TINY SHELL

Phạm Văn Thông

Ngày 1 tháng 12 năm 2014

MỘT CHƯƠNG TRÌNH ĐANG THỰC THI ĐƯỢC GỌI LÀ MỘT TIẾN TRÌNH (PROCESS). Nếu bạn có hai cửa sổ *terminal* đang hiển thị trên màn hình, thì bạn đang thực thi cùng một chương trình *terminal* hai lần - bạn đang có hai tiến trình *terminal*. Mỗi cửa sổ terminal chạy một *shell*, mỗi *shell* đang chạy được gọi là một tiến trình khác. Khi bạn thực thi (*invoke*) một câu lệnh từ một *shell*, tương ứng có một chương tring trình được thực thi trong một tiến trình mới, tiến trình *shell* tiếp tục được thực hiện khi tiến trình kia hoàn thành.

Những lập trình viên chuyên nghiệp thường sử dụng đa tiến trình đồng thời trong một ứng dụng để giúp chương trình làm được nhiều hơn một việc trong một lúc, giúp tăng hiệu quả cho chương trình, và để sử dụng những chương trình đã sẵn có.

Phần lớn những *hàm thao tác tiến trình* được sử dụng trong bài viết này giống nhau trong những hệ thống UNIX khác. Phàn lớn được mô tả trong tệp tiêu đề <unistd.h>, kiểm tra man page cho mỗi hàm để chắc chắn điều đó.

1 Tiến trình

Có nhiều tiến trình được thực thi trong máy tính của bạn. Mỗi chương trình thực thi sử dụng một hoặc nhiều tiến trình. Hãy bắt đầu bằng việc nhìn vào những tiến trình đang có sẵn trên máy tính của bạn.

1.1 Định danh tiến trình (Process IDs)

Mỗi chương trình trong một hệ thống Linux được định danh bởi một đơn vị được gọi là *Process ID-PID*, đôi lúc chúng được nhắc tới như là *pid*. Định danh tiến trình (*Process IDs*) là một con số 16 bit, được gán bởi Linux cho một tiến trình đã được khởi tạo.

Mỗi tiến trình chỉ có duy nhất một tiến trình mẹ (*parent process*). Bạn có thể nghĩ về tiến trình trong một hệ thống Linux được tổ chức như là một cái cây, tiến trình *init* (tiến trình khởi tạo) là gốc. Mỗi *Parent Process ID-PPID* đơn giản là một định danh của tiến trình mẹ.

Khi nhắc đến PID trong một chương trình C/C++, ta thường sử dụng định nghĩa kiểu pid_t được định nghĩa trong <sys/types.h>. Một chương trình có thể nhận được PID của tiến trình đang thực thi bởi lời gọi hệ trống getpid(), và

nó có thể lấy được PID của tiến trình mẹ bằng lời gọi hệ thống (*system call*) getppid. Ví dụ chương trình sau sẽ in ra PID và PPID.

```
#include <stdio.h>
#include <unistd.h>
int main()
{
        printf("Process ID: %d\n",(int)getpid());
        printf("Parent Process ID: %d\n",(int)getppid());
        return 0;
}
```

Để ý rằng nếu bạn chạy chương trình này một vài lần thì sẽ có hiển thị PID khác nhau bởi vì mỗi lần gọi, chương trình được thực hiện trong một tiến trình mới. Dĩ nhiên, nếu bạn thực hiện chương trình nhiều lần trong cùng một shell thì PPID là giống nhau, bởi vì đó chính là PID của tiến trình shell.

1.2 Xem tiến trình đang hoạt động

Lệnh ps dùng để hiển thị những tiến trình đang hoạt động trên hệ thống của bạn. Lệnh ps của các phiên bản GNU/Linux sẽ có rất nhiều lựa chọn bởi vì nó cố gắng tương thích với những phiên bản của lệnh ps trên một vài hệ thống UNIX khác. Đây là những tùy chọn điều khiển những tiến trình được liệt kê ra và những thông tin tin mà chúng được hiển thị.

Mặc định, lệnh ps sẽ hiển thị những tiến trình được điều khiển bởi terminal hoặc cửa sổ terminal mà nơi lệnh ps được thực thi. Ví dụ:

```
%ps
PID    TTY    TIME     CMD
21552  pts/8  00:00:00 bash
21642  pts/8  00:00:00 ps
```

Lệnh ps này hiển thị hai tiến trình. Đầu tiên là bash, nó là shell đang chạy trên trên terminal này. Cái thứ hai chính là thông tin của chính lệnh ps đang được thực thi. Cột đầu tiên là nhãn được đặt cho PID (label), hiển thi PID cho mỗi tiến trình. Để có chi tiết hơn về những gì đang thực hiện trên hệ thống GNU/Linux của bạn, hãy chạy lệnh:

```
%ps -e -o pid,ppid,command
```

Tùy chọn -e sẽ chỉ cho ps hiển thị tất cả các tiến trình đang chạy trên hệ thống. Tùy chọn textsf-o pid,ppid,command sẽ giúp hiển thị những thông tin về mỗi tiến trình. Trong trường hợp này là PID,PPID, và lệnh đang được thực thi của mỗi tiến trình.

