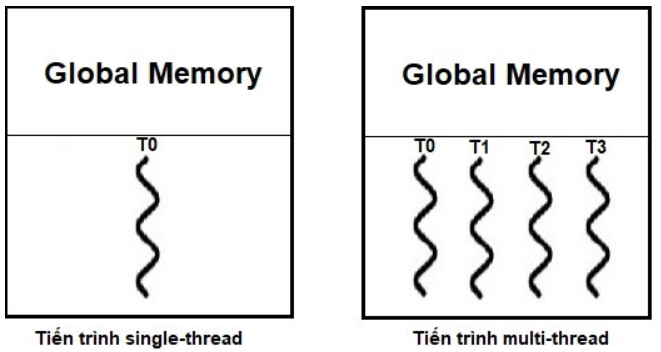
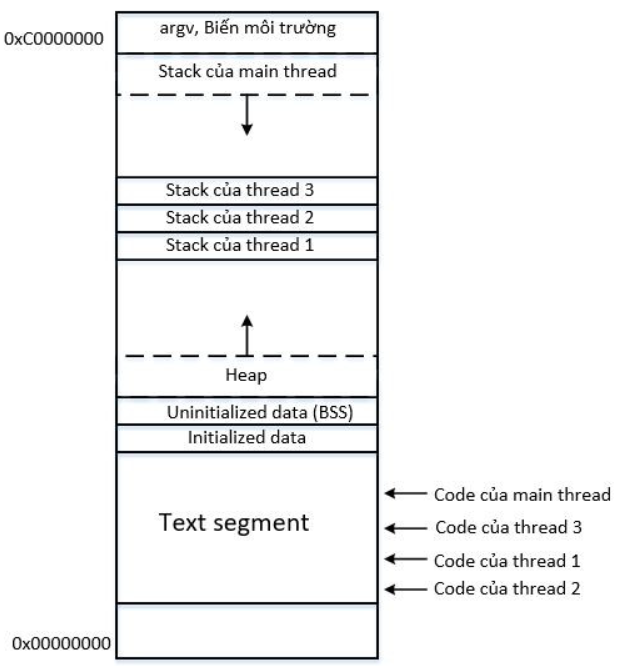
**Thread và Posix Thread**

Luồng là một thành phần của tiến trình, một tiến trình có thể chứa một hoặc nhiều thread. Hệ điều hành Unix quan niệm rằng mỗi tiến trình khi bắt đầu chạy luôn có một thread chính (main thread); nếu không có thread nào được tạo thêm thì tiến trình đó được gọi là đơn luồng (single – thread), ngược lại nếu có thêm thread thì được gọi là đa luồng (multi – thread). **Các thread trong tiến trình chia sẻ các vùng nhớ toàn cục (global memory) của tiến trình bao gồm initialized data, uninitialized data và vùng nhớ heap**.

Hình vẽ dưới đây mô tả về một tiên trình đơn luồng (single – thread) và đa luồng (multi – thread):



Trong hình vẽ trên, một tiến trình có 4 thread, bao gồm 1 main thread (T0) được tạo ra khi tiến trình chạy hàm main (), **và 3 thread lần lượt là T1, T2 và T3 được tạo mới trong hàm main(). Bốn thread sử dụng chung vùng nhớ toàn cục (global memory) nhưng mỗi thread có phân vùng stack riêng của mình**, cụ thể như hình vẽ dưới đây.



Các thread trong tiến trình có thể thực thi đồng thời và độc lập với nhau. Nghĩa là nếu một thread bị block do đang chờ I/O thì các thread khác vẫn được lập lịch và thực thi thay vì tiến trình bị block.

**So sánh Process và Thread**

Quay trở lại ví dụ trên, tiến trình server tạo ra các tiến trình con để phục vụ yêu cầu multi-task. Cách giải này tuy giải quyết được yêu cầu nhưng tồn tại hạn chế sau đây:

- Việc chia sẻ dữ liệu giữa các tiến trình khá khó khăn. Vì mỗi tiến trình trong Linux có không gian bộ nhớ riêng biệt nên chúng ta phải sử dụng các phương pháp giao tiếp liên tiến trình (IPC) như shared memory, message queue, socket…. Để chia sẻ dữ liệu.

