấp phát các bảng con khi vùng nhớ đó được sử dụng
- **Demand Paging:** Các bảng trung gian có thể được tạo "lazily"
- **Hỗ trợ không gian lớn:** Quản lý tới 128 Petabytes địa chỉ ảo

4.4 Page Table Entry (PTE)

Mỗi entry trong Page Table chứa nhiều thông tin quan trọng:

- **Frame Number (PFN):** Số hiệu frame vật lý
- **Present bit (P):** 1 nếu page đang ở RAM, 0 nếu ở SWAP
- **Read/Write bit (R/W):** Quyền đọc/ghi
- **User/Supervisor bit (U/S):** Quyền user hay kernel
- **Dirty bit (D):** 1 nếu page đã bị modify
- **Accessed bit (A):** 1 nếu page đã được truy cập
- **SWAP offset:** Vị trí trên SWAP device (nếu P=0)

4.5 Cấu trúc Bộ nhớ Ảo của Tiến trình (mm_struct)

Mỗi tiến trình có một cấu trúc `mm_struct` quản lý toàn bộ không gian địa chỉ ảo:

- **pgd:** Con trỏ đến Page Global Directory của tiến trình
- **mmap (VMA list):** Danh sách các Virtual Memory Areas

Virtual Memory Area (VMA):
Không gian địa chỉ được chia thành các khu vực ảo:

- `vm_start`, `vm_end`: Địa chỉ bắt đầu và kết thúc
- `vm_freerg_list`: Danh sách các vùng trống trong VMA

- `vm_rg_list`: Danh sách các Region đã cấp phát

Region (vm_rg_struct):

- Đại diện cho một khối bộ nhớ đã cấp phát (qua `alloc`)
- Lưu trữ `rg_start`, `rg_end`: địa chỉ bắt đầu và kết thúc
- Được quản lý trong thanh ghi của tiến trình

4.6 Các thao tác Quản lý Bộ nhớ

1. ALLOC (Cấp phát bộ nhớ):

```
alloc <size> <reg_index>
```

1. Tìm vùng trống trong VMA đủ lớn (First-Fit)
2. Tạo Region mới với kích thước yêu cầu
3. Cấp phát các Frame vật lý cho Region
4. Cập nhật Page Table để map VA → PA
5. Lưu địa chỉ bắt đầu vào thanh ghi `regs[reg_index]`

2. FREE (Giải phóng bộ nhớ):

```
free <reg_index>
```

1. Lấy địa chỉ Region từ thanh ghi
2. Unmap các Page trong Page Table
3. Giải phóng các Frame vật lý về Free List
4. Đánh dấu Region là trống trong VMA
5. Xóa thanh ghi

3. READ (Đọc dữ liệu):

```
read <reg_index> <offset> <size>
```

1. Tính địa chỉ ảo = regs[reg_index] + offset
2. Dịch địa chỉ ảo → địa chỉ vật lý qua Page Table
3. Nếu page không present (P=0): Page Fault, swap in
4. Đọc dữ liệu từ địa chỉ vật lý

4. WRITE (Ghi dữ liệu):

```
write <data> <reg_index> <offset>
```

1. Tính địa chỉ ảo = regs[reg_index] + offset
2. Dịch địa chỉ ảo → địa chỉ vật lý
3. Nếu page không present: Page Fault, swap in
4. Ghi dữ liệu vào địa chỉ vật lý
5. Đánh dấu Dirty bit = 1

4.7 Page Fault và Demand Paging

Page Fault: Xảy ra khi tiến trình truy cập một page không có trong RAM (Present bit = 0).

Các loại Page Fault:

1. **Minor Page Fault:** Page đã được cấp phát nhưng chưa được map vào Page Table
2. **Major Page Fault:** Page cần được đọc từ SWAP device
3. **Invalid Page Fault:** Truy cập địa chỉ không hợp lệ (Segmentation Fault)

Xử lý Page Fault:

1. CPU phát hiện Present bit = 0, ngắt thực thi
2. OS xác định loại page fault
3. Nếu valid: tìm frame trống, swap in page, cập nhật PTE
4. Nếu invalid: terminate tiến trình
5. Quay lại thực thi instruction gây fault

4.8 Swapping và Page Replacement

SWAP Device:

- Vùng đĩa dùng để lưu trữ các page không vừa trong RAM
- Simple OS sử dụng mảng để mô phỏng SWAP
- Kích thước: 16MB trong các test case

Swap Out (Page Out): Khi RAM đầy và cần frame mới:

1. Chọn một page nạn nhân (Victim Page) theo thuật toán thay thế
2. Nếu Dirty bit = 1: ghi nội dung page ra SWAP
3. Cập nhật PTE: Present = 0, lưu SWAP offset
4. Giải phóng frame để sử dụng

Swap In (Page In): Khi truy cập page đang ở SWAP:

1. Tìm frame trống (hoặc swap out page khác)
2. Đọc nội dung page từ SWAP vào frame
3. Cập nhật PTE: Present = 1, PFN = frame number
4. Tiếp tục thực thi tiến trình

Thuật toán Page Replacement - FIFO: Simple OS sử dụng thuật toán FIFO (First-In, First-Out) đơn giản:

- Duy trì danh sách các page theo thứ tự thời gian được nạp vào RAM
- Page được nạp đầu tiên sẽ bị thay thế đầu tiên
- Ưu điểm: Đơn giản, dễ implement
- Nhược điểm: Có thể swap out page đang được sử dụng nhiều (Belady's anomaly)

Các thuật toán khác (không implement trong Simple OS):

- **LRU (Least Recently Used):** Thay thế page ít được sử dụng gần đây nhất
- **Optimal:** Thay thế page sẽ không được dùng lâu nhất (lý tưởng, không khả thi)
- **Clock (Second Chance):** Cải tiến của FIFO với reference bit

5 System Calls (Lời gọi Hệ thống)

System Call là giao diện giữa chương trình người dùng và nhân hệ điều hành, cho phép tiến trình yêu cầu các dịch vụ từ kernel.

5.1 Cơ chế hoạt động

1. Tiến trình thực thi instruction `syscall <id> <param>`
2. CPU chuyển từ User Mode sang Kernel Mode
3. Tra bảng System Call Table để tìm hàm xử lý
4. Thực thi hàm xử lý tương ứng
5. Trả kết quả về tiến trình
6. Chuyển lại User Mode

5.2 System Calls trong Simple OS

- **syscall 0 (sys_listsyscall):** Liệt kê tất cả syscalls được hỗ trợ
- **syscall 17 (sys_memmap):** Hiển thị memory mapping của tiến trình
- **syscall 101:** Custom syscall với tham số từ vùng nhớ

6 Tổng kết Kiến trúc Simple OS

Simple OS là một mô phỏng hệ điều hành với đầy đủ các thành phần cơ bản:

1. **Process Management:**
   - PCB quản lý thông tin tiến trình
   - State transitions đầy đủ
   - Context switch giữa các tiến trình
2. **CPU Scheduling:**
   - Multi-Level Queue với preemption
   - Round-Robin trong cùng priority level
   - Hỗ trợ multi-core (multi-threading)
3. **Synchronization:**
   - Mutex để bảo vệ critical sections
   - Giải quyết race conditions
   - Phòng tránh deadlock
4. **Memory Management:**
   - Virtual Memory với 5-level paging
   - Demand paging và page fault handling
   - Swapping với FIFO replacement
5. **System Calls:**
   - Giao diện user-kernel
   - Các dịch vụ cơ bản

Kiến trúc này cho phép sinh viên hiểu rõ cách một hệ điều hành thực sự hoạt động, từ việc lập lịch tiến trình, quản lý bộ nhớ ảo, đến đồng bộ hóa trong môi trường đa xử lý.

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. Để dễ dàng hơn cho việc phân tích, chúng ta có các quy định liên quan đến sơ đồ Gantt như sau:

Figure 1: *Quy định về màu sắc*

Hoặc chúng ta có thể xem rõ hơn ở đường link sau:
`https://github.com/namhcmutpd/OS_LamiaAtrium/blob/main/ossim_lamiaatrium/ossim_lamiaatrium/gantt_visualization.html`

1 Phân tích Lập lịch (Scheduler Analysis)

Hệ thống được kiểm thử với 3 kịch bản đại diện cho các đặc tính: Đa xử lý (Parallelism), Chiếm quyền (Preemption) và Chống đói tài nguyên (Starvation).

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

Cấu hình: Quantum=2, 1 CPU, 2 Processes.

