HỆ ĐIỀU HÀNH
TUẦN 6
QUẢN LÝ TIẾN TRÌNH

Tóm tắt lý thuyết

Khi bạn bắt đầu một tiến trình (chạy một lệnh), có 2 cách để bạn chạy nó:

- Foreground Process
- Background Process

Tiến trình Foreground trong Unix/Linux

Theo mặc định, mọi tiến trình mà bạn bắt đầu chạy là Foreground Process. Nó nhận input từ bàn phím và gửi output tới màn hình.

Bạn có thể quan sát điều này xảy ra với lệnh **`ls`**. Nếu bạn muốn liệt kê tất cả các file trong thư mục hiện tại, bạn có thể sử dụng lệnh sau:

```
~$ ls
```

Tiến trình chạy trong Foreground, kết quả của nó được hướng trực tiếp trên màn hình và nếu lệnh **`ls`** muốn bất kỳ đầu vào nào, nó đợi từ bàn phím.

Trong khi một chương trình đang chạy trong Foreground và cần một khoảng thời gian dài, chúng ta *không thể* chạy bất kỳ lệnh khác (bắt đầu một tiến trình khác) bởi vì dòng nhắc không có sẵn tới khi chương trình đang chạy kết thúc tiến trình và thoát ra.

Tiến trình Background trong Unix/Linux

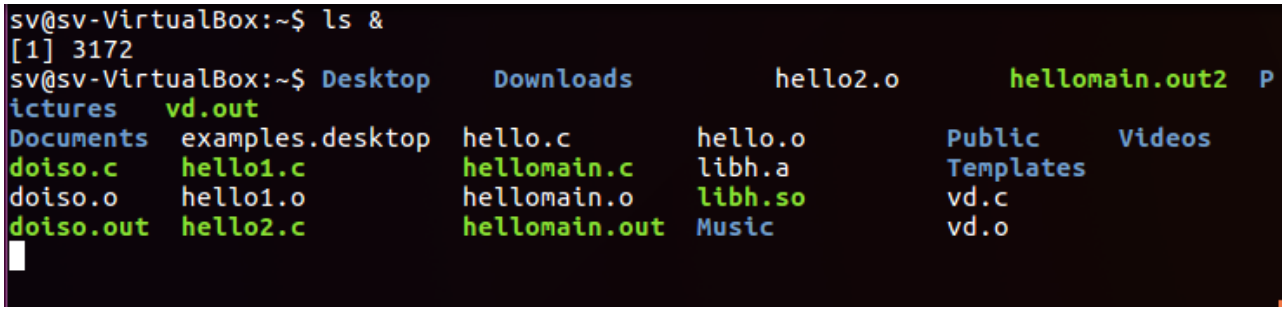
Background Process chạy mà không được kết nối với bàn phím của bạn. Nếu tiến trình Background yêu cầu bất cứ đầu vào từ bàn phím, nó đợi.

Lợi thế của chạy một chương trình trong Background là bạn có thể chạy các lệnh khác; bạn không phải đợi tới khi nó kết thúc để bắt đầu một tiến trình mới!

Cách đơn giản nhất để bắt đầu một tiến trình Background là thêm dấu và (**&**) tại phần cuối của lệnh.

```
~$ ls &
```

Tại đây, nếu lệnh **`ls`** muốn bất kỳ đầu vào nào (mà nó không), nó tiến vào trạng thái dừng tới khi bạn di chuyển nó vào trong Foreground và cung cấp cho nó dữ liệu từ bàn phím.



Dòng đầu tiên chứa các thông tin về Background Process - số công việc (job number) và Process ID. Bạn cần biết về Job number để thao tác nó giữa Background và Foreground. Nếu bạn nhấn phím **Enter** bây giờ, bạn nhìn thấy như sau:

```
[1]+  Done                    ls --color=auto
sv@sv-VirtualBox:~$
```

Dòng đầu tiên nói cho bạn rằng lệnh **`ls`** trong Background Process đã hoàn thành một cách thành công. Dòng thứ hai là một dòng nhắc cho một lệnh khác.