- Tạo ra một tiến trình mới bằng system call fork() khá “tốn kém” về mặt tài nguyên cũng như thời gian vì phải tạo ra các vùng nhớ riêng biệt cho tiến trình con. Điều này khá quan trọng trong các hệ thống embedded có phần cứng hạn chế.

Thread có thể giải quyết được 2 vấn đề trên vì có các ưu điểm sau:

- Chia sẻ dữ liệu giữa các thread trong tiến trình rất đơn giản vì chúng có chung không gian bộ nhớ toàn cục. Do vậy, chỉ cần tạo dữ liệu ở trong các vùng nhớ toàn cục này thì các thread đều có thể truy xuất được.

- Việc tạo ra một thread mới nhanh hơn đáng kể so với việc tạo ra một tiến trình mới vì **các thread dùng chung nhiều phần không gian bộ nhớ nên chỉ cần tạo không gian bộ nhớ cho những phần riêng thay vì phải nhân bản toàn bộ các vùng nhớ như khi tạo tiến trình con.**

Hiển nhiên thread cũng không phải là chìa khóa vạn năng. Việc sử dụng thread cũng có các nhược điểm sau:

- Vì các thread dùng chung vùng nhớ toàn cục nên việc lập trình trên các thread "nguy hiểm" hơn trên process. Nếu một thread gây ra lỗi trên vùng nhớ toàn cục thì sẽ kéo theo các thread khác cũng bị lỗi theo.

- Các thread cùng chia sẻ vùng nhớ toàn cục của một tiến trình (3 GB với hệ thống 32 bit), cụ thể mỗi thread sẽ được cung cấp một vùng nhớ riêng trong tổng thể bộ nhớ của tiến trình. Bộ nhớ của tiến trình tuy là lớn nhưng cũng là một số hữu hạn. Nên một tiến trình cũng bị giới hạn bởi số lượng thread có thể tạo ra hoặc tạo ra các thread cần bộ nhớ lớn.

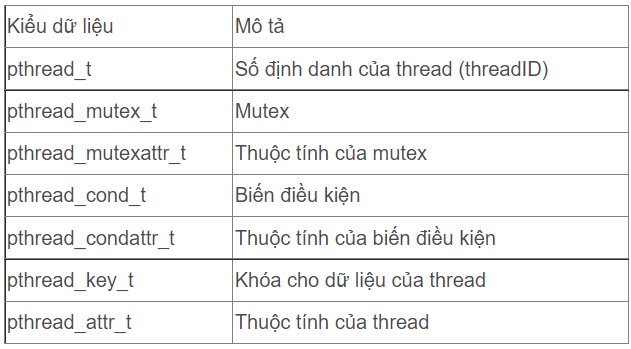
Cả hai nhược điểm trên không xảy ra trên tiến trình vì mỗi tiến trình có không gian bộ nhớ riêng.

**2. Posix Thread**

Quay lại thời điểm sơ khai của thread, khi đó mỗi nhà cung cấp phần cứng triển khai thread và cung cấp các API để lập tình thread của riêng mình. Điều này gây khó khăn cho các developer khi phải học nhiều phiên bản thread và viết 1 chương trình thread chạy đa nền tảng phần cứng. Trước như cầu xây dựng một giao diện lập trình thread chung, tiêu chuẩn POSIX Thread (pthread) cung cấp các giao diện lập trình thread trên ngôn ngữ C/C++ đã ra đời.

