HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

KHOA CÔNG NGHỆ THÔNG TIN I

Bài tập lớn môn Hệ điều hành

Giảng viên hướng dẫn: Nguyễn Đình Quân
Nhóm: [D23-118] Nhóm 3

Chủ đề: Giao tiếp giữa các tiến trình (IPC) & System Calls

Thành viên nhóm:

- Nguyễn Hữu Anh Tài B23DCCN730
- Nhâm Trọng Dư B23DCKH027
- Nghiêm Viết Đức Toàn B23DCKH117
- Bùi Vĩnh Phúc B23DCCN653
- Lê Xuân Nhân B23DCKH083
- Nguyễn Hoàng Tùng B23DCKH129
- Trần Hưng Trung B23DCKH123
- Trịnh Tất Đạt B23DCCN153
- Phạm Duy Giang B23DCCN261
- Lê Hồng Phong B23DCKH087

Hà Nội – 2025

Mục lục

CHƯƠNG 1: GIỚI THIỆU & NỀN TẢNG

1.1. Đặt vấn đề và Mục tiêu Dự án

1.1.1. Đặt vấn đề

- **Tính tất yếu của Đa Tiến trình:** Các hệ điều hành hiện đại (như Linux) đều hoạt động dựa trên nguyên tắc **đa tiến trình** để tận dụng hiệu suất của CPU đa lõi (multi-core).
- **Vấn đề Phát sinh:** Khi nhiều tiến trình chạy đồng thời, chúng cần **giao tiếp** (trao đổi dữ liệu) và **điều phối** (tránh xung đột tài nguyên chung) để hệ thống hoạt động chính xác.

→ **IPC (Giao tiếp giữa các Tiến trình)** và **System Call** là hai cơ chế cốt lõi mà Kernel Linux cung cấp để giải quyết vấn đề này.

1.1.2. Mục tiêu Dự án

- Phân tích và trình diễn các phương pháp **IPC** (Pipe, Semaphore, Shared Memory, Message Queue, Signal), đánh giá đồng bộ hóa và hiệu suất.
- Làm rõ vai trò của **System Call** như cầu nối và cơ chế bảo vệ giữa **User Mode** và **Kernel Mode**.

1.2. Lý thuyết nền tảng

1.2.1. Tiến trình (Process)

- Tiến trình là chương trình đang thực thi, được hệ điều hành cấp phát tài nguyên và quản lý.
- Mỗi tiến trình có không gian địa chỉ riêng gồm: **Code**, **Data**, **Stack** và **Heap**. Việc tách biệt tiến trình giúp hệ điều hành quản lý tài nguyên hiệu quả và cho phép chạy nhiều bản sao chương trình mà không xung đột.

1.2.2. IPC (Giao tiếp giữa các Tiến trình)

- **IPC** là tập hợp các cơ chế giúp các tiến trình **trao đổi dữ liệu** và **đồng bộ hóa hoạt động** với nhau, vượt qua rào cản bộ nhớ riêng biệt. IPC được phân loại theo phương thức truyền tải:
    - **Message Passing (Truyền thông điệp):** Dữ liệu được truyền qua kênh do Kernel quản lý (ví dụ: Pipe, Message Queue).
    - **Shared Memory (Bộ nhớ chia sẻ):** Các tiến trình cùng truy cập một vùng nhớ chung. Cơ chế này đạt **tốc độ cao** nhưng đòi hỏi **cơ chế đồng bộ** (ví dụ: Semaphore) để tránh xung đột.

1.2.3 System Call (Lời gọi hệ thống)

- **System Call** là giao diện lập trình mà tiến trình ở **User Mode** sử dụng để yêu cầu **Kernel** thực hiện một dịch vụ đặc quyền (ví dụ: tạo file, truy cập thiết bị).
    - **Cơ chế Bảo mật:** Khi System Call được gọi, hệ thống chuyển quyền kiểm soát từ User Mode sang **Kernel Mode** để Kernel kiểm tra, xác thực và thực thi yêu cầu một cách an toàn.

