Bài 3: XỬ LÝ TIẾN TRÌNH TRONG LINUX

I. Lý Thuyết

1. Khái quát

- Một trong những đặc điểm nổi bật của Linux là khả năng chạy đồng thời nhiều chương trình. Hệ Điều Hành xem mỗi đơn thể mã lệnh mà nó điều khiển là tiến trình (process). Một chương trình có thể bao gồm nhiều tiến trình kết hợp với nhau.
- Đối với Hệ Điều Hành, các tiến trình cùng hoạt động chia sẻ tốc độ xử lý của CPU, cùng dùng chung vùng nhớ và tài nguyên hệ thống khác. Các tiến trình được điều phối xoay vòng bởi Hệ Điều Hành. Một chương trình của chúng ta nếu mở rộng dần ra, sẽ có lúc cần phải tách ra thành nhiều tiến trình để xử lý những công việc độc lập với nhau. Các lệnh của Linux thực tế là những lệnh riêng lẻ có khả năng kết hợp và truyền dữ liệu cho nhau thông qua các cơ chế như : đường ống pipe, chuyển hướng xuất nhập (redirect), phát sinh tín hiệu (signal), … Chúng được gọi là cơ chế giao tiếp liên tiến trình (IPC – Inter Process Comunication). Đối với tiến trình, chúng ta sẽ tìm hiểu cách tạo, hủy, tạm dừng tiến trình, đồng bộ hóa tiến trình và giao tiếp giữa các tiến trình với nhau.
- Xây dựng ứng dụng trong môi trường đa tiến trình như Linux là công việc khó khăn. Không như môi trường đơn nhiệm, trong môi trường đa nhiệm tiến trình có tài nguyên rất hạn hẹp. Tiến trình của chúng ta khi hoạt động phải luôn ở trạng thái tôn trọng và sẵn sàng nhường quyền xử lý CPU cho các tiến trình khác ở bất kỳ thời điểm nào, khi hệ thống có yêu cầu. Nếu tiến trình của chúng ta được xây dựng không tốt, khi đổ vỡ và gây ra lỗi, nó có thể làm treo các tiến trình khác trong hệ thống hay thậm chí phá vỡ (crash) Hệ Điều Hành.
- **Định nghĩa của tiến trình**: là một thực thể điều khiển đoạn mã lệnh có riêng một không gian địa chỉ, có ngăn xếp stack riêng rẽ, có bảng chứa các thông số mô tả file được mở cùng tiến trình và đặc biệt có một định danh PID (Process Identify) duy nhất trong toàn bộ hệ thống vào thời điểm tiến trình đang chạy.

Như chúng ta đã thấy, tiến trình không phải là một chương trình (tuy đôi lúc một chương trình đơn giản chỉ cần một tiến trình duy nhất để hoàn thành tác vụ, trong trường hợp này thì chúng ta có thể xem tiến trình và chương trình là một). Rất nhiều tiến trình có thể thực thi trên cùng một máy với cùng một Hệ Điều Hành, cùng một người dùng hoặc nhiều người dùng đăng nhập khác nhau. Ví dụ shell bash là một tiến trình có thể thực thi lệnh **`ls`** hay **`cp`**. Bản thân **`ls`**, **`cp`** lại là những tiến trình có thể hoạt động tách biệt khác.

- Trong Linux, tiến trình được cấp không gian địa chỉ bộ nhớ phẳng là 4GB. Dữ liệu của tiến trình này không thể đọc và truy xuất được bởi các tiến trình khác. Hai tiến trình khác nhau không thể xâm phạm biến của nhau. Tuy nhiên, nếu chúng ta muốn chia sẻ dữ liệu giữa hai tiến trình, Linux có thể cung cấp cho chúng ta một vùng không gian địa chỉ chung để làm điều này.