**2.1 Pthread data type**

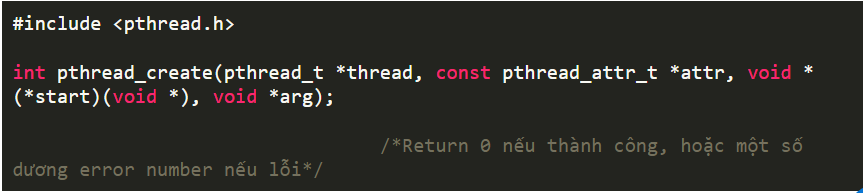
Trước khi bắt tay vào khám phá các API của pthread, chúng ta cần lướt qua một số kiểu dữ liệu pthread định nghĩa riêng dưới đây:



Các bạn đừng lo lắng nếu thấy các kiểu dữ liệu này lạ lẫm, các kiểu dữ liệu cần thiết sẽ được giải thích dần ở các bài sau. Trong bài này, chúng ta chỉ cần quan tâm đến kiểu pthread\_t là một số định danh cho thread sẽ được giải thích ở bên dưới.

**2.2 Tạo thread mới**

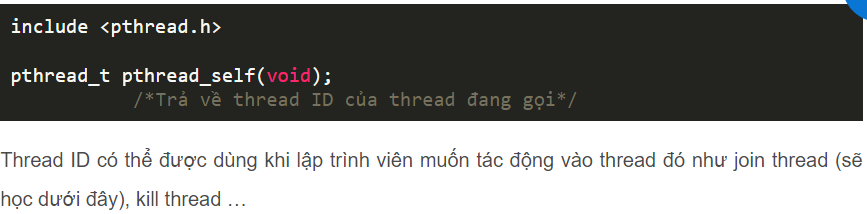
Để tạo ra một thread mới, ta sử dụng hàm pthread\_creadte() với prototype như sau:



Hàm pthread\_create() tạo ra một thread mới trong tiến trình, entry point của thread mới này hàm “start()” với đối số là “arg” (start(arg)). Main thread của tiến trình tiếp tục thực thi với các câu lệnh sau hàm pthread\_create() đó. Đối số “arg” được truyền vào có kiểu void, nghĩa là ta có thể truyền bất kỳ kiểu dữ liệu nào vào hàm start(), hoặc truyền vào con trỏ NULL nếu hàm start() không cần đối số. **Nếu muồn truyền nhiều đối số vào hàm start(), ta có thể khai báo đối số “arg” dưới dạng một con trỏ trỏ đến một cấu trúc với các đối số là các trường hợp riêng biệt (chúng ta sẽ xét một ví dụ bên dưới để rõ hơn về cách này).**

**2.3 Thread ID**

Mỗi thread trogn tiến trình có 1 số định danh duy nhất là thread ID. Thread ID trong Posix có kiểu dữ liệu là pthread\_t, chính là giá trị pthread\_t \*thread được gán vào trong hàm pthread\_create() ở trên. Ta có thể kiểm tra được thread ID của thread đang chạy bằng hàm pthread\_self() với prototype như sau:



**2.4 Kết thúc thread**

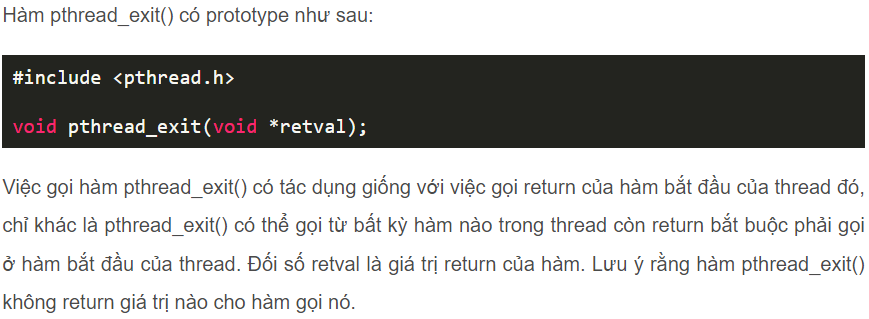
Một thread đang thực thi có thể được kết thúc bằng một trong các cách sau:

- Hàm bắt đầu của thread thực thi câu lệnh return.

- Một hàm bất kỳ trong thread gọi hàm pthread\_exit(), chúng ta sẽ nói về hàm này ở dưới đây.