CHƯƠNG 2: KIẾN THỨC CỐT LÕI

2.1. Kiến Trúc Bảo Vệ: User Mode & Kernel Mode

Hệ thống máy tính hiện đại phân chia quyền lực thành 2 chế độ riêng biệt để đảm bảo an toàn và ổn định:

2.1.1. User Mode (Chế độ người dùng)

- **Đối tượng**: Các ứng dụng thông thường (Trình duyệt, Word, Game...).
- **Quyền hạn**: Bị hạn chế (Restricted). Không được phép can thiệp trực tiếp vào phần cứng hoặc vùng nhớ hệ thống.
- **Mode Bit**: 1

2.1.2. Kernel Mode (Chế độ nhân)

- **Đối tượng**: Mã nguồn cốt lõi của Hệ điều hành.
- **Quyền hạn**: Toàn quyền (Privileged). Kiểm soát tuyệt đối CPU, RAM, I/O và các thanh ghi đặc biệt.
- **Mode Bit**: 0

2.2. System Call Interface (SCI)

Định nghĩa: System Call là cơ chế (giao diện) duy nhất cho phép chương trình ở **User Mode** gửi yêu cầu sử dụng dịch vụ hoặc tài nguyên từ **Kernel**.

2.2.1. Tại sao cần SCI?

- Ngăn chặn ứng dụng truy cập tự do gây lỗi hệ thống.
- Cung cấp một chuẩn giao tiếp thống nhất.

2.2.2. Quy trình hoạt động:

1. Ứng dụng gọi hàm thư viện (API).
2. Hệ thống kích hoạt Ngắt mềm (Trap/Software Interrupt).
3. CPU đổi bit mode từ 1 sang 0 (vào Kernel Mode).
4. Kernel thực hiện tác vụ và trả kết quả.
5. CPU đổi bit mode về 1 (quay lại User Mode).

2.3. Quản Lý Tiến Trình: fork() và exec()

Ghi chú: Đây là hai System Call quan trọng nhất trong họ Unix/Linux để chạy chương trình.

2.3.1. Lệnh fork() - Nhân bản

- **Chức năng:** Tạo ra một tiến trình mới (**Child**) là bản sao y hệt của tiến trình hiện tại (**Parent**).
- **Đặc điểm:** Tiến trình con sao chép toàn bộ bộ nhớ (**Stack, Heap, Data**) của cha nhưng có **ID** riêng (**PID**).

- **Giá trị trả về:**
  - Trong tiến trình con: Nhận về 0.
  - Trong tiến trình cha: Nhận về PID của con.

2.3.2. Lệnh exec() - Thay thế

- **Chức năng:** Nạp một chương trình hoàn toàn mới vào không gian nhớ của tiến trình hiện tại.
- **Cơ chế:** Xóa sạch mã lệnh và dữ liệu cũ, thay thế bằng mã lệnh từ file thực thi mới trên ổ cứng.
- **Lưu ý:** exec() không tạo tiến trình mới, nó chỉ "thay não" cho tiến trình đang chạy.

2.4. Quy Trình Tổng Hợp (Workflow)

Để chạy một chương trình mới (ví dụ bạn gõ lệnh ls trong terminal), hệ điều hành thực hiện theo mô hình 3 bước:

1. Bước 1 - fork(): Terminal (Cha) nhân bản chính nó tạo ra tiến trình Con.
2. Bước 2 - exec(): Tiến trình Con gọi exec("ls") để tự thay thế code của mình bằng code của lệnh ls.
3. Bước 3 - wait(): Terminal (Cha) tạm dừng chờ tiến trình Con (lệnh ls) chạy xong để thu hồi tài nguyên và báo cáo kết quả.

CHƯƠNG 3: GIAO TIẾP IPC - TRUYỀN THÔNG ĐIỆP

3.1. Pipe & FIFO

3.1.1. Pipe vô danh (Unnamed pipe)

- **Khái niệm :** Pipe Vô danh là một kênh dữ liệu một chiều .Một pipe được tạo ra bằng hệ thống gọi **pipe()**, hàm này cung cấp một cặp File Descriptor (FDs): fd đọc dữ liệu, fd ghi dữ liệu.
- **Cách hoạt động ở mức kernel:**
  - pipe() tạo hai file descriptor trỏ vào cùng một **pipe buffer** nằm trong kernel sau đó dùng **buffer vòng** và khóa để lưu dữ liệu giữa process ghi và đọc.
  - write() copy dữ liệu từ user đến kernel; nếu đầy thì block, nếu O_NONBLOCK thì lỗi.