2. Cách hoạt động của tiến trình

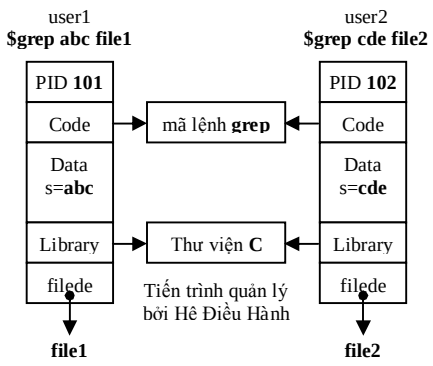
- Khi 1 chương trình đang chạy từ dòng lệnh, chúng ta có thể nhấn phím **`Ctrl+z`** để tạm dùng chương trình và đưa nó vào hoạt động phía hậu trường (background). Tiến trình của Linux có các trạng thái:

+ Đang chạy (running) : đây là lúc tiến trình chiếm quyền xử lý CPU dùng tính toán hay thực các công việc của mình.
+ Chờ (waiting) : tiến trình bị Hệ Điều Hành tước quyền xử lý CPU, và chờ đến lược cấp phát khác.
+ Tạm dừng (suspend) : Hệ Điều Hành tạm dừng tiến trình. Tiến trình được đưa vào trạng thái ngủ (sleep). Khi cần thiết và có nhu cầu, Hệ Điều Hành sẽ đánh thức (wake up) hay nạp lại mã lệnh của tiến trình vào bộ nhớ. Cấp phát tài nguyên CPU để tiến trình tiếp tục hoạt động.

- Trên dòng lệnh, thay vì dùng lệnh Ctrl+z, chúng ta có thể sử dụng lệnh **`bg`** để đưa một tiến trình vào hoạt động phía hậu trường. Chúng ta cũng có thể yêu cầu 1 tiến trình chạy nền bằng cú pháp **`&`**. Ví dụ: **`$ls –R &`**
Lệnh **`fg`** sẽ đem tiến trình trở về hoạt động ưu tiên phía trước. Thực tế khi chúng ta đăng nhập vào hệ thống và tương tác trên dòng lệnh, cũng là lúc chúng ta đang ở trong tiến trình shell của bash. Khi gọi một lệnh có nghĩa là chúng ta đã yêu cầu bash tạo thêm một tiến trình con thực thi khác. Về mặt lập trình, chúng ta có thể dùng lệnh **`fork()`** để nhân bản tiến trình mới từ tiến trình cũ. Hoặc dùng lệnh **`system()`** để triệu gọi một tiến trình của Hệ Điều Hành. Hàm **`exec()`** cũng có khả năng tạo ra tiến trình mới khác.

3. Cấu trúc tiến trình

- Chúng ta hãy xem Hệ Điều Hành quản lý tiến trình như thế nào? Nếu có hai người dùng: **`user1`** và **`user2`** cùng đăng nhập vào chạy chương trình **`grep`** đồng thời, thực tế, Hệ Điều Hành sẽ quản lý và nạp mã của chương trình **`grep`** vào hai vùng nhớ khác nhau và gọi mỗi phân vùng như vậy là tiến trình. Hình sau cho thấy cách phân chia chương trình **`grep`** thành hai tiến trình cho hai người khác nhau sử dụng



Trong hình này, **`user1`** chạy chương trình **`grep`** tìm chuỗi **`abc`** trong tập tin **`file1`**.