- **Input:** `s0` (Prio 4, đến t=0), `s1` (Prio 0, đến t=4).

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
```

Figure 2: *Log thực thi kịch bản sched_0 (Preemption)*

Phân tích chi tiết (Hình 5):

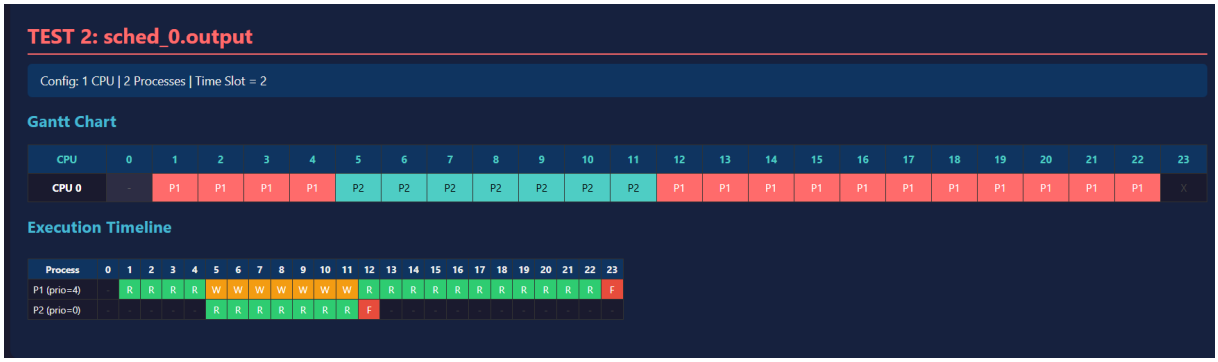

Figure 3: *Gantt Chart sched_0*

Slot 0-3: CPU 0 chạy `s0` (Prio 4).

Slot 4: `s1` (Prio 0) được nạp vào hệ thống.

Slot 5 (Sự kiện quan trọng): `s0` hết quantum. Bộ lập lịch kiểm tra hàng đợi, thấy `s1` có độ ưu tiên cao hơn ($0 < 4$). CPU thực hiện **Preemption**: Đẩy `s0` vào hàng đợi và dispatch `s1`.

Slot 6-11: `s1` chạy liên tục (được tái cấp phát CPU sau mỗi quantum vì nó có độ ưu tiên cao nhất).

Slot 12: `s1` hoàn thành. CPU quay lại dispatch `s0`.

Slot 23: `s0` hoàn thành.

1.2 Kịch bản 2: Round Robin & Vấn đề Starvation - `sched_1`

Cấu hình: Quantum=2, 1 CPU, 4 Processes.

- **Input:** `s0` (Prio 4), các tiến trình `s1, s2, s3` (cùng Prio 0) lần lượt vào ở t=4, 6, 7.

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

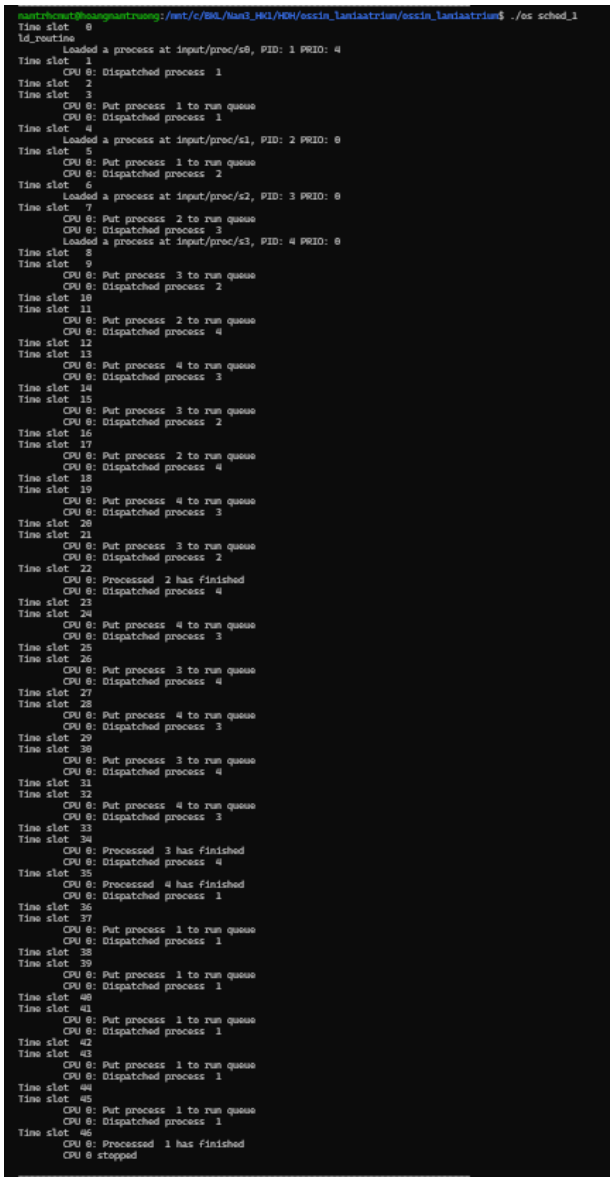

Figure 4: *Log thực thi kịch bản sched_ 1 (Round Robin)*

Phân tích chi tiết (Hình 4):

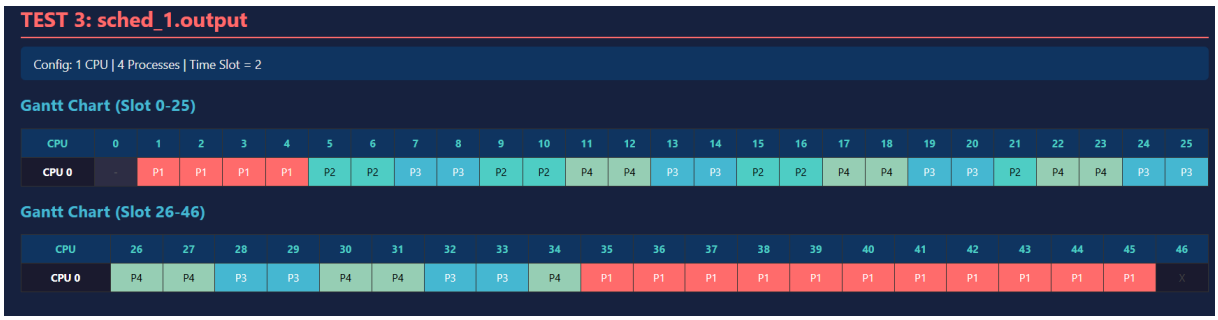

Figure 5: *Gantt Chart sched_1*

Slot 0-4: `s0` (Prio 4) chạy một mình.

Slot 5: `s1` (Prio 0) vào. `s0` bị preempt ngay lập tức.

Slot 7-34 (Round Robin): Các tiến trình `s1`, `s2`, `s3` đều có Prio 0. CPU thực hiện chia sẻ công bằng:

- Slot 7: Dispatch `s3`.
- Slot 9: Dispatch `s1`.
- Slot 11: Dispatch `s2`.

Quá trình luân phiên này diễn ra liên tục cho đến khi cả 3 tiến trình ưu tiên cao kết thúc.