    - read() lấy dữ liệu từ kernel đến user; nếu rỗng thì block, nếu không còn writer thì trả EOF.
    - Ghi ≤ PIPE_BUF được xử lý **nguyên tử**, kernel dùng wait-queue để sleep/wakeup reader–writer.
- **Ứng dụng thực tế :** Lệnh shell pipeline **ls | grep pattern** là một ví dụ điển hình về việc sử dụng Pipe Vô danh.Một số ứng dụng của lệnh : Tìm thư mục con theo từ khóa, Kiểm tra file cấu hình, Lọc file lỗi, log theo ngày, Tìm file thực thi, ...
  VD : Tìm tất cả file chứa chữ "code" : *ls | grep code*
  Tìm các file Python: *ls | grep '\.py$'*
- **Hạn chế** : Hạn chế kiến trúc cốt lõi của Pipe Vô danh là tính phụ thuộc vào mối quan hệ tiến trình. Điều này do: Cơ chế chỉ cho phép giao tiếp giữa các tiến trình có quan hệ (cha con hoặc anh em); Hai tiến trình đều dùng chung pipe, các fd của pipe được tự động sao chép từ tiến trình cha sang con trong lúc fork().

3.1.2. Pipe FIFO (named pipe)

- **Khái niệm :** FIFO (named pipe) là một file đặc (special file) trong hệ thống tệp cho phép truyền dữ liệu theo kiểu hàng đợi một chiều giữa các tiến trình không cần quan hệ cha-con.
- **Cách hoạt động ở mức kernel:**
    - mkfifo tạo một inode loại FIFO trong filesystem; nhưng nội dung dữ liệu không nằm trên đĩa mà nằm trong buffer của kernel khi có các tiến trình đọc/ghi.
    - Khi một tiến trình open() FIFO để đọc, nếu chưa có tiến trình mở để ghi thì open() mặc định sẽ block cho đến khi có writer (và ngược lại khi mở để ghi).
    - read() lấy dữ liệu từ buffer kernel vào không gian người dùng; write() copy từ người dùng vào buffer kernel.
    - Nếu buffer đầy, write() (không có O_NONBLOCK) sẽ block; nếu rỗng, read() (không có O_NONBLOCK) sẽ block cho đến có dữ liệu hoặc đến khi writer đóng (trả EOF).
    - Việc ghi với kích thước ≤ PIPE_BUF thường được đảm bảo nguyên tử (atomic): các lần ghi nhỏ hơn hoặc bằng PIPE_BUF sẽ không bị xen kẽ với các ghi khác. (Kiểm tra giá trị: getconf PIPE_BUF /tmp)
- **Ứng dụng thực tế :** FIFO là công cụ lý tưởng để kết nối hai ứng dụng độc lập không có quan hệ cha–con. Nó thường được dùng để truyền dữ liệu thời gian thực giữa các tiến trình. Ví dụ: Tách một chương trình sinh log và một chương trình đọc log, truyền dữ liệu cảm biến hoặc trạng thái từ một script sang một bộ xử lý khác,...
- **Hạn chế** : FIFO là một luồng byte—nó không hỗ trợ random access hay lưu trữ

lâu dài. Dữ liệu được tiêu thụ khi có reader; một bản ghi bị đọc sẽ không có sẵn cho reader khác. Vì FIFO tồn tại như một file trong filesystem, bạn phải quản lý quyền truy cập và xóa nó khi không dùng.

3.2. IPC - Message Queue

3.2.1. Khái niệm

- **Message Queue** là một cơ chế IPC (Giao tiếp giữa các Tiến trình) cho phép các tiến trình trao đổi dữ liệu dưới dạng các khối thông điệp (message) có cấu trúc.
- Các thông điệp này được lưu trữ trong một hàng đợi do Kernel quản lý.
- Khác với Pipe (chỉ truyền luồng byte không cấu trúc), Message Queue duy trì cấu trúc của thông điệp, bao gồm:
    - **Kích thước** (Size) của dữ liệu.
    - **Loại thông điệp** (Message Type/Priority), được sử dụng cho cơ chế ưu tiên.