- Thread bị hủy bỏ bằng hàm pthread\_cancel()

- Một thread bất kỳ của tiến trình gọi hàm exit() hoặc thread chính của tiến trình (hàm main()) gọi return. Cả 2 cách này đều có tác dụng kết thúc tiến trình đang chạy và tất nhiên cả các thread của tiến trình đó.



**LẬP TRÌNH ĐA LUỒNG (MULTI – THREAD)**

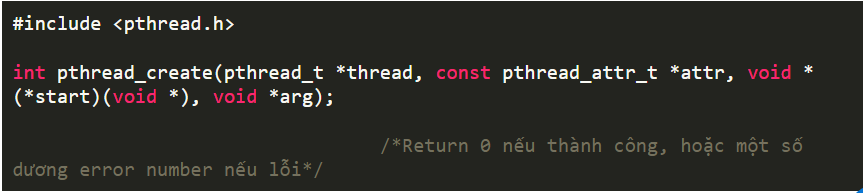
**1. Luồng là gì? Tại sao phải dùng luồng?**

Luồng là một phần của tiến trình sở hữu riêng ngăn xếp (stack) và thực thi độc lập ngay trong mã lệnh của tiến trình. Nếu như một HĐH có nhiều tiến trình thì bên trong mỗi tiến trình lại có thể tạo ra nhiều luồng hoạt động song song với nhau.

Ưu điểm của tuyến là chúng hoạt động trong cùng không gian địa chỉ của tiến trình. Cơ chế liên lạc giữa các tuyến đơn giản và hiệu quả.

**2. Tạo lập và hủy luồng.**

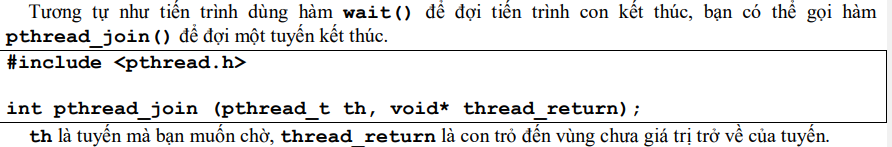
Khi chương trình chính bắt đầu, nó chính là một luồng. Luồng điều khiển hàm main() được gọi là luồng chính. Các luồng khác do tiến trình tạo ra sau đó được gọi là luồng phụ. Mỗi luồng được cung cấp cho một số định danh gọi là thread ID. Để tạo ra một luồng mới ngoài luồng chính, ta gọi hàm pthread\_create(). Hàm này được khai báo như sau:



Hàm pthread\_create () nhận 4 tham số, tham số thứ nhất có kiểu cấu trúc pthread\_t để lưu các thông tin về luồng sau khi tạo ra. **Tham số thứ hai dùng để đặt thuộc tính cho luồng (trong trường hợp ta đặt giá trị NULL thì luồng được tạo ra với các thuộc tính mặc định).** Tham số thứu ba là địa chỉ của hàm mà tuyến sẽ dùng để thực thi.

**3. Chờ tuyến kết thúc**

**a. Chờ tuyến hoàn thành xong tác vụ**



**b. Chờ đồng thời nhiều tuyến**

Thường trong các ứng dụng dịch vụ hoạt động theo mô hình khách chủ (client / server), trình chủ (server) của bạn phải mở nhiều luồng để phục vụ trình khác. Hay trong các ứng dụng trò chơi bạn phải mở cùng lúc nhiều luồng, mỗi luồng thực hiện thao tác điều khiển một nhân vật hoạt hình nào đó. Kiểm soát và chờ đồng thời nhiều tuyến, bạn cũng dùng hàm **pthread\_join().**

**4. Đồng bộ hóa tuyến với đối tượng mutex**

Một trong những vấn đề quan tâm hàng đầu của việc điều khiển lập trình đa luồng trong cùng không gian đại chỉ của tiến trình đó là đồng bộ hóa. Bạn phải đảm bảo được nguyên tắc các tuyến không dẫm chân lên nhau. Ví dụ một tuyến chuẩn bị để đọc dữ liệu từ đĩa, thao tác đọc chưa hoàn tất thì một luồng khác đã ghi đè dữ liệu mới lên dữ liệu cũ. Hay đơn giản và thường gặp hơn đó là xảy ra đụng độ khi truy cập và xử lý biến chung.