Đây là một số dòng đầu tiên và một số dòng cuối cùng được hiển thị từ lệnh ps trên máy của tôi. Bạn có thể không trông giông như thể này, điều đó tùy thuộc vào những gì đang chạy trên hệ thống của bạn.

```
% ps -e -o pid,ppid,command
```

```
PID  PPID  COMMAND
1         0    init [5]
2       1    [kflushd]
3       1    [kupdate]
...
21725  21693    xterm
21727  21725    bash
21728  21727    ps -e -o pid,ppid,command
```

Chú ý rằng PPID của lệnh ps là 21727, chính là PID của bash, là shell nơi mà ps được thực thi. Và PPID của bash lại là PID của chương trình xterm, nơi mà shell đang được thực thi.

1.3 Hủy tiến trinh

Bạn có thể hủy một tiến trình bằng lệnh kill, và tham số chính là PID của chương trình bạn muốn hủy. Lệnh kill hoạt động bằng cách gửi đến tiến trình một SIGTERM (termination signal-tín hiệu kết thúc). Lệnh này sẽ hủy bỏ tiến trình, trừ khi tiến trình đang thực thi đang được xử lý. Ngoài ra, bạn còn có thể sử dụng lệnh kill để gửi những tín hiệu khác đên tiến trình.

2 Khởi tạo tiến trình

Có hai kỹ thuật thường dùng để khởi tạo một tiến trình. Kỹ thuật đầu tiên thì đơn giản, nhưng không nên lạm dụng, bởi vì nó không hiệu quả và có những rủi ro về bảo mật.

2.1 Sử dụng hàm*system*

Hàm system trong thư viện chuẩn của C cho ta một cách đơn giản để thực hiện lệnh bên trong một chương trình. Thực sự là lệnh system sẽ tạo một chương trình con và chạy shell Bourne shell (/bin/sh) chuẩn và thực thi lệnh bên trong shell. Ví dụ chương trình sau sẽ thực hiện lệnh ls để hiển thị những nội dung bên trong thư mục gốc (root), giống như việt bạn thực thi lệnh ls -l / bên trong một shell.

```
#include <stdlib.h>
int main()
{
        int return_value;
        return_value=system("ls -l /");
        return return_value;
}
```

Lệnh system sẽ trả về trạng thái thực hiện của lệnh trong shell. Nếu shell không thể được thực thi, system sẽ trả về 127, nếu có lỗi khác, system sẽ trả về -1.

Bởi vì lệnh system sử dụng shell để thực thi lệnh của bạn, nó sẽ có những ưu điểm, hạn chế, và những lỗ hổng bảo mật của shell. Bạn không thể biết shell nào đang có săn, hoặc những thông tin về phiên bản của Bourne shell. Trong nhiều hệ thống UNIX, /bin/sh sẽ được liên kết tới một shell khác. Ví dụ như trong phần lớn những hệ thống Linux, /bin/sh sẽ liên kết tới shell bash. và những bản phân phối khác của GNU/Linux sẽ có những phiên bản bash khác nhau. Thực thi một chương trình với đặc quyền root với hàm system sẽ có thể cho nhiều kết quả khác nhau trên những hệ thống khác nhau. Vì những điều đó, kỹ thuật được yêu thích để khởi tạo một tiến trình là sử dụng hàm fork và exec.

2.2 Sử dụng fork và exec

Trong DOS và Windows API chứa một nhóm những hàm dùng để spawn (tạm dịch là phát sinh). Những hàm này sẽ lấy các tham số như là tên của chương trình và tạo một tiến trình mới cho chương trình. Trong Linux không có một hàm đơn lẻ nào có thể làm tất cả những một bước duy nhất. Thay vào đó, Linux cung cấp một hàm, fork, để khởi tạo một tiến trình con. Tiến trình con đó chính xác là bản copy của chính tiến trình mẹ của nó. Linux cung cấp một bộ các hàm khác, exec family (họ các hàm exec), dùng để tạo ra một tiến hoàn toàn riêng biệt để dừng việc thực hiện chương trình và thay thế cho việc trở thành 1 chương trình khác (Linux provides another set of functions, the exec family, that causes a particular process to cease being an instance of one program and to instead become an instance of another program). Để phát sinh (spawn) một tiến trình mới, đầu tiên bạn sử dụng fork để tạo một bản sao từ tiến trình hiện thời, sau đó sử dụng exec để biến một trong số chúng thành chương trình bạn muốn phát sinh.

Lời gọi fork Khi một chương trình gọi tới fork, một bản sao của tiến trình, được gọi là *tiến trình con*, được khởi tạo. Tiến trình mẹ tiếp tục việc thực hiện chương trình từ vị trí hàm fork được gọi. Tiến trình con cũng vậy, tiếp tục thực hiện một chương trình y hệt tại cùng vị trí đó.

Vậy làm thế nào để phân biệt sự khác nhau giữa hai tiến trình?