`$grep abc file1`

`user2` chạy chương trình **`grep`** và tìm chuỗi **`cde`** trong tập tin

file2.
$grep cde file2
Chúng ta cần ta cần nhớ là hai người dùng **user1** và **user2** có thể ở hai máy tính khác nhau đăng nhập vào máy chủ Linux và gọi **grep** chạy đồng thời. Hình trên là hiện trạng không gian bộ nhớ Hệ Điều Hành Linux khi chương trình **grep** phục vụ người dùng.
- Nếu dùng lệnh **ps**, hệ thống sẽ liệt kê cho chúng ta thông tin về các tiến trình mà Hệ Điều Hành đang kiểm soát, Ví dụ: **$ps –af**
Mỗi tiến trình được gán cho một định danh để nhận dạng gọi là **PID** (process identify). **PID** thường là số nguyên dương có giá trị từ 2-32768. Khi một tiến trình mới yêu cầu khởi động, Hệ Điều Hành sẽ chọn lấy một số (chưa bị tiến trình nào đang chạy chiếm giữ) trong khoảng số nguyên trên và cấp phát cho tiến trình mới. Khi tiến trình chấm dứt, hệ thống sẽ thu hồi số **PID** để cấp phát cho tiến trình khác trong lần sau. **PID** bắt đầu từ giá trị 2 bởi vì giá trị 1 được dành cho tiến trình đầu tiên gọi là **init**. Tiến trình **init** được và chạy ngay khi chúng ta khởi động Hệ Điều Hành. **init** là tiến trình quản lý và tạo ra mọi tiến trình con khác. Ở ví dụ trên, chúng ta thấy lệnh **ps –af** sẽ hiển thị 2 tiến trình **grep** chạy bởi **user1** và **user2** với số **PID** lần lượt là **101** và **102**.
- Mã lệnh thực thi của lệnh **grep** chứa trong tập tin chương trình nằm trên đĩa cứng được Hệ Điều Hành nạp vào vùng nhớ. Như chúng ta đã thấy ở lược đồ trên, mỗi tiến trình được Hệ Điều hành phân chia rõ ràng: vùng chứa mã lệnh (code) và vùng chứa dữ liệu (data). Mã lệnh thường là giống nhau và có thể sử dụng chung. Linux quản lý cho phép tiến trình của cùng một chương trình có thể sử dụng chung mã lệnh của nhau.
Thư viện cũng vậy. Trừ những thư viện đặc thù còn thì các thư viện chuẩn sẽ được Hệ Điều Hành cho phép chia sẻ và dùng chung bởi mọi tiến trình trong hệ thống. Bằng cách chia sẻ thư viện, kích thước chương trình giảm đi đáng kể. Mã lệnh của chương trình khi chạy trong hệ thống ở dạng tiến trình cũng sẽ đỡ tốn bộ nhớ hơn.
- Trừ mã lệnh và thư viện có thể chia sẻ, còn dữ liệu thì không thể chia sẻ bởi các tiến trình. Mỗi tiến trình sở hữu phân đoạn dữ liệu riêng. Ví dụ tiến trình **grep** do **user1** nắm giữ lưu giữ biến **s** có giá trị là '**abc**', trong khi **grep** do **user2** nắm giữ lại có biến **s** với giá trị là '**cde**'.
Mỗi tiến trình cũng được hệ thống dành riêng cho một bảng mô tả file (file description table). Bảng này chứa các số mô tả áp đặt cho các file đang được mở. Khi mỗi tiến trình khởi động, thường Hệ Điều Hành sẽ mở sẵn cho chúng ta 3 file : **stdin** (số mô tả **0**), **stdout** (số mô tả **1**), và **stderr** (số mô tả **2**). Các file này tượng trưng cho các thiết bị nhập, xuất, và thông báo lỗi. Chúng ta có thể mở thêm các file khác. Ví dụ **user1** mở file **file1**, và **user2** mở file **file2**. Hệ Điều Hành cấp phát số mô tả file cho mỗi tiến trình và lưu riêng chúng trong bảng mô tả file của tiến trình đó.
- Ngoài ra, mỗi tiến trình có riêng ngăn xếp stack để lưu biến cục bộ và các giá trị trả về sau lời gọi hàm. Tiến trình cũng được dành cho khoảng không gian riêng để lưu các biến môi trường. Chúng ta sẽ dùng lệnh **putenv** và **getenv** để đặt riêng biến môi trường cho tiến trình.

a) Bảng thông tin tiến trình
- Hệ Điều Hành lưu giữ một cấu trúc danh sách bên trong hệ thống gọi là bảng tiến trình (process table). Bảng tiến trình quản lý tất cả **PID** của hệ thống cùng với thông tin chi tiết về các tiến trình đang chạy. Ví dụng khi chúng ta gọi lệnh **ps**, Linux thường đọc thông tin trong bảng tiến trình này và hiển thị những lệnh hay tên tiến trình được gọi: thời gian chiếm giữ CPU của tiến trình, tên người sử dụng tiến trình, …