3.2.2. Cơ chế hàng đợi và Ưu tiên tiên

- **Nguyên tắc hàng đợi (Queue):**
    - Về mặc định, các thông điệp được gửi vào queue sẽ được Kernel duy trì theo cơ chế FIFO (First-In, First-Out - Vào trước Ra trước).
    - Cơ chế này đảm bảo rằng các tiến trình có thể trao đổi dữ liệu một cách không đồng bộ (asynchronous) và tin cậy.
- **Cơ chế Ưu tiên (Priority):**
    - Message Queue (theo chuẩn System V hoặc POSIX) cho phép gán một mức ưu tiên (dựa trên *Loại thông điệp*) cho mỗi thông điệp khi nó được gửi đi.
    - Khi một tiến trình nhận (dequeue) thông điệp, nó có thể chỉ định nhận thông điệp theo loại/ưu tiên. Kernel sẽ ưu tiên trả về thông điệp có **ưu tiên cao nhất** trước, bất kể thời điểm nó được gửi vào queue.
    - Cơ chế này rất quan trọng để đảm bảo các yêu cầu hoặc sự kiện khẩn cấp được xử lý ngay lập tức, vượt lên trên các thông điệp thông thường.

3.3. IPC - Tín hiệu (Signal)

3.3.1. Khái niệm và Cơ chế

- **Tín hiệu (Signal)** là một cơ chế giao tiếp liên tiến trình (IPC) hoạt động theo kiểu **bất đồng bộ** (asynchronous) trong các hệ điều hành Unix/Linux.

- **Mục đích:** Cho phép **Kernel** hoặc một tiến trình gửi thông báo nhanh cho tiến trình khác về một sự kiện đã xảy ra (ví dụ: lỗi hệ thống, yêu cầu tắt chương trình).
- **Hành động Mặc định:** Khi nhận tín hiệu, tiến trình có thể bị **kết thúc** (**SIGKILL**), **tạm dừng** (**SIGSTOP**), bị bỏ qua, hoặc thực thi một **Signal Handler**.

3.3.2. Phân loại Tín hiệu Trọng yếu

- **SIGINT (2) / SIGTERM (15):** Yêu cầu kết thúc tiến trình một cách **an toàn**. Các tín hiệu này **có thể bị xử lý** (bắt bằng Handler).
- **SIGKILL (9): Buộc** tiến trình kết thúc ngay lập tức. Đây là tín hiệu **không thể bị chặn** hoặc xử lý bởi Handler, đảm bảo Kernel có quyền kiểm soát tối đa.
- **SIGHUP (1):** Tín hiệu "Hang Up". Thường dùng để yêu cầu các dịch vụ chạy nền **tải lại cấu hình** mà không cần khởi động lại.

3.3.3. Signal Handler và Ứng dụng Thực tiễn

- **Signal Handler:** Là một hàm được gọi khi tín hiệu đến, hoạt động như một cơ chế **ngắt (interrupt)**. Nó cho phép chương trình **can thiệp** và dọn dẹp an toàn trước khi thoát (ví dụ: Handler cho SIGINT).
- **Ứng dụng Web Server:** Web Server sử dụng Handler cho **SIGHUP** để kích hoạt logic **tải lại cấu hình**. Khi nhận SIGHUP, Server không bị tắt mà chỉ cập nhật tham số hoạt động, đảm bảo dịch vụ **Zero Downtime**.

CHƯƠNG 4: IPC - BỘ NHỚ CHIA SẺ VÀ ĐỒNG BỘ HÓA

4.1. IPC-Shared Memory (Bộ nhớ chia sẻ)

4.1.1. Khái niệm

- **Shared Memory** (bộ nhớ chia sẻ) là một cơ chế IPC cho phép nhiều tiến trình cùng truy cập trực tiếp vào chung một vùng nhớ trong RAM
- Khác với Pipe, FIFO, Message Queue – dữ liệu không cần phải copy qua kernel nhiều lần, mà chỉ cần đặt trực tiếp vào vùng nhớ chung

4.1.2. Cơ chế hoạt động

B1: Tạo và thiết lập vùng nhớ chung

B2: Ánh xạ vào không gian tiến trình

B3: Đồng bộ & giải phóng nó

4.1.3. So sánh tốc độ với các cơ chế khác

Khác với Pipe, FIFO, Message Queue, Shared Memory không cần phải copy qua kernel để lấy dữ liệu, dữ liệu được lấy trực tiếp từ bộ nhớ chung giữa 2 tiến trình nên về hiệu suất cơ chế này là nhanh nhất trong việc truyền dữ liệu