Thứ nhất, tiên trình con là một tiến trình mới và vì vậy nó có một định danh tiến trình mới (PID), khác với định danh của tiến trình mẹ của nó (PPID). Một cách để chương trình phân biệt đâu là tiến trình mẹ hoặc tiến trình con là sử dụng hàm getpid. Tuy nhiên hàm fork cung cấp một giá trị trả về khác nhau cho tiến trình mẹ và tiến trình con.Giá trị trả về cho tiến trình mẹ chính là PID của tiến trình con - một tiến trình bắt đầu hàm fork và hai tiến trình được trả về với giá trị trả về khác nhau. Giá trị trả về cho hàm mẹ chính là PID của tiến trình con vừa được tạo ra. Và giá trị trả về của tiến trình con là 0. Bởi vì không có tiến trình nào có PID là 0, nên điều đó sẽ dễ dàng cho cho chương trình biết nó đang chạy như là tiến trình mẹ hay tiến trình con.

Ví dụ sau đây sử dụng fork để nhân bản một tiến trình của một chương trình. Lưu ý răng khối lệnh của câu lệnh if chỉ được thực hiện trong tiến trình mẹ, trong khi câu lệnh trong else chỉ được thực thi ở tiến trình con.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main ()
{
    pid_t child_pid;
    printf("the main program process ID is %d\n", (int) getpid ());
    child_pid = fork ();
    if (child_pid != 0)
    {
        printf ("this is the parent process, with id %d\n", (int) getpid ());
        printf ("the child's process ID is %d\n", (int) child_pid);
    }
    else
        printf ("this is the child process, with id %d\n", (int) getpid ());
    return 0;
}
```

Sử dụng exec Một hàm exec thay thế một chương trình đang chạy bởi một chương trình khác. Khi một chương trình gọi đến hàm exec, tiến trình đó ngay lập tức dừng việc thực hiện chương trình này và bắt đầu thực hiện một chương trình mới tại điểm bắt đầu, giả sử như hàm exec không gặp phải một lỗi nào.

Sau đây là nhưng hàm thuộc họ exec:

- Những hàm chứa chữ cái p trong tên của chúng, execvp và execlp, chấp nhận một tên chương trình là tìm kiếm chương trình bởi tên đó trong đường dẫn hiện thời. Những hàm không chứa p sẽ phải đưa vào đường dẫn đầy đủ của chương trình cần được thực thi.

- Những hàm có chứa v trong tên của nó, execvp execv và execve, nhận danh sách tham sô cho chương trình mới nhu là một mảng con trỏ trỏ tới những xâu, kết thúc bởi NULL-terminated. Những hàm chứa kí tự l (execl, execlp và execle chấp nhận tham số là danh sách sử dụng kĩ thuật truyền tha số của ngôn ngữ lập trình C.

- Những hàm chứa kí tự e trong tên của nó (execve và execle) chấp nhận một tham số nữa, đó là một mảng của những biến môi trường. Tham số phải là một mảng của những con trỏ trỏ tới xâu kết thúc bởi NULL-terminated. Mỗi xâu có dạng "VARIABLE=value".

Vì hàm exec thay thế chương trình đang thực thi bởi một cái khác, nó không bao giờ *return* trừ khi xảy ra lỗi.

Danh sách tham số được truyền vào cho chương trình trương tự như những biến bạn truyền vào cho shell từ dòng lệnh. *They are available through the argc and argv parameters to main.* Nên nhớ rằng, khi một chương trình được gọi từ shell, phần tử đầu tiên của danh sách tham số argv[0] được đặt là tên của chương trình, phần tử tiếp theo argv[1] là tham số dòng lệnh đầu tiên, và tiếp tục như

vậy. Khi bạn thực hiện hàm exec trong chương trình, bạn cũng nên để tên hàm như là tham số đầu tiên trong danh sách các tham số.

Sử dụng fork kết hợp với exec Một cách để chạy một chương trình con trong một chương trình là sử dụng fork, sau đó là exec để thực hiện chương trình con. Điều này sẽ giúp chương trình mẹ tiếp tục thực hiện công việc trong khi chương trình được gọi lên được thay thể bởi một chương trình con khác.

Ví dụ sau đây liệt kê ra danh sách của thư mục gốc, sử dụng câu lệnh ls. Không giống như ví dụ trước, dường như câu lệnh ls được thực hiện trực tiếp bằng tham số dòng lệnh *-l* và */* hơn là thực hiện nó thông qua một shell.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

/* Spawn a child process running a new program. PROGRAM is the name
of the program to run; the path will be searched for this program.
ARG_LIST is a NULL-terminated list of character strings to be
passed as the program's argument list. Returns the process ID of
the spawned process. */
int spawn (char* program, char** arg_list)
{
    pid_t child_pid;
    /* Duplicate this process. */
    child_pid = fork ();
    if (child_pid != 0)
        /* This is the parent process. */
        return child_pid;
    else
    {
        /* Now execute PROGRAM, searching for it in the path. */
        execvp (program, arg_list);
        /* The execvp function returns only if an error occurs. */
        fprintf (stderr, ‘‘an error occurred in execvp\n’’);
        abort ();
    }
}
int main ()
{
    /* The argument list to pass to the ‘‘ls’’ command. */
    char* arg_list[] =
    {
          ‘‘ls’’,
          /* argv[0], the name of the program. */
          ‘‘-l’’,
```

```
        ‘‘/’’,
        NULL
        /* The argument list must end with a NULL.*/
    };
    /* Spawn a child process running the ‘‘ls’’ command.
    returned child process ID. */
    spawn (‘‘ls’’, arg_list);
    printf (‘‘done with main program\n’’);
    return 0;
}
```

