**Bài 1: Tiến trình**

**I. Tiến trình**

**1. Tiến trình (process)**: tiens trình con sao chép toàn bộ không gian địa chỉ của cha( ví dụ tên biến,…) thực thi cùng mã nguồn như tiến trình cha và bắt đầu chạy mã nguồn riêng , và có vùng không gian bộ nhớ riêng biệt với tiến trình cha

**2. Luồng (thread)** đó:1 luồng thực thi mã nguồn trong 1 chương trình, chỉ có ý nghĩa trong tiến trình đó, chia sẻ dữ liệu sử dụng trong tiến trình

**\*\*\*\*\*Tuy nhiên kernel sẽ mặc định phân chia đều tài nguyên cho các tiến trình mặc dù khác số luồng**

**3. Tiến trình nhẹ:** là 1 tiến trình, nhưng thay vì có không gian địa chỉ, tài nguyên riêng biệt thì chúng có thể chia sẻ tài nguyên cho nhau như luồng. Kernel sẽ coi nó như 1 tiến trình bình thường, cung cấp tài nguyện, lập lịch làm việc cho tiến trình nhẹ

=>> cơ bản tạo 1 tiến trình đa luồng là cho mỗi luồng tạo ra 1 tiến trình nhẹ,=>> ko lo khi 1 luồng thực hiện blocking syscall, thì luồng còn lại ko bị ảnh hưởng

o

ví dụ về ứng dụng chơi cờ: Nếu gán hai luồng của ứng dụng vào hai tiến trình nhẹ, thì dùng luồng một (chờ đợi các nước đi của người chơi để thể hiện trên hình ảnh) có thực thi một blocking system call, thì luồng thứ hai (tính toán các nước đi tiếp theo của máy) sẽ vẫn được thực hiện liên tục do kernel sẽ lập lịch làm việc cho luồng thứ hai.

**II. Quản lý tiến trình**

1.Mối quan hệ:

- real parent: trỏ tới bộ mo tả tiến trình tạo ra P, hoặc trỏ tới tiến trình init nếu cha của P ko tồn tại.

-Parent: trỏ tới cha hiện tại của P

-childrent: trỏ tới danh sách các tiến trình liên kết tạo bởi P

-sibling: trỏ tới danh sách các tiến trình có cùng cha vs P

**2. Tạo tiến trình**

-Thông thường thì tạo 1 tiến trình con, toàn bộ tài nguyên tiến trình cha sẽ sao chép cho tiến trình con => chậm

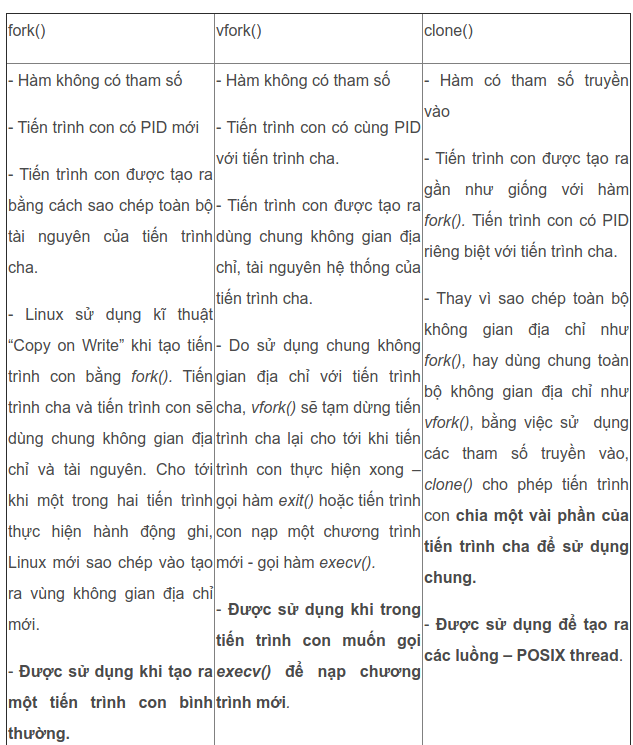
=> Các cách khác:

+Copy on write: khi con đc tạo ra, sẽ ko copy mà dùng chung không gian địa chỉ vs cha. Chỉ dùng chung khi chỉ có đọc. KHi thực hiện ghi,=> kernel mới sao chép toàn bộ nội dung của cha sang không gian của con

+Tiến trình nhẹ: như trên;

+vfork() system call: tạo ra 1 tiến trình con dùng không gian địa chỉ của cha, và tiến trình cha sẽ bị khóa cho tới khi con kết thúc hoặc tiến trình con tạo 1 chương trình mới

-Các lệnh tạo tiến trình:



**3. Kết thúc tiến trình**

**-**Các lệnh kêts thúc:

+ exit\_group: kết thúc toàn bộ các luồng của 1 tiến trình

+exit(): chỉ kết thúc tiến trình đó

- Tiến trình cha có thể kiểm tra các trạng thái của các tiến trình con của nó

+ wait: dừng chương trình cha cho tới khi tiến trình con nó kết thúc. Với việc kiểm tra giá trị trả về của con => có biết thành công hay ko