b) Xem thông tin của tiến trình
- Lệnh **ps** của Hệ Điều Hành dùng để hiển thị thông tin chi tiết về tiến trình. Tùy theo tham số, **ps** sẽ cho biết thông tin về tiến trình người dùng, tiến trình của hệ thống hoặc tất cả các tiến trình đang chạy. Ví dụ **ps** sẽ đưa ra chi tiết bằng tham số -**af**
- Trong các thông tin do **ps** trả về, **UID** là tên của người dùng đã gọi tiến trình, **PID** là số định danh mà hệ thống cấp cho tiến trình, **PPID** là số định danh của tiến trình cha (parent **PID**). Ở đây chúng ta sẽ gặp một số tiến trình có định danh **PPID** là 1, là định danh của tiến trình **init**, được gọi chạy khi hệ thống khởi động. Nếu chúng ta hủy tiến trình **init** thì Hệ Điều Hành sẽ chấm dứt phiên làm việc. **STIME** là thời điểm tiến trình được đưa vào sử dụng. **TIME** là thời gian chiếm dụng CPU của tiến trình. **CMD** là toàn bộ dòng lệnh khi tiến trình được triệu gọi. **TTY** là màn hình terminal ảo nơi gọi thực thi tiến trình. Như chúng ta đã biết, người dùng có thể đăng nhập vào hệ thống Linux từ rất nhiều terminal khác nhau để gọi tiến trình. Để liệt kê các tiến trình hệ thống, chúng ta sử dụng lệnh: **$ps –ax**

4. Tạo lập tiến trình

a) Gọi tiến trình mới bằng hàm system()
- Chúng ta có thể gọi một tiến trình khác bên trong một chương trình đang thực thi bằng hàm **system()**. Có nghĩa là chúng ta có thể tạo ra một tiến trình mới từ một tiến trình đang chạy. Hàm **system()** được khai báo như sau:

```
#include <stdlib.h>
int system( const char (cmdstr) )
```

Hàm này gọi chuỗi lệnh **cmdstr** thực thi và chờ lệnh chấm dứt mới quay về nơi gọi hàm. Nó tương đương

với việc bạn gọi shell thực thi lệnh của hệ thống: **$sh –c cmdstr**
system() sẽ trả về mã lỗi 127 nếu như không khởi động được shell để gọi lệnh **cmdstr**. Mã lỗi -1 nếu gặp các lỗi khác. Còn lại, mã trả về của **system()** là mã lỗi do **cmdstr** sau khi lệnh được gọi trả về.
Ví dụ sử dụng hàm **system()**, **system.c**

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
      printf( "Thuc thi lenh ps voi system\n" );
      system( "ps –ax" );
      system("mkdir daihoc");
      system("mkdir caodang");

      printf( "Thuc hien xong. \n" );
      exit( 0 );
}
```

Hàm **system()** của chúng ta được sử dụng để gọi lệnh "**ps –ax**" của Hệ Điều Hành.

b) Thay thế tiến trình hiện hành với các hàm exec

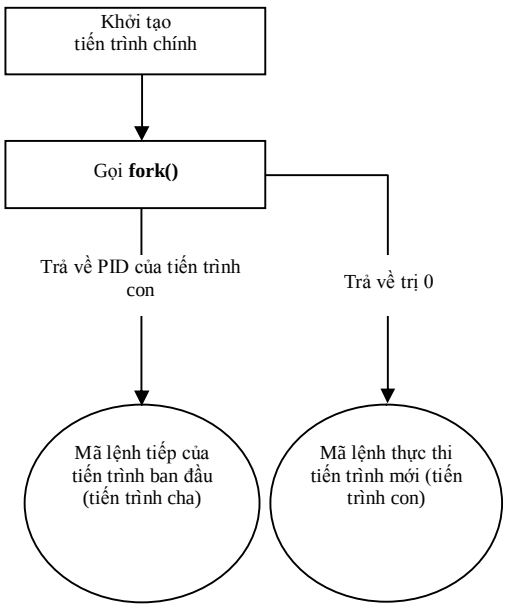
- Mỗi tiến trình được Hệ Điều Hành cấp cho 1 không gian nhớ tách biệt để tiến trình tự do hoạt động. Nếu tiến trình A của chúng ta triệu gọi một chương trình ngoài B (bằng hàm **system()** chẳng hạn), Hệ Điều Hành thường thực hiện các thao tác như: cấp phát không gian bộ nhớ cho tiến trình mới, điều chỉnh lại danh sách các tiến trình, nạp mã lệnh của chương trình B trên đĩa cứng và không gian nhớ vừa cấp phát cho tiến trình. Đưa tiến trình mới vào danh sách cần điều phối của Hệ Điều Hành. Những công việc này thường mất thời gian đáng kể và chiếm giữ thêm tài nguyên của hệ thống.

Nếu tiến trình A đang chạy và nếu chúng ta muốn tiến trình B khởi động chạy trong không gian bộ nhớ đã có sẵn của tiến trình A thì có thể sử dụng các hàm **exec** được cung cấp bởi Linux. Các hàm **exec** sẽ thay thế toàn bộ ảnh của tiến trình A (bao gồm mã lệnh, dữ liệu, bảng mô tả file) thành ảnh của một tiến trình B hoàn toàn khác. Chỉ có số định danh **PID** của tiến trình A là còn giữ lại. Tập hàm **exec** bao gồm các hàm sau:

```
#include <unistd.h>
extern char **environ;
int execl( const char *path, const char *arg, ... );
int execlp( const char *file, const char *arg, ... );
int execle( const char *path, const char *arg, ..., char *const envp[] );
int exect( const char *path, char *const argv[] );
int execv( const char *path, char *const argv[] );
int execvp( const char *file, char *const argv[] );
```

- Đa số các hàm này đều yêu cầu chúng ta chỉ đối số **path** hoặc **file** là đường dẫn đến tên chương trình cần thực thi trên đĩa. **arg** là các đối số cần truyền cho chương trình thực thi, những đối số này tương tự cách chúng ta gọi chương trình từ dòng lệnh.

c) Nhân bản tiến trình với hàm fork()



Cơ chế phân chia tiến trình của **fork()**

- Thay thế tiến trình đôi khi bất lợi với chúng ta. Đó là tiến trình mới chiếm giữ toàn bộ không gian của tiến trình cũ và chúng ta sẽ không có khả năng kiểm soát cũng như điều khiển tiếp tiến trình hiện hành của mình sau khi gọi hàm **exec** nữa. Cách đơn giản mà các chương trình Linux thường dùng đó là sử dụng hàm **fork()** để nhân bản hay tạo bản sao mới của tiến trình. **fork()** là một hàm khá đặc biệt, khi thực thi, nó sẽ trả về 2 giá trị khác nhau trong lần thực thi, so với hàm bình thường chỉ trả về 1 giá trị trong lần thực thi. Khai báo của hàm **fork()** như sau:

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork()
```

- Nếu thành công, **fork()** sẽ tách tiến trình hiện hành 2 tiến trình (dĩ nhiên Hệ Điều Hành phải cấp phát thêm không gian bộ nhớ để tiến trình mới hoạt động). Tiến trình ban đầu gọi là tiến trình cha (parent process) trong khi tiến trình mới gọi là tiến trình con (child process). Tiến trình con sẽ có một số định danh **PID** riêng biệt. ngoài ra, tiến trình con còn mang thêm một định danh **PPID** là số định danh **PID** của tiến trình cha.

- Sau khi tách tiến trình, mã lệnh thực thi ở cả hai tiến trình được sao chép là hoàn toàn giống nhau. Chỉ có một dấu hiệu để chúng ta có thể nhận dạng tiến trình cha và tiến trình con, đó là trị trả về của hàm **fork()**. Bên trong tiến trình con, hàm **fork()** sẽ trả về trị 0. Trong khi bên trong tiến trình cha, hàm **fork()** sẽ trả về trị số nguyên chỉ là **PID** của tiến trình con vừa tạo. Trường hợp không tách được tiến trình, **fork()** sẽ trả về trị -1. Kiểu **pid_t** được khai báo và định nghĩa trong **uinstd.h** là kiểu số nguyên (**int**).
- Đoạn mã điều khiển và sử dụng hàm **fork()** thường có dạng chuẩn sau:

```
pid_t new_pid;
new_pid = fork(); // tách tiến trình
switch (new_pid)
{
  case -1: printf( "Khong the tao tien trinh moi" ); break;
  case 0:  printf( "Day la tien trinh con" );
         // mã lệnh dành cho tiến trình con đặt ở đây
         break;
  default: printf( "Day la tien trinh cha" );
         // mã lệnh dành cho tiến trình cha đặt ở đây
         break;
}
```