2.3 Lập lịch tiến trình

Linux lập lịch tiến trình mẹ và tiến trình con độc lập với nhau. Không có sự đảm bảo nào cho việc tiến trình nào sẽ chạy trước, hoặc nó sẽ chạy bao lâu trước khi Linux ngắt (interrupts) nó và để cho tiến trình khác (hoặc một số tiến trình khác trên hệ thống) được thực thi. Trong trường hợp này, một phần,toàn bộ hoặc cũng có thể là không có phần nào của lệnh ls có thể chạy trong tiến trình con trước khi tiến trình mẹ hoàn thành. Linux chỉ đảm bảo rằng mỗi tiến trình đều được thực hiện - không có tiến trình nào bị đói tài nguyên cho viếc thực thi.

Bạn có thể chỉ ra tiến trình nào ít quan trọng hơn - và nên ít được ưu tiên hơn bằng cách gán cho nó giá trị *niceness* cao hơn. Theo mặc định, mỗi tiến trình đều có giá trị niceness là 0. Giá trị niceness cao hơn có nghĩa là tiến trình sẽ ít được ưu tiên thực thi hơn. Để chạy một chương trình với chỉ số niceness, sử dụng lệnh nice cùng với giá trị niceness trong tùy chọn -n. Ví dụ bạn có một lệnh như sau: "sort input.txt > output.txt" diễn ra trong một thời gian khá dài. Bằng cách giảm độ ưu tiên, nó sẽ không làm chậm hệ thống của bạn quá nhiều:

```
% nice -n 10 sort input.txt > output.txt
```

Ban có thể sử dụng lệnh renice để thay đổi chỉ số niceness của một tiến trình đang thực thi từ dòng lệnh. Để thay đôi niceness của một tiến trình trong chương trình, bạn sử dụng hàm nice. Tham số của nó là một gia số, nó sẽ được thêm vào giá trị niceness của tiến trình gọi nó. Chú ý rằng một giá trị dương sẽ tăng chỉ số niceness của tiến trình và giảm độ ưu tiên thực hiện của tiến trình.

Lưu ý rằng chỉ có tiến trình được thực thực thi với quyền root mới có thể có giá trị niceness âm, hoặc thay đổi giá trị niceness của một tiến trình đang thực thi. Điều đó có nghĩa rằng bạn có thể thiết lập một giá trị âm cho lệnh nice và renice chỉ khi bạn đăng nhập như là *root*, và chỉ có tiến trình thực thi với quyền root có thể nhận giá trị niceness âm. Điều đó ngăn cản những người dùng thưởng khỏi việc chiếm quyền thực thi tiến trình của người dùng khác trong hệ thống.

3 Tín hiệu (singals)

Signals (tín hiệu) là kỹ thuật giao tiếp và điều khiển giữa các tiến trình trong Linux. Chủ đề về tín hiệu là một chủ để rộng lớn, ở đây chúng ta chỉ nói về một số tín hiệu quan trọng và kĩ thuật sử dụng chúng để điều khiển hoạt động giữa các tiến trình.

Mội tín hiệu là một lời nhắn đặc biệt được gửi cho một tiến trình. Những tín hiệu thì không đồng bộ; khi một tiến trình nhận một tín hiệu, nó xử lý tín hiệu ngay tức thì mà không thực thi tiếp những câu lệnh một cách bình thường. Có một hàng tá những tín hiệu khác nhau, mỗi loại có một ý nghĩa riêng. Một loại tín hiệu được đặc tả bởi một số con số (singal number), nhưng trong chương trình, bạn thường sử dụng tên của nó. Trong Linux, chúng được định nghĩa ở /usr/include/bits/signum.h. (Bạn không nên include file header này trực tiếp vào chương trình, thay vào đó, hãy sử dụng <singnals.h>).

Khi một tiến trình nhận một tín hiệu, nó có thể làm một hoặc một vài việc, tùy thuộc vào sự sắp đặt của tín hiệu (signal's disposition). Với mỗi loại tín hiệu, có một *disposition* mặc định, cái mà sẽ quyết định việc gì sẽ xảy ra đối với tiến trình nếu chương trình không chỉ rõ một vài cách xử lý khác. Với phần lớn các loại tín hiệu, một chương trình có thể chỉ rõ một vài cách xử lý khác - hoặc là bỏ qua tín hiệu, hoặc gọi một hàm đặc biệt xử lý tín hiệu (special singal-handler function) để trả lời tín hiệu đó. Nếu một hàm xử lý tín hiệu được sử dụng, việc thực hiện của chương trình hiện tại sẽ được tạm dừng cho đến khi việc xử lý tín hiệu hoàn tất.