Ví dụ: Cùng với 1 file nặng 10MB khi dùng pipe sẽ mất 0,03s còn Shared memory mất 0,002s. Tức là cơ chế này nhanh hơn pipe khoảng 10-20 lần

4.1.4. Ưu và nhược điểm

- **Ưu điểm**
  - Rất nhanh (không copy nhiều).
  - Thích hợp truyền dữ liệu lớn (buffer video, bảng cache, dữ liệu lớn).
- **Nhược điểm**
  - Không đồng bộ tự động → có thể race condition.
  - Phải quản lý khởi tạo/xóa (cleanup).
  - Nếu sai, có thể gây crash hoặc leak resource (vùng shm còn tồn tại).

4.2. Đồng bộ hóa

4.2.1. Vấn đề đồng bộ hóa

- **Khái niệm :** Đồng bộ hóa là kỹ thuật đảm bảo chỉ một hoặc một số luồng được phép truy cập tài nguyên dùng chung tại một thời điểm
- **Vấn đề :**
  - Cần cơ chế phối hợp hoạt động
  - Đảm bảo tính nhất quán dữ liệu
  - Nhiều tiến trình truy cập cùng tài nguyên

Ví dụ : nhiều người cùng rút tiền từ 1 tài khoản; 2 người cùng đặt 1 chỗ khi mua vé….

4.2.2. Race Condition (Tình trạng chạy đua)

- **Khái niệm :** xảy ra khi nhiều tiến trình/ luồng cùng truy cập và thao tác trên tài nguyên dùng chung, mà kết quả cuối cùng phụ thuộc vào thứ tự thực thi của các tiến trình.

→ Kết quả có thể khác nhau mỗi lần chạy

- *Nguyên nhân* : truy cập đồng thời vào tài nguyên dùng chung; không có cơ chế đồng bộ hóa; thao tác không nguyên tử ….

4.2.3. Semaphore

- **Khái niệm :** là một cơ chế đồng bộ hóa trong lập trình đa luồng, hoạt động như một **biến đếm nguyên không âm** được sử dụng để kiểm soát truy cập vào tài nguyên dùng chung.
- **Nguyên lý hoạt động :**
    - Semaphore hoạt động như một **biến đếm**
    - Mỗi khi tiến trình muốn truy cập tài nguyên dùng chung, nó phải **yêu cầu semaphore**
    - Nếu semaphore có giá trị dương → được phép truy cập
    - Nếu semaphore = 0 → phải chờ cho đến khi có tín hiệu
- **Phân Loại Semaphore:**

Binary Semaphore (Semaphore Nhị Phân) :Chỉ có 2 giá trị: 0 và 1 , đảm bảo chỉ có 1 tiến trình truy cập tài nguyên tại một thời điểm

Counting Semaphore (Semaphore Đếm) : Có giá trị nguyên không âm,cho phép nhiều tiến trình truy cập cùng lúc (giới hạn bởi giá trị semaphore)

Ưu điểm:

- Đồng bộ hóa linh hoạt
- Kiểm soát truy cập theo số lượng
- Giải quyết bài toán Producer-Consumer
- Tránh được race condition

Nhược điểm:

- Dễ gây deadlock nếu dùng sai
- Khó debug và kiểm soát
- Có thể gây starvation
- Phức tạp hơn mutex

4.2.4. Bài toán Producer-Consumer(dùng semaphore để đồng bộ)

- Bài toán **Producer-Consumer** (Nhà sản xuất - Người tiêu thụ) là một vấn đề đồng bộ hóa kinh điển, trong đó:
    - **Producer**: Tạo ra dữ liệu và đưa vào buffer
    - **Consumer**: Lấy dữ liệu từ buffer để xử lý
    - **Buffer**: Vùng nhớ dùng chung với kích thước cố định

```
// Ba semaphore cho đồng bộ hóa
sem_t empty;       // Đếm slot trống trong buffer
sem_t full;        // Đếm slot có dữ liệu
sem_t mutex;       // Khóa nhị phân bảo vệ truy cập buffer
```

Luồng Producer:

```
sem_wait(&empty);  // Chờ nếu buffer đầy
sem_wait(&mutex);  // Khóa critical section
// Ghi dữ liệu vào buffer
sem_post(&mutex);  // Mở khóa
sem_post(&full);   // Báo có dữ liệu mới
```