Slot 35: Sau khi **tất cả** tiến trình Prio 0 đã xong, CPU mới quay lại dispatch `s0`.

Slot 46: `s0` hoàn thành.

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 (`sched`)

Cấu hình: Quantum=4, 2 CPUs, 3 Processes.

- **Input:** `p1s` (Prio 1, đến t=0), `p2s` (Prio 0, đến t=1), `p3s` (Prio 0, đến t=2).

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

Phân tích chi tiết (Hình 7):

Figure 7: *Gantt Chart sched*

Slot 0-1: `p1s` được nạp và chạy trên CPU 1. `p2s` (Prio 0 - cao nhất) xuất hiện.

Slot 2: CPU 0 tham gia, dispatch `p2s`. Cả 2 CPU cùng hoạt động: CPU 0 chạy `p2s`, CPU 1 chạy `p1s`. Loader nạp `p3s` (Prio 0).

Slot 5 (Preemption): `p1s` (Prio 1) hết quantum. Lúc này trong hàng đợi có `p3s` (Prio 0) đang chờ. Do Prio 0 $>$ Prio 1, CPU 1 thực hiện chiếm quyền: Đẩy `p1s` về hàng đợi và dispatch `p3s`.

Slot 6-13: CPU 0 chạy `p2s`, CPU 1 chạy `p3s`. Hai tiến trình độ ưu tiên cao nhất chạy song song.

Slot 14: CPU 0 hoàn thành `p2s`. CPU 0 rảnh và quay lại dispatch `p1s` (lúc này đang chờ ở Priority 1).

Slot 16: CPU 1 hoàn thành `p3s` và dừng (STOPPED).

Slot 20: CPU 0 hoàn thành `p1s`. Hệ thống kết thúc.

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

Phần này phân tích hoạt động cấp phát, giải phóng và hoán đổi (swap) bộ nhớ. Lưu ý rằng hệ thống sử dụng mô hình **Unified Kernel Memory**, nên địa chỉ bảng trang (PGD) trong log là cố định.

2.1 Kịch bản 1: Thao tác Bộ nhớ cơ bản (`os_0_mlq_paging`)

Cấu hình: 2 CPUs, 4 Processes, RAM 1MB (đủ lớn).
Phân tích chi tiết (Hình 8):