Hệ thống Linux tới tiến trình những tín hiệu phản hồi để đặc trưng cho những điều kiện. Cụ thể là: SIGBUS (bus error), SIGSEGV (segmentation violation), và SIGFPE (floating point exception) có thể được gửi tới một tiến trình mà nó đang cố gắng thực hiện những thao tác không hợp lệ.. Mặc định, những tín hiệu này sẽ hảy tiến trình và produce a core file.

Một tiến trình có thể cũng gửi tín hiệu tới những tiến trình khác. Kĩ thuật này thường dùng để kết thúc một tiến trình khác bằng cách gửi tới nó tín hiệu SIGTERM hoặc SIGKILL. Nó cũng thường được sử dụng để gửi một lệnh tới một chương trình đang thực thi. Hai tín hiệu do người dùng tự định nghĩa được sử dụng được thực hiện việc này: SIGUSR1 và SIGUSR2. Tín hiệu SIGUP đôi khi cũng được sử dụng tốt cho việc này, nó thường dùng để đánh thức một chương trình đang ngừng hoạt động hoặc bảo chương trình đọc lại file cấu hình của nó.

Hàm sigaction có thể được sử dụng để cài đặt một *sigal disposition*. Tham số đầu tiên là số tín hiệu, hai tham số tiếp theo là nhũng con trỏ tới cấu trúc sigaction, con trỏ đầu tiên chứa điều kiện disposition cho số tín hiệu, còn con trỏ thứ hai là disposition cũ (previous disposition). Trường quan trọng nhất trong cấu trúc sigaction là sa_handler. Nó có thể nhận một trong ba giá trị:

- SIG_DFL chỉ rõ disposition mặc định cho tín hiệu.
- SIG_IGN chỉ rõ rằng tín hiệu nên được bỏ qua.
- Một con trỏ tới một hàm xử lý tín hiệu. Hàm này nên nhận một tham số là số tín hiệu, và trả về void.

Bới vì tín hiệu là không đồng bộ, chương trình chính có thể rất không ổn định khi mà một tín hiệu được xử lý , trong khi hàm xử lý tín hiệu đang thực thi. Vì vậy, bạn nên tránh xa việc thực hiện bất cứ thao tác vào ra nào, hoặc gọi tới phần lớn những thư viện hoặc hàm của hệ thống bên trong những hàm xử lý tín hiệu.

Một hàm xử lý tín hiệu nên làm tối thiểu những việc cần để có phản hồi tới tín hiệu, và trả quyền điều khiển về cho chương trình chính (hoặc hủy bỏ chương trình chính). Trong phần lớn các trường hợp, điều đó bao gồm việc ghi lại sự việc có một tín hiệu đã xảy xa. Chương trình chính sau đó sẽ kiểm tra một cách định kỳ có hay không một tín hiệu đã xảy ra và phản ứng lại cho phù hợp.

Môt hàm xử lý tín hiệu có thể bị ngắt (interrupt) bởi sự vận chuyển của tín hiệu khác. Việc này nghe có vẻ hiếm khi xảy ra, và nếu nó xảy ra, sẽ rất khó để chẩn đoán và gỡ rối (debug). Vì vậy, bạn nên phải rất cẩn thận về việc chương trình của bạn làm gì trong hàm xử lý tín hiệu (signal handler).

Thậm chí việc gán một gái trị tới một biến toàn cục có thể trở nên nguy hiểm bởi vì phép gán có thể trở thành một hoặc nhiều chỉ thị máy, và một tín hiệu có thể xảy ra giữa những chỉ thị kia, và khiến cho phép gán có kết quả không như ý định của người lập trình. Nếu bạn sử dụng một biến toàn cục để làm cờ cho một tín hiệu trong hàm xử lý tín hiệu, nó nên có kiểu đặc biệt là sig_atomic_t. Linux bảo đảm rằng phép gán với biến có kiểu dữ liệu này sẽ được thực hiện trong một chỉ thì, và vì vậy sẽ không bị ngắt giữa chùng. Trong Linux, sig_atomic_t là một số nguyên bình thường. Nếu bạn muốn biết một chương trình có thể chạy trên mõi hệ thống UNIX chuẩn, hãy sử dụng kiểu sig_atomic_t cho những biến toàn cục này.

Sau đây là một ví dụ cho chương trình sử dụng hàm xử lý tín hiệu để đếm số lần mà chương trình nhận tín hiệu SIGUSR1, một trong những tín hiệu dành này được dành riêng cho ứng dụng để sử dụng.