Luồng Consumer:

```
sem_wait(&full);   // Chờ nếu buffer rỗng
sem_wait(&mutex);  // Khóa critical section
// Đọc dữ liệu từ buffer
sem_post(&mutex);  // Mở khóa
sem_post(&empty);  // Báo có chỗ trống
```

Kết quả:

- ➔ **Producer** không được ghi khi buffer đầy
- ➔ **Consumer** không được đọc khi buffer rỗng
- ➔ Tránh **race condition** khi truy cập buffer
- ➔ Đảm bảo hiệu suất không bị **blocked** không cần thiết.

CHƯƠNG 5: SYSTEM CALL CHUYÊN SÂU & BẢO VỆ

5.1. Cách ứng dụng gọi các dịch vụ kernel

Khi một ứng dụng cần gọi dịch vụ kernel linux (system call), quy trình thực hiện qua các bước sau:

- ***Bước 1: Ứng dụng gọi hàm thư viện (Library Function)***
- Ứng dụng chạy ở user mode gọi hàm từ thư viện chuẩn (như read(), write(), open())

- Ví dụ: read(fd, buffer, size) trong C hay write(fd, buffer, size)

- ***Bước 2: Thư viện chuẩn chuẩn bị tham số***

- Thư viện C (glibc trên Linux) đóng gói tham số
- Đặt mã số system call vào thanh ghi đặc biệt (thường là EAX/RAX trên x86)
- Đặt các tham số vào các thanh ghi khác (EBX, ECX, EDX...)

- ***Bước 3: Gọi System Call (Software Interrupt)***

- Thực thi lệnh đặc biệt để chuyển sang kernel mode:
    - int 0x80 (cũ trên x86 32-bit)
    - syscall (x86-64) : trong demo sử dụng linux kernel bản Ubuntu x86-64 bit nên lệnh thực thi sẽ là syscall
    - svc (ARM)
- CPU chuyển từ user mode → kernel mode

- ***Bước 4: CPU chuyển quyền điều khiển cho Kernel***

- CPU lưu trạng thái hiện tại (program counter, stack pointer, registers)
- Nhảy đến system call handler trong kernel
- Kernel có đầy đủ quyền truy cập phần cứng

- ***Bước 5: Kernel xử lý System Call***

- Kernel kiểm tra mã số system call
- Tra bảng system call table để tìm hàm xử lý tương ứng
- Kiểm tra tính hợp lệ của tham số (security checks)
- Thực thi tác vụ yêu cầu (đọc file, cấp phát bộ nhớ, v.v.)

- ***Bước 6: Kernel trả kết quả***

- Đặt giá trị trả về vào thanh ghi (EAX/RAX)
- Khôi phục trạng thái của process
- Chuyển quyền điều khiển về user mode

- ***Bước 7: Ứng dụng nhận kết quả***

- Thư viện C nhận giá trị trả về từ thanh ghi
- Xử lý lỗi nếu có
- Trả kết quả về cho ứng dụng

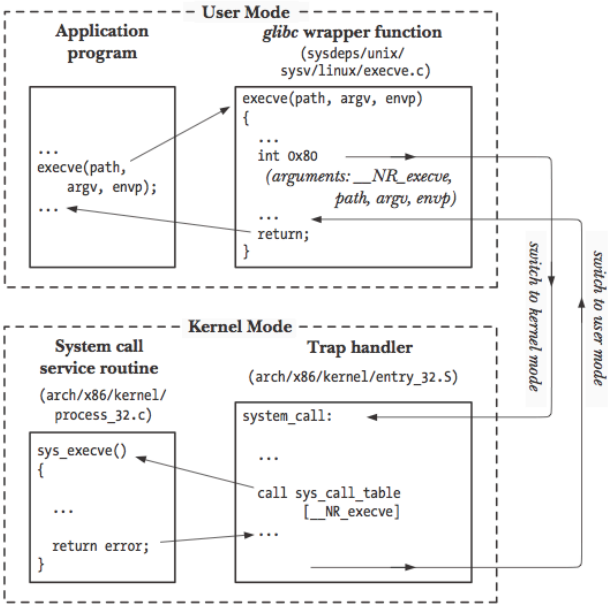

Figure 3-1: Steps in the execution of a system call

5.2. Sử dụng strace để theo dõi system call

strace là công cụ cho phép ghi lại toàn bộ system call mà tiến trình thực hiện trên linux, bao gồm các thao tác về bộ nhớ, file, signal, thread, exit…