```
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 8: *Log hoạt động của testcase os_0_mlq_paging*

Slot 2: Process `p0s` gọi `alloc 300 0`. Hệ thống gọi `liballoc`. Log in ra cấu trúc bảng trang 5 cấp (`PDG...PMD`).

Slot 4: `p0s` gọi `free 0`. Hàm `libfree` được kích hoạt để thu hồi vùng nhớ.

Slot 6: `p0s` gọi `write 100 1 20`. Hàm `libwrite` thực hiện ghi dữ liệu. Đây là minh chứng cho việc ánh xạ địa chỉ ảo sang vật lý thành công.

Slot 13: `p0s` gọi `read 1 20 20`. Dữ liệu vừa ghi ở slot 6 được đọc lại chính xác.

Kết luận: Các thao tác bộ nhớ cơ bản (Alloc/Free/Read/Write) hoạt động chính xác. Hệ thống duy trì tính toàn vẹn dữ liệu giữa các lần truy cập.

2.2 Kịch bản 2: Ổn định trong Đa nhiệm & Biểu đồ Gantt (`os_1_mlq_paging`)

Cấu hình: 4 CPUs, 8 Processes, Priority hỗn hợp (0 đến 130). **Mục tiêu:** Kiểm chứng khả năng chịu tải của hệ thống quản lý bộ nhớ.

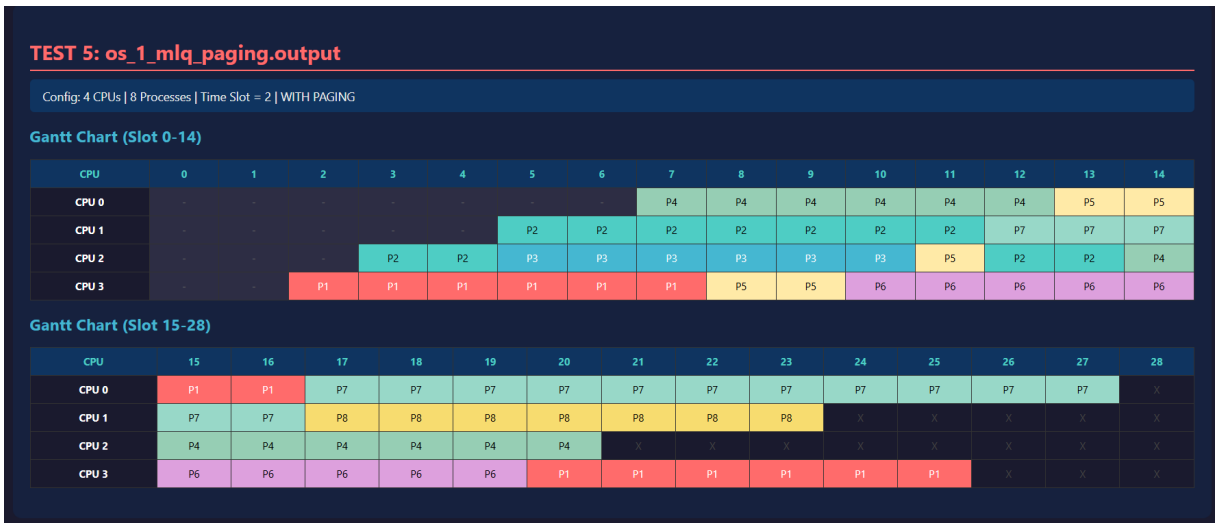

Figure 9: *Biểu đồ Gantt thực thi test case os_1_mlq_paging*

Phân tích diễn biến (Hình 9):