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/types>
#include <unistd.h>

sig_atomic_t sigusr1_count=0;

void handler (int signal_number)
{
    ++sigusr1_count;
}

int main()
{
   struct sigaction sa;
   memset(&sa,0,sizeof(sa));
```

```
    sa.sa_handler = &handler;
    sigaction(SIGUSR1, &sa, NULL);

    /* Do some lengthy stuff here */
    /*.................................*/

    printf("SIGUSR1 was raised %d times \n",sigusr1_count);
    return 0;
}
```

4 Hủy bỏ tiến trình

Bình thường, một tiến trình bị hủy bỏ bởi một trong hai cách, hoặc chương trình đang thực hiện gọi tới hàm exit, hoặc hàm main chính return.

Một tiến trình cũng có thể bị hủy bỏ một cách bất thường, trong sự phản hồi của một tín hiệu. Cụ thể, tín hiệu SIGBUS,SIGSEGV, và SIGFPE là nguyên nhân khiến cho những tiến trình bị hủy bỏ. Một số tín hiệu khác cũng được dùng để hủy bỏ tiến trình. Tín hiệu SIGINT được gửi tới một tiến trình khi người sử dụng cố gắng kết thúc nó bằng các sử dụng tổ hợp phím CTRL+C trong *terminal*. Tín hiệu SIGTERM được gửi tới bởi lệnh kill. Cách thức xử lý cho cả hai trường hợp này đều là hủy bỏ tiến trình. Bằng cách gọi hàm abort, một tiến trình tự gửi tín hiệu SIGART, tín hiệu này sẽ hủy bỏ tiến trình. Tín hiệu hủy bỏ mạnh nhất là SIGKILL, nó sẽ hủy bỏ tức thì và không thể bị block hay điều khiển bởi một chương trình nào.

Một vài trong số những tín hiệu này có thể sử dụng lệnh kill cùng bằng cách chỉ rõ những cờ lệnh mở rộng (extra commandline-flag), ví dụ, để kết thúc một tiến trình với PID (Process ID) là pid, sử dụng câu lệnh:

%kill -KILL pid

Để gửi một tín hiệu từ một chương trình, sử dụng lệnh kill. Tham số đầu tiên là PID của tiến trình, tham số tiếp theo là số tín hiệu, mặc định là SIGTERM. Ví dụ, nếu pid chứa PID của tiến trình con, bạn có thể sử dụng hàm kill để hủy bỏ tiến trình con từ tiến trình mẹ băng cách sử dụng câu lệnh như thế này:

kill (child_pid, SIGTERM);

Bạn cần *include* <sys/types.h> và <signal.h> để sử dụng hàm kill.

Theo quy ước, exit code dùng để chỉ ra chương trình có được thực hiện đúng hay không. Exit code là 0 chỉ ra rằng chương trình đã thực hiện đúng, trong khi đó, exit code khác không sẽ cho thấy đã xảy ra lỗi nào đó trong quá trình thực hiện. Trong những trường hợp gần đây (latter case), giá trị trả về của chương trình có thể cho ta một vài chỉ dẫn về bản chất của lỗi đã xảy ra. Sử dụng những quy ước về lỗi trong chương trình của bạn là một ý tưởng tốt bởi những thành phần khác của hệ thống GNU/Linux đều tuân theo những quy ước này. Ví dụ, shell tuân theo quy ước khi bạn kết nối nhiều chương trình trong một câu lệnh bằng các toán tử logic như && hay ||. Vì vậy bạn nên trả về 0 trong chương trình chính, trừ khi có lỗi xảy ra.

Trong phần lớn các shell, chúng ta có thể nhận được exit code của chương

trình được thực thi gần đây nhất bằng cách dùng biến đặc biệt: $?. Sau đây là một ví dụ về lệnh ls và hiển thị exit code của nó. Trong trường hợp đầu, ls thực hiến chính xác, và trả về exit code là 0, trường hợp thứ hai xảy ra lỗi (bởi vì tên file/directory được chỉ ra không tồn tại), và vì vậy không trả về giá trị 0.

```
% ls /
bin    code    etc    lib          misc    nfs    proc    sbin   usr
boot   dev     home   lost+found   mnt     opt    root    tmp    var
%echo $?
0
%ls bogusfile
ls: bogusfile: No such file or directory
%echo $?
1
```

Chú ý rằng ngay cả khi tham số của hàm exit là số nguyên và giá trị trả về trong hàm main là số nguyên, Linux không bảo toàn đầy đủ 32 bit của giá trị trả về. Thực tế, bạn nên sử dụng exit code nằm trong khoảng từ 0 đến 127. Exit code có giá trị lớn hơn 128 sẽ có một ý nghĩa đặc biệt - khi tiến trình bị hủy bỏ bởi một tín hiệu, exit code của nó sẽ là 128 cộng với số tín hiệu.

4.1 Chờ đợi tiến trình

Nếu bạn chạy ví dụ về fork và exec, vạn có thể nhận thấy rằng output từ chương trình ls có thể xuất hiện sau khi "main program" đã hoàn thành. Đó là bởi vì tiến trình con được lập lịch độc lập với tiến trình mẹ. Bởi vì Linux là một hệ điều hành đa nhiệm, cả hai tiến trình xuất hiệt để thực thi xảy ra đồng thời, và bạn không thể nói trước được rằng chương trình ls sẽ có một cơ hội chạy trước hoặc sau tiến khi tiến trình mẹ được thực thi.

Trong một số hoàn cảnh, chúng ta muốn cho tiến trình mẹ phải chờ cho đến khi một hoặc nhiều tiến trình con đều đã được hoàn thành. Ta có thẻ thực hiện những việc đó bằng nhóm các lời gọi hệ thống - wait. Những hàm này chập nhận việc chờ đợi một tiến trình kết thúc việc thực thi, và kích hoạt tiến trình mẹ để nhận các thông tin về việc kết thúc của các tiến trình con của nó. Có bốn system call khác nhau trong họ các ham wait, bạn có thẻ chọn ít hoặc nhiều thông tin về tiến trình vừa kết thúc, và lựa chọn việc bạn có quan tâm đến tiến trình con nào kết thúc hay không.

4.2 wait system call

Sử dụng hàm wait sẽ block tiến trình đang gọi cho đến khi tiến trình con của nó thực hiện xong (hoặc xảy ra lỗi). Nó trả về một mã là là một con trỏ nguyên, trỏ đến nơi mà bạn có thẻ trích ra thông tin về việc tiến trình con đó kết thúc như thế nào. Sử dụng macro WEXITSTATUS để xem thông tin về exit code của tiến trình con. Bạn có thể sử dụng macro WIFEXITED để xác địnhtừ một tiến trình con có kết thúc bình thường (bởi hàm exit hoặc hàm return từ main) hay kết thúc bởi một tín hiệu.

Đây là một hàm main tù ví dụ fork và exec. Lần này, tiến trình mẹ gọi wait để chờ cho đến khi tiến trình con kết thúc, tức là khi lệnh ls kết thúc quá trình thực hiện.