d) Kiểm soát và đợi tiến trình con

- Khi **fork()** tách tiến trình chính thành hai tiến trình cha và con, trên thực tế cả hai tiến trình cha lẫn tiến trình con đều hoạt động độc lập. Đôi lúc tiến trình cha cần phải đợi tiến trình con thực hiện xong tác vụ thì mới tiếp tục thực thi. Ở ví dụ trên, khi thực thi, chúng ta sẽ thấy rằng tiến trình cha đã kết thúc mà tiến trình con vẫn in thông báo và cả tiến trình cha và tiến trình con đều tranh nhau gởi kết quả ra màn hình. Chúng ta không muốn điều này, chúng ta muốn rằng khi tiến trình cha kết thúc thì tiến trình con cũng hoàn tất thao tác của nó. Hơn nữa, chương trình con cần thực hiện xong tác vụ của nó thì mới đến chương trình cha. Để làm được việc này, chúng ta hãy sử dụng hàm **wait()**

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int &stat_loc);
```

Hàm **wait** khi được gọi sẽ yêu cầu tiến trình cha dừng lại chờ tiến trình con kết thúc trước khi thực hiện tiếp các lệnh điều khiển trong tiến trình cha. **wait()** làm cho sự liên hệ giữa tiến trình cha và tiến trình con trở nên tuần tự. Khi tiến trình con kết thúc, hàm sẽ trả về số **PID** tương ứng của tiến trình con. Nếu chúng ta truyền thêm đối số **stat_loc** khác **NULL** cho hàm thì **wait()** cũng sẽ trả về trạng thái mà tiến trình con kết thúc trong biến **stat_loc**. Chúng ta có thể sử dụng các macro khai báo sẵn trong **sys/wait.h** như sau:

WIFEXITED (stat_loc)	Trả về trị khác 0 nếu tiến trình con kết thúc bình thường.
WEXITSTATUS (stat_loc)	Nếu **WIFEXITED** trả về trị khác 0, macro này sẽ trả về mã lỗi của tiến trình con.
WIFSIGNALED (stat_loc)	Trả về trị khác 0 nếu tiến trình con kết thúc bởi một tín hiệu gửi đến.
WTERMSIG(stat_loc)	Nếu **WIFSIGNALED** khác 0, macro này sẽ cho biết số tín hiệu đã hủy tiến trình con.
WIFSTOPPED(stat_loc)	Trả về trị khác 0 nếu tiến trình con đã dừng.
WSTOPSIG(stat_loc)	Nếu **WIFSTOPPED** trả về trị khác 0, macro này trả về số hiệu của signal.

II. Thực Hành

Bài 1. Sử dụng hàm **system(), system_demo.c** tạo các tiến trình sau:

- Tạo thư mục **ThucHanh1** và **ThucHanh2**
- Tạo tập tin **Tho.c** trong thư mục **ThucHanh1** và ghi chuỗi **"troi hom nay that dep !"** vào tập tin vừa tạo (sử dụng lệnh **echo** để ghi chuỗi vào tập tin: **echo noi_dung_chuoi >ten_tap_tin**).
- Sao chép tập tin vừa tạo sang thư mục **ThucHanh2** và hiển thị lên màn hình.

Bài 2. Sử dụng hàm **execlp** để thay thế tiến trình hiện tại bằng tiến trình **ps –af** của Hệ Điều Hành.