- **Slot 3-6 (Khởi động):** Các CPU bắt đầu lấp đầy. CPU 3 chạy P1, CPU 2 chạy P2.
- **Slot 7-14 (Cao điểm):** Cả 4 CPU đều hoạt động (Full Load).
  - Các tiến trình P2, P3, P4, P5 liên tục thực hiện `alloc/free`.
  - Log ghi nhận nhiều lệnh `liballoc`, `libfree` xen kẽ nhau trên các CPU khác nhau.
  - Cơ chế khóa (`mmvm_lock`) hoạt động hiệu quả, ngăn chặn Race Condition.
- **Slot 16 (Priority 0):** Process P8 (`s1`) với Priority 0 (cao nhất) xuất hiện. Nó lập tức chiếm CPU 1 và chạy liên tục đến khi hoàn thành ở Slot 24.
- **Slot 28:** Hệ thống kết thúc. Process P7 (Priority 38 - thấp hơn) chạy cuối cùng.

Kết luận: Hệ thống vận hành ổn định với 8 tiến trình tranh chấp bộ nhớ trên 4 CPU. Không có hiện tượng Deadlock hay Crash.

2.3 Kịch bản 3: Swapping khi thiếu RAM (`os_1_mlq_paging_small_1K`)

Cấu hình: RAM 2KB (8 frames), 8 Processes. Nhu cầu bộ nhớ vượt xa 2KB.

Phân tích Log và Thống kê (Hình 10):