```
int main()
{
    int child_status;
    /* The argument list to pass to the "ls" command. */
    char *arg_list[]={"ls","-l","/",NULL};
    /*spawn a child process running "ls" command. Ignore the
    returned child process ID. */
    spawn("ls",arg_list);
    /*wait for the child process to complete */
    wait(&child_status);
    if (WIFEXITED (child_status))
        printf("The child process exited normally, with exit code %d\n",
            WEXITSTATUS( child_status));
    else
        printf("The child process exit abnormally\n");
        return 0;
}
```

Có một vài lời gọi hệ thống tương tự trong Linux, những hàm này linh động hơn hoặc cung cấp nhiều thông tin hơn về sự thoạt ra của tiến trình con. Hàm waitpid có thể sử dụng để chờ một tiến trình còn cụ thể nào đó kết thúc. Hàm wait3 trả về trạng thái sử dụng CPU của tiến trình con, và hàm wait4 cho phép bạn chỉ rõ những điều kiện mở rộng về những tiến trình mà bạn đang đợi.

4.3 Tiến trình ma (Zombie Processes)

Nếu một tiến trình con kết thúc trong khi tiến trình mẹ đang gọi một hàm wait, tiến trình con sẽ biến mất mà trạng thái kết thúc của nó được gửi tới cho tiến trình mẹ thông qua hàm wait. Nhưng điều gì sẽ xảy ra khi tiến trình con kết thúc và tiến trình mẹ không sử dụng hàm wait?. Có phải nó chỉ đơn giản là biến mất. Không, bởi vì sau đó, thông tin về sự kết thúc của nó - chẳng hạn như là nó có kết thúc bình thường không, hay nếu có thì trạng thái kết thúc (exit status) của nó là gì - nó có bị mất không. Thay vào đó, khi một tiến trình con kết thúc, nó trở thành zombie process (tiến trình ma).

Một *zombie process* là một tiến trình đã kết thúc nhưng chưa được dọn dẹp. Trách nhiệm của tiến trình mẹ là phải dọn dẹp những tiến trình con zombie của nó. Những hàm wait củng đảm nhận việc này, vì vạy, bạn không cẩn phải kiểm tra xem (track) những tiến trình con của mình xem nó vẫn còn đang thức thi trước khi chờ đợi nó. Giả sử, một chương trình sử dụng lời gọi fork tạo ra một tiến trình con, sau đó làm một số thao tác khác, rồi tiến hành chờ. Nếu tiến trình con không kết thúc tại thời điểm này, tiến trình mẹ sẽ bị block tại hàm wait cho đến khi tiến trình con kết thúc. Nếu tiến trình con kết thúc trước khi

tiến trình mẹ gọi hàm wait, nó sẽ trở thành 1 zombie process. Khi mà tiến trình mẹ gọi hàm wait, trạng thái thoát ra của tiến trình con được giải mã, tiến trình con sẽ bị xóa, và hàm wait return tức thì.

Chuyện gì xảy ra nếu tiến trình mẹ không dọn dẹp tiến trình con? Chúng rẽ ở đâu đó trong hệ thống, như những zombie process. Hãy xem ví dụ sau:

Chương trình này gọi lên một tiến trình con bằng hàm fork, sau đó sẽ sleep trong vòng 1 phút và không dọn dẹp tiếp trình con của nó.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
    pid_t childPID;
    childPID = fork();
    if (childPID)
    {
        /*This is parent process*/
        sleep (60);
    }
    else
    {
        /*This is child process. Exit immediately */
        exit (0);
    }
    return 0;
}
```