-

=> kernel ko có phép xóa dữ liệu tiến trình khi nó kết thúc, chỉ xóa khi tiến trình cha gọi wait() => trạng thái zombie

-Khi cha kết thúc trước => con sẽ chuyển về làm con của tiến trình int

- KHi cha ko gọi wait90 => tiến trình con thành zombie’

**- Giải quyết tránh gọi wait() trong tiến trình cha liên t**

+ fork 2 lần: (safe fork):

**Bài 2: Ngắt**

**I. Ngắt và ngoại lệ**

- Ngắt có 2 loại: đồng bộ( do CPU tạo ra) hoặc ko đồng bộ( do nguồn khác)

→ ko đồng bộ gọi là Interrupt, đồng bộ là ngoại lệ

**1. Interrupt**

**-** 2 trạng th:

+ ko che: 1 sự kiện khẩn cấp ( lỗi phần cứng) bắt CPU xử lý luôn

+ che: (các thiết bị vào ra)

=>CPU chỉ xử lý ngắt ko che

**Bài 4: Đồng bộ trong Linux Kernel**

**Đồng bộ ở đây là đồng bộ dữ liệu dùng chung giữa các tiến trình hoặc các luồng hoặc các CPU**

**I. Các phương pháp đồng bộ**

1. Per-CPU Variable

- Là dữ liệu tạo riêng cho mỗi CPU, đc lưu ở phần bộ nhớ dành riêng cho mỗi CPU

- Nhược điểm là bộ nhớ dành riêng cho mỗi CPU là hữu hạn

- Có 2 loại :cấp phát tĩnh ( ở thời điểm biên dịch) và cấp phát động ( ở thời điểm runtime)

- Ko thể gọi trực tiếp mà phải gọi thông qua hàm riêng

2. Atomic Operations

- Cơ chế của CPU là read-modify-write

-Atomic bằng cách cung cấp các câu lệnh asm có opcode chứa lock byte 0xf0, tức là khi CPU thực hiện 1 câu lệnh ass có chứa lock byte, nó sẽ khóa đường bus truy cập bộ nhớ memory bus cho tới khi thực hiện xong câu lệnh đó

- Linux cung cấp phép toán **atomic** để đảm bảo trình biên dịch tạo ra lệnh atomic, có kiểu dữ liệu là **atomic\_t ,**

- 1 số phép toán thực hiện vs atomic : atomic\_read(v), atomic\_set(v,i), atomic\_add(i,v),….

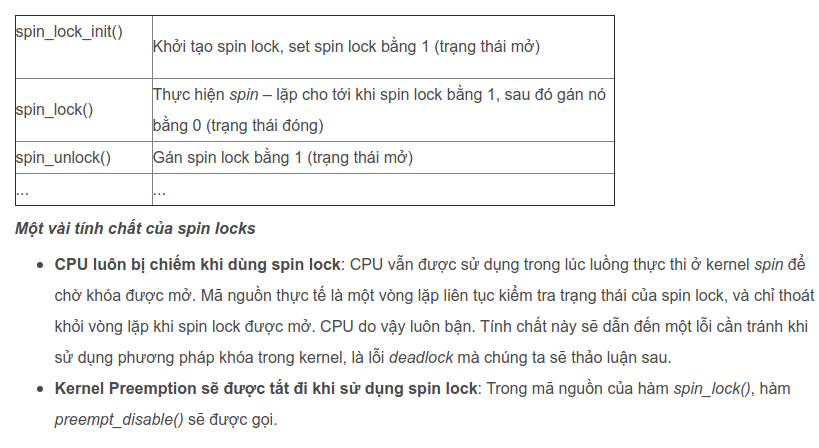
**3. Spin Locks(dùng nhiều)**

**- Như nhà vệ sinh, mỗi thời điểm chỉ 1 người vào và sẽ khóa lại**

**-**KHi 1 luồng thực thi ở kernel thấy spin lock đang mở thì nó sẽ giành lấy khóa và thực thi.

- KHi spin lock đóng, nó sẽ spin quay vòng, lặp đi lặp lại cho tới khi spin lock mở và giảnh lấy khóa và tiếp tục thực thi

- Trong kernel, spin lock đc thể hiển bằng kiểu dữ liêu **spinlock\_t.**  Các api đi kèm như sau



**Read/Write Spin locks**

**-**  Mỗi thời điểm chỉ có 1 luồng truy cập vào vùng CR, nếu chỉ có read thì có thể nhiều luồng vào => 2 chế độ hoạt động Read/Write Spinlock

+Khóa Read cho phép các luồng chỉ read đc truy câp

+ Khóa Write thì ko cho phép vào khi đang spinlock đang đóng

**Quản lý bộ nhớ**