**Để giải quyết tranh chấp và xử lý đồng bộ hóa chúng ta sử dụng một khái niệm gọi là mutex.**

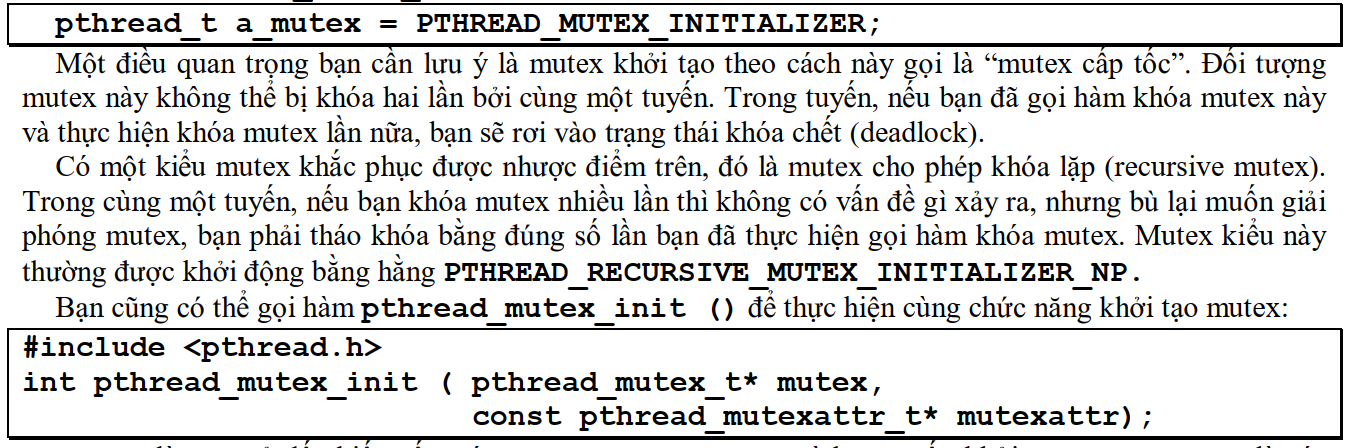
**a. Mutex là gì**

**Mutex thực sự là một cờ hiệu, hay đối với hệ thống, mutex là một đối tượng mang hai trạng thái: đang được sử dụng và chưa được sử dụng (trạng thái sẵn sàng).**

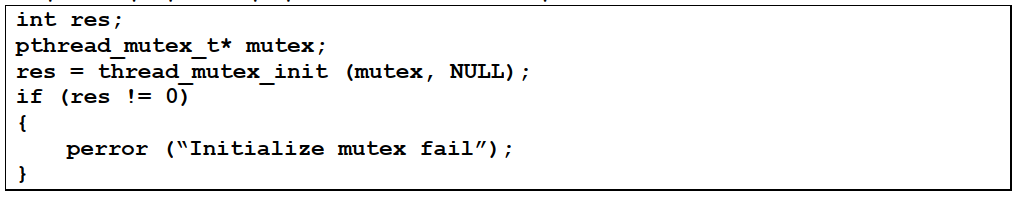
**Khi mutex bật, một tuyến sẽ bước vào sử dụng tài nguyên và tắt mutex. Tuyến khác sẽ không sử dụng được tài nguyên cho đến khi tuyến trước đó được bật lại mutex ở trạng thái sẵn sàng.**

**b. Tạo và khởi động mutex**

**Để tạo ra đối tượng mutex, trước hết bạn cần khai báo biến kiểu cấu trúc pthread\_mutex\_t, đồng thời khởi tạo giá trị ban đầu cho biến này.** Cách đơn giản nhất để khởi tạo cấu trúc mutex là dùng hằng định nghĩa trước PTHREAD\_MUTEX\_INITIALIZER. Mã khai báo mutex thường có dạng sau:

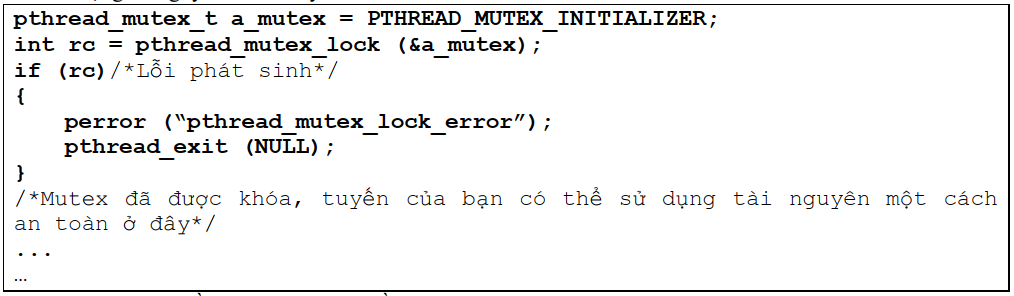


Mutex là con trỏ đến biến cấu trúc pthread\_mutex\_t mà bạn muốn khởi tạo. mutexattr là các thuộc tính của mutex (mutex đơn hay mutex cho phép khóa lặp). Nếu bạn đặt trị NULL thì mutex với thuộc tính mặc định sẽ được tạo ra. Cách thứ hai để khởi tạo mutex sẽ là:

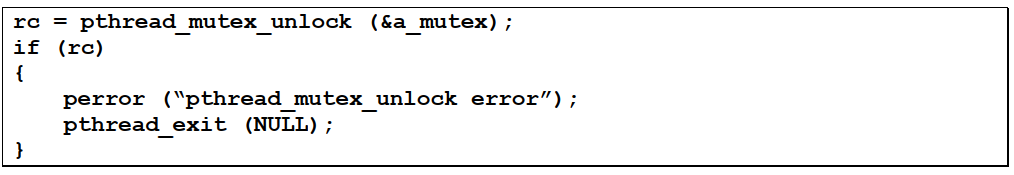


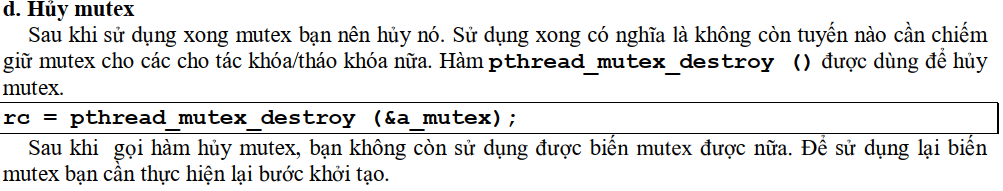
**c. Khóa và tháo khóa cho mutex**

Để khóa mutex bạn có thể sử dụng hàm **pthread\_mutex\_lock(),** nếu không khóa được (mutex đã bị tuyến khác khóa trước đó) hàm sẽ đặt tuyến hiện hành vào trạng thái ngủ (chờ). Trong trường hợp này khi mutex được tháo khóa, tuyến hiện hành sẽ được đánh thức dậy để tiếp tục thử khóa mutex trước khi đi vào sử dụng tài nguyên. Dưới đây là cách khóa mutex:



Một khi không cần sử dụng độc quyền tài nguyên nữa, bạn nên gọi hàm **pthread\_mutex\_unlock()** để tháo khóa mutex trả lại quyền sử dụng tài nguyên cho tuyến khác. Bạn tháo khóa mutex như sau:





Bài 1: Chương trình tạo lập tuyến: chúng ta tạo hàm do\_loop() để in ra các số nguyên. Hàm do\_loop() này được gọi thwucj thi ở hai nơi: một trong tuyến chính (hàm main) và một trong tuyến phụ tạo ra bởi hàm pthread\_create();