Hãy thử biên dịch nó thành một file thực thi tên là make-zombie. Chạy nó, và trong khi nó vẫn đang chạy, hãy hiển thị những tiến trình đang chạy trong hệ thống của bạn bằng câu lệnh:

```
% ps -e -o pid,pid,stat,cmd
```

Danh sách này hiển thị PID, PPID, process status, và lệnh tương ứng trên cùng một dòng, theo dõi nó, bạn sẽ thấy ngoài tiến trình mẹ là make-zoombie, còn có một tiến trình make-zombie nữa cùng được liệt kê. Đó là tiến trình con của nó. Chú ý rằng PPID của tiến trình make-zombie là PID của tiến trình make-zombie còn lại. Tiến trình con được đánh dấu là <defunt> và trạng thái của nó là Z - dành cho zombie.

Điều gì xảy ra khi hàm main của chương trình make-zombie kết thúc khi mà tiến trình mẹ của nó kết thúc mà không sử dụng lời gọi wait? Phải chăng zombie process vẫn tồn tại ở đâu đó? Không, hãy thử chạy lại lệnh ps một lần nữa, và chú ý rằng cả hai tiến trình make-zombie đều đã biên mất. Khi một chương trình kết thúc, những tiến trình con của nó được kế thừa (inherited) bởi

một tiến trình đặc biệt, đó chính là chương trình init, cái mà luôn được chạy với process ID là 1 (đó chính là tiến trình Linux thực hiện đầu tiên khi khởi động). Tiến trình init sẽ tự động dọn dẹp bất cứ zombie process nào nó thừa hưởng.

4.4 Dọn dẹp tiến trình con không đồn bộ

Nếu bạn đang sử dụng một tiến trình con đơn giản chỉ để thực hiện một chương trình khác (bằng hàm exec), sẽ thật tốt nếu bạn gọi hàm wait trực tiếp trong tiến trình mẹ, nó sẽ bị block cho đến khi tiến trình con hoàn thành. Nhưng thông thường, bạn sẽ muốn tiến trình mẹ sẽ tiếp tục chạy như là một hoặc nhiều tiến trình con đồng bộ với nhau. Làm thế nào bạn có thể chắc chắng răng bạn đã dọn dẹp những tiến trình con đã hoàn thành, không để lại những zombie process, cái mà sẽ tiêu phí tài nguyên của bạn.

Một cách giải quyết cho tiến trình mẹ là sử dụng hàm wait3 và wait4 định kì, để dọn dẹp những zombie process. Gọi hàm wait vào múc đích này sẽ không làm việc tốt, bởi vì nếu không có tiến trình con nào kết thúc, nó sẽ block cho đến khi một tiến trình con kết thúc. Tuy nhiên, hàm wait3 và wait4 có thêm một tham số phụ nữa là cờ hiệu. Nếu bạn đặt vào cờ hiệu (flag) giá trịnh là WNOHANG, hàm sẽ chạy trong trạng thái *nonblocking mode* - nó sẽ dọn dẹp những tiến trình con đã kết thúc (nếu chúng tồn tại), hoặc trả về nếu không có tiến trình con nào vừa kết thúc. Giá trị trả về của lời gọi này là PID của tiến trình con đã kết thúc, hoặc bằng 0 (nếu không có tiến trình nào phải dọn dẹp).

Một giải pháp tốt hơn là là báo cho tiến trình mẹ biết khi một tiến trình con kết thúc. Khi một tiến trình con kết thúc, Linux gửi một tín hiệu SIGCHLD tới tiến trình mẹ. mặc định, tín hiệu này sẽ không làm gì cả, which is why you might not have noticed it before.

Vì vậy, cách dễ dàng hơn để dọn dẹp tiến trình con là điều khiển tín hiệu SIGCHLD. Dĩ nhiên, khi dọn dẹp tiến trình con, điều quan trọng là bạn phải lưu thông tin về trạng thái kết thúc của nó nếu những thông tin đó là cần thiết, bởi vì một khi tiến trình vừa được dọn dẹp bằng hàm wait, thông tin của nó sẽ không còn nữa.

Chương trình sau đây ví dụ về cách sử dụng SIGCHLD để dọn dẹp những tiến trình con.

```
#include <signal.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>

sig_atomic_t child_exit_status;

void clean_up_child_process (int signal_number)
{
    /* Clean up the child process. */
    int status;
    wait (&status);
```

```
    /* Store its exit status in a global variable.*/
    child_exit_status = status;
}

int main ()
{
    /* Handle SIGCHLD by calling clean_up_child_process. */
    struct sigaction sigchld_action;
    memset (&sigchld_action, 0, sizeof (sigchld_action));
    sigchld_action.sa_handler = &clean_up_child_process;
    sigaction (SIGCHLD, &sigchld_action, NULL);
    /* Now do things, including forking a child process.*/
    /* ... */
    return 0;
}
```

Chú ý cách sử dụng tín hiệu lưu thông tin vào một biến toàn cục, và từ chương trình chính có thể truy cập nó. Bởi vị biến này được gán giá trị trong một signal_handler, kiểu giá trị của nó là sig_atomic_t