- Gõ strace ./<file_name>

Ví dụ: chương trình in ra dòng chữ "Hello, World!"

```
lexua@lexua-VMware-Virtual-Platform:~/Desktop/hdh_demo$ strace ./main
mmap(0x7d766d628000, 1605632, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000)
7d766d628000
mmap(0x7d766d7b0000, 323584, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b0000) = 0x7d766d
00
mmap(0x7d766d7ff000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1fe000)
7d766d7ff000
mmap(0x7d766d805000, 52624, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7d
805000
close(3)                                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7d766d8e4000
arch_prctl(ARCH_SET_FS, 0x7d766d8e4740) = 0
set_tid_address(0x7d766d8e4a10)         = 4663
set_robust_list(0x7d766d8e4a20, 24)     = 0
rseq(0x7d766d8e5060, 0x20, 0, 0x53053053) = 0
mprotect(0x7d766d7ff000, 16384, PROT_READ) = 0
mprotect(0x5d077dd7e000, 4096, PROT_READ) = 0
mprotect(0x7d766d936000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
munmap(0x7d766d8e7000, 65911)           = 0
write(1, "Hello, World!\n", 14Hello, World!
)         = 14
exit_group(0)                           = ?
```

Giải thích các lời gọi hàm được thực hiện trong kết quả của strace:

- mmap loader: nạp thư viện
- close: đóng file
- arch_prctl, set_tid_address, set_robust_list: thiết lập thread
- openat, read: chương trình đọc /etc/localtime
- write: ghi chuỗi "Hello, World!" ra stdout
- exit_group: kết thúc chương trình

5.3. Phân tích UID/GID, quyền truy nhập file , vai trò kiểm soát của SC

Trong hệ điều hành Linux, mỗi tiến trình và tập tin đều gắn với UID(user id) và GID(group id) dùng để xác định chủ sở hữu và nhóm quyền hạn.

- **UID:**định danh cho người dùng sở hữu tệp hoặc tiến trình.
- **GID:** định danh cho nhóm mà người dùng thuộc về.

Quyền truy cập được chia thành ba loại đối tượng:

- **Owner** (chủ sở hữu)
- **Group** (nhóm)
- **Others** (người dùng còn lại)

Với mỗi loại, bộ quyền bao gồm:

- **r** (read) – cho phép đọc nội dung tệp
- **w** (write) – cho phép sửa/xóa
- **x** (execute) – cho phép chạy tệp hoặc truy cập thư mục

Hệ thống bảo vệ được thực thi khi một tiến trình thực hiện thao tác. Kernel sẽ so khớp:

1. UID của tiến trình
2. UID của file
3. GID của tiến trình
4. Quyền r/w/x tương ứng.

→ Từ đó quyết định cho phép hay từ chối truy cập.
Trong IPC, cơ chế này đảm bảo rằng tiến trình không được phép đọc hoặc ghi vào tài nguyên IPC (Shared Memory, FIFO, Message Queue…) nếu không có quyền.

Vai trò kiểm soát của SC:

1. Kiểm soát quyền truy nhập
2. Kiểm tra tham số và xác thực
3. Chuyển từ user mode sang kernel mode một cách an toàn
4. Ghi log và audit (trong các hệ thống bảo mật cao).

5.4. Kernel module và sự tương tác với System call

5.4.1. Khái niệm Kernel Module

Kernel Module (module nhân) là các đoạn code có thể được load/unload động vào Linux kernel mà không cần khởi động lại hệ thống, giúp mở rộng chức năng của kernel khi cần thiết.