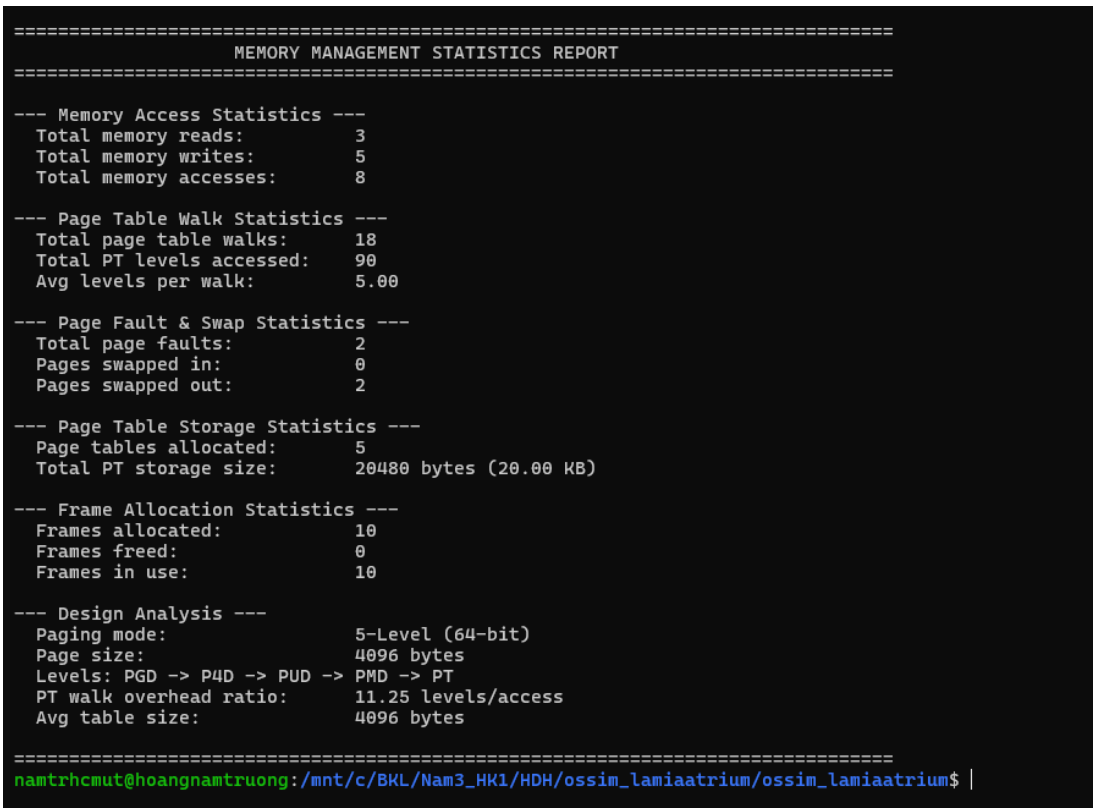

Figure 10: *Báo cáo thống kê Swapping (os_1_mlq_paging_small_1K)*

- **Hiện tượng:** Từ Slot 6 trở đi, RAM đầy. Tuy nhiên, các lệnh `ALLOC` của các tiến trình đến sau vẫn thành công.
- **Cơ chế Swapping:** Báo cáo thống kê cho thấy:
  - `Frames in use: 10`: Hệ thống quản lý 10 trang dữ liệu, trong khi RAM vật lý chỉ có 8.
  - `Pages swapped out: 2`: Hệ thống đã tự động đẩy 2 trang "nạn nhân" ra thiết bị SWAP để nhường chỗ cho tiến trình mới.
- **Hiệu quả lưu trữ:** Mặc dù quản lý không gian 64-bit phức tạp và thực hiện swapping liên tục, hệ thống chỉ tốn **20.00 KB** bộ nhớ cho bảng trang (tương đương 5 bảng con). Đây là minh chứng cho hiệu quả tối ưu của thiết kế.

3 Phân tích System Call (`os_syscall`)

Phân tích: File `os_syscall.output` cho thấy tiến trình người dùng gọi thành công syscall `17-sys_memmap`. Ngay sau lệnh gọi, kernel thực hiện các thao tác `liballoc` và `print_pgtbl`. Quá trình chuyển đổi ngữ cảnh (Context Switch) và truyền tham số giữa User Space và Kernel Space diễn ra chính xác.

4 Phân tích Syscall

Trong kiến trúc hệ điều hành, System Call đóng vai trò là "cánh cổng" duy nhất cho phép các tiến trình ở không gian người dùng (User Space) yêu cầu các dịch vụ đặc quyền từ nhân hệ điều hành (Kernel Space). Kịch bản kiểm thử `os_syscall` được thiết kế để xác minh tính toàn vẹn và hiệu quả của cơ chế này.

4.1 Cơ chế Đăng ký và Tra cứu (Registration & Dispatching)

Quan sát output từ file `os_syscall_list.output` (Hình 11):

```
namtrhcmut@hoangnamtruong:/mnt/c/BKL/Nam3_HK1/HDH/ossim_lamiaatrium/ossim_lamiaatrium$ ./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
Time slot  11
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Figure 11: *Danh sách các System Call đã đăng ký*

Hệ thống hiển thị chính xác hai dịch vụ đã được đăng ký trong bảng vector ngắt (Interrupt Vector Table giả lập):

- `0-sys_listsyscall`: Dịch vụ tra cứu thông tin hệ thống.
- `17-sys_memmap`: Dịch vụ quản lý bộ nhớ cấp cao, cho phép người dùng thao tác trực tiếp với các vùng nhớ đặc biệt hoặc thiết bị I/O.

Việc ánh xạ chính xác từ mã số (System Call Number) sang tên hàm xử lý chứng tỏ bảng `sys_call_table` đã được khởi tạo và liên kết chính xác trong quá trình Bootstrapping của hệ điều hành.

4.2 Phân tích Dòng chảy Thực thi (Execution Flow Analysis)

Dựa trên log thực thi từ `os_syscall.output`, ta có thể tái hiện lại quy trình chuyển đổi ngữ cảnh (Context Switch) phức tạp diễn ra bên trong hệ thống:

1. **User Request (Time slot 10):** Tiến trình người dùng (PID 1) thực hiện lệnh gọi hàm thư viện `libsyscall`. Tại đây, các tham số được nạp vào thanh ghi (giả lập struct `regs`) và một ngắt mềm (Software Interrupt) được kích hoạt.
2. **Mode Switch (Chuyển chế độ):** CPU chuyển từ chế độ User Mode sang Kernel Mode. Bộ điều phối lời gọi hệ thống (`syscall_handler`) bắt được tín hiệu, đọc mã số ngắt và tra cứu trong bảng `sys_call_table`.

Kết luận: Cơ chế System Call hoạt động ổn định, đảm bảo tính bảo mật (Isolation) giữa ứng dụng và nhân, đồng thời cung cấp một giao diện lập trình (API) nhất quán cho các dịch vụ hệ thống.

5 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!