```
#include <unistd.h>
#include <stdio.h>
int main()
{
  printf( "Thuc thi lenh ps voi execlp\n" );
  execlp( "ps", "ps", "-ax", 0 );
  printf( "Thuc hien xong! Nhung chung ta se khong thay duoc dong nay.\n"
);
  exit( 0 );
}
```

Bài 3. Tạo tập **tin fork_demo.c** sử dụng hàm **fork()** trong đó:

- In ra câu thông báo: "**Khong the tao tien trinh con !**" nếu hàm **fork()** trả về giá trị -1.
- Ngược lại:
  - In ra 5 lần câu thông báo: "**Day la tien trinh con !**" nếu mã trả về là 0.
  - In ra 3 lần câu thông báo: "**Day la tien trinh cha !**" nếu mã trả về là PID của tiến trình con.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main()
{
   pid_t pid;
   char * message;
   int n;
   pid = fork();
   switch ( pid )
   {
         case -1:
               printf( "Khong the tao tien trinh con !" );
               exit(1);
         case 0:
               message = "Day la tien trinh con !";
               n = 0;
               for ( ; n < 5; n++ ) {
                     printf( "%s", message );
                     sleep( 1 );
               }
               break;
         default:
               message = "Day la tien trinh cha !";
               n = 0;
               for ( ; n < 3; n++ ) {
                     printf( "%s", message );
                     sleep( 1 );
               }
               break;
   }
   exit( 0 );
}
```

Biên dịch và thực thi chương trình này, chúng ta sẽ thấy rằng cả 2 tiến trình hoạt động đồng thời và in ra kết quả đan xen nhau. Nếu muốn xem sự liên quan về **PID** và **PPID** của cả 2 tiến trình cha và con khi lệnh **fork()** phát sinh, chúng ta có thể thực hiện chương trình như sau:

$./fork_demo & ps – af

Bài 4. Sử dụng hàm **wait()** để chờ tiến trình con kết thúc sau khi gọi **fork()**, **wait_child.c**

```
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
#include <sys/types.h>
int main()
{
    pid_t pid;
    int child_status;
    int n;
    // nhân bản tiến trình, tạo bản sao mới
    pid = fork();
    switch ( pid ) {
         case -1: // fork không tạo được tiến trình mới
              printf("Khong the tao tien trinh moi");
              exit( 1 );
         case 0:  // fork thành công, chúng ta đang ở trong tiến trình con
              n = 0;
              for ( ; n < 5; n++ ) {
                   printf( "Tien trinh con" );
                   sleep( 1 );
              }
              exit( 0 );              // Mã lỗi trả về của tiến trình con
         default: //  fork thành công, chúng ta đang ở trong tiến trình cha
              printf("Tien trinh cha, cho tien trinh con hoan thanh.\n”);
              // Chờ tiến trình con kết thúc
              wait( &child_status );
              printf("Tien trinh cha – tien trinh con hoan thanh.\n");
    }
    return ( 0 );
}
```

Bài 5. Sử dụng hàm **wait()** để chờ tiến trình con kết thúc sau khi gọi **fork()**, **wait_child2.c,** kiểm tra mã lỗi trả về từ tiến trình con.

```
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
#include <sys/types.h>
int main()
{
    pid_t pid;
    int child_status;
    int n;
    // nhân bản tiến trình, tạo bản sao mới
    pid = fork();
    switch ( pid ) {
          case -1: // fork không tạo được tiến trình mới
                printf("Khong the tao tien trinh moi");
                exit( 1 );
          case 0:  // fork thành công, chúng ta đang ở trong tiến trình con
                n = 0;
                for ( ; n < 5; n++ ) {
                     printf( "Tien trinh con" );
                     sleep( 1 );
                }
                exit( 37 );          // Mã lỗi trả về của tiến trình con
          default: // fork thành công, chúng ta đang ở trong tiến trình cha
                n = 3;
                for ( ; n > 0; n-- ) {
                     printf( "Tien trinh cha" );
                     sleep( 1 );
                }
```

```
                // Chờ tiến trình con kết thúc
                wait( &child_status );
                // Kiểm tra và in mã lỗi trả về của tiến trình con
                printf( "Tien trinh con hoan thanh: PID = %d\n", pid );
                if ( WIFEXITED( child_status ))
                     printf( "Tien trinh con thoat ra voi ma %d\n",
                     WEXITSTATUS( child_status ) );
                else
                     printf( "Tien trinh con ket thuc binh thuong\n" );
                 break;
    }
    exit( 0 );
}
```