Các loại module phổ biến:

- Device drivers (USB, network cards, sound cards)
- Filesystem drivers (ext4, NTFS)
- Network protocols
- Security modules

5.4.2. Tương tác giữa Kernel Module và System call

Cách Kernel Module tương tác với System Call:

- User Application (User Space): Chương trình người dùng gọi hàm hệ thống (như read(), write(), ioctl()) -> yêu cầu chuyển xuống kernel.
- System Call Interface: Lời gọi hệ thống được chuyển vào bảng sys_call_table để xác định hàm xử lý trong kernel.
- Kernel (Kernel Space): Kernel nhận system call, kiểm tra tham số, rồi chuyển yêu cầu đến driver hoặc kernel module tương ứng.
- Kernel Module / Device Driver: Module xử lý yêu cầu thông qua các hàm trong file_operations (ví dụ: my_read(), my_write(), my_ioctl()).
  Nếu cần, module sẽ giao tiếp với phần cứng.
- Hardware: Thiết bị thực hiện thao tác (đọc/ghi/điều khiển)

-> trả dữ liệu ngược lên phía user space: module -> kernel -> system call -> chương trình người dùng.

CHƯƠNG 6: ĐÁNH GIÁ, KẾT LUẬN

6.1. Bảng so sánh IPC

IPC	Tốc độ	Đồng bộ?	Ưu điểm chính	Nhược điểm chính
Shared Memory	Siêu nhanh	Không đồng bộ	Nhanh nhất, không copy dữ liệu	Phải tự đồng bộ (dùng semaphore)
Anonymous Pipe	Rất nhanh	Đồng bộ	Cực kỳ đơn giản, an toàn	Chỉ parent-child, 1 chiều
Named Pipe (FIFO)	Rất nhanh	Đồng bộ	Dùng được cho mọi process	1 chiều, phải tạo file trước
Message Queue	Trung bình	Có/Không đồng bộ	Gửi message độc lập, có ưu tiên	Chậm hơn + giới hạn kích thước message
Signal	Siêu nhanh	Không đồng bộ	Gửi thông báo tức thì, nhẹ nhất	Chỉ truyền được số (1–64)
Semaphore	Nhanh	Công cụ đồng bộ	Bảo vệ critical section đáng tin cậy	Không dùng để truyền dữ liệu

6.2 Ứng dụng thực tiễn tổng thể: ví dụ Microservices và vai trò của IPC

Microservices là mô hình chia hệ thống thành nhiều dịch vụ nhỏ , mỗi dịch vụ chạy độc lập. Để các dịch vụ phối hợp với nhau chúng phải giao tiếp qua IPC.

Ví dụ Microservice : hệ thống đặt hàng online

Hệ thống gồm 4 service tách rời

1. Order service:nhận yêu cầu từ khách hàng.
2. Inventory service: kiểm tra số lượng tồn kho.
3. Payment Service: xử lý thanh toán.
4. Notification Service: Gửi email/sms thông báo.

Luồng hoạt động :

1. Người dùng gửi yêu cầu tạo đơn sau đó Order Service nhận
2. Order Service gọi Inventory Service để kiểm tra còn hàng không
3. Nếu còn hàng, Order Service tạo đơn và gửi ordercreated.
4. Payment Service nhận và xử lý thanh toán.
5. Notification Service nhận kết quả sau đó gửi email thông báo

Vai trò của IPC:

- Trao đổi dữ liệu giữa các service.
- Gọi chức năng của service khác.
- Đảm bảo luồng xử lý liền mạch dù chạy ở nhiều tiến trình khác nhau.
- Tăng khả năng mở rộng.

6.3. Kết luận

Trong bài tập lớn này, nhóm đã tìm hiểu các cơ chế giao tiếp giữa các tiến trình (IPC) và quá trình thực thi System Call trong Linux. Các cơ chế như Pipe, FIFO, Message Queue, Signal, Shared Memory và Semaphore được phân tích rõ về cách hoạt động và phạm vi ứng dụng. Nhóm cũng sử dụng *strace* để quan sát trực tiếp quá trình gọi hệ thống của chương trình, qua đó hiểu rõ hơn cách User Mode tương tác với Kernel Mode.

Ngoài ra, trong phần slide sẽ xây dựng và trình bày cây dự án kèm hình minh họa, giúp mô tả cấu trúc tổ chức file và các phần liên quan.

Từ đó, nhóm rút ra rằng IPC và System Call là nền tảng quan trọng của Hệ điều hành, hỗ trợ tiến trình phối hợp hiệu quả và đảm bảo truy cập tài nguyên an toàn.

Link code demo: https://github.com/AnhTaizz/BTL_He_Dieu_Hanh

Link slide:
https://www.canva.com/design/DAG5IaYZiHY/i_MWNFQDDP3wDhi-UB-D7g/edit