Liệt kê các tiến trình đang chạy trong Unix/Linux

Nó là dễ dàng để quan sát các tiến trình của bạn bằng cách chạy lệnh **`ps`** (viết tắt của process status) như sau:

`ps [options]`

Các option:

-a: hiển thị các tiến trình của user được liên kết

-e: hiển thị tất cả tiến trình

-f: hiển thị PID của tiến trình cha và thời điểm bắt đầu

-l: tương tự -f

Ví dụ: **`~$ ps`**

```
sv@sv-VirtualBox:~$ ps
  PID TTY          TIME CMD
 2128 pts/1    00:00:00 bash
 3192 pts/1    00:00:00 ps
sv@sv-VirtualBox:~$ ps -f
UID          PID  PPID  C STIME TTY          TIME CMD
sv          2128  2091  0 21:26 pts/1    00:00:00 /bin/bash
sv          3196  2128  0 21:56 pts/1    00:00:00 ps -f
sv@sv-VirtualBox:~$ ps -af
UID          PID  PPID  C STIME TTY          TIME CMD
sv          3197  2128  0 21:56 pts/1    00:00:00 ps -af
```

Mỗi một tiến trình Unix có hai ID được gán cho nó: Process ID (**pid**) và Parent Process ID (**ppid**). Mỗi tiến trình trong hệ thống có một Parent Process (gốc).

Tiến trình Zombie và Orphan

Do thời gian tồn tại của process cha và process con là khác nhau, điều đó gây nên 2 vấn đề:

Nếu *process cha kết thúc trước* thì ai sẽ là process cha của các process con? Vấn đề này kernel giải quyết bằng cách cho process init (process có PID = 1) làm cha của các process **orphan** này.

Điều gì xảy ra nếu *process con kết thúc trước khi process cha gọi hàm wait()*? Kernel giải quyết vấn đề này bằng cách đưa process con về trạng thái **zombie**. Điều này có nghĩa là hầu hết tài nguyên của process con đều đã được giải phóng, ngoại trừ một bảng ghi thông tin process như ID, trạng thái kết thúc, thông số sử dụng tài nguyên của process con. Sau khi

process cha thực hiện wait() thì kernel sẽ xóa zombie đi. Mặt khác nếu process cha kết thúc mà không gọi wait(), init process sẽ nhận process con làm con và tự động gọi wait() khi process con kết thúc để xóa zombie ra khỏi hệ thống.

Nếu một process tạo process con mà không gọi wait() khi process con kết thúc thì thông tin của zombie sẽ được lưu mãi mãi ở bảng process của kernel. Khi một số lượng lớn zombie tồn tại, vượt quá bảng process của kernel thì sẽ không tạo được process mới nữa. Do zombie không thể bị tắt bởi signal nên chỉ có một cách để xóa nó, đó là kết thúc process cha (hoặc đợi process cha kết thúc). Việc kết thúc của process con xảy ra là không đồng bộ nên process cha không thể dự đoán được khi nào các process con của nó kết thúc.

Ta đã có 2 cách sử dụng wait() để tránh zombie là:

- Process cha gọi wait() hoặc waitpid() không kèm cờ WNOHANG, trong trường hợp này thì system call sẽ block nếu process con chưa kết thúc.

- Process cha định kì kiểm tra một process cụ thể bằng hàm waitpid() kèm theo cờ WNOHANG.

Tuy nhiên hai cách này khá bất tiện vì hoặc process cha sẽ bị block để đợi process con, hoặc process cha phải kiểm tra nhiều lần gây tốn CPU. Để khắc phục thì ta có thể sử dụng SIGCHL signal. SIGCHL signal sẽ được gửi đến process cha khi một process con của nó kết thúc. Mặc định thì tín hiệu này bị bỏ qua, nhưng ta có thể catch nó bằng signal handler.

* Dùng hàm **`wait()`** hoặc **`waitpid()`** ở tiến trình cha để lấy trạng thái trả về từ tiến trình con.

Ta có thể tạo ra tiến trình zombie bằng cách tạo ra 1 tiến trình cha, sau đó cho tiến trình này về trạng thái background, và chờ cho tiến trình con kết thúc. Lúc đó tiến trình sẽ trở thành zombie.

Ví dụ:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>
void main()
{
        pid_t pid;
        pid=fork();
```

```
        if(pid>0)
        {
                printf("hello %d \n",getpid());
                sleep(10);
                exit(0);

        }
        else
        if (pid==0)
        {
                printf("child %d \n",getpid());
                exit(0);

        }
        else
                printf ("errror \n");
}
```

Khi thực thi, tại tiến trình cha, bấm Ctrl + Z, sau đó hãy dùng lệnh **ps** để kiểm tra.

Lệnh hủy tiến trình

Hủy tiến trình:

kill [-signal | -s signal] pid

Thường kết hợp với lệnh **ps** để lấy ID tiến trình.

Các signal:

2: tương đương CTRL + C

9: buộc kết thúc

5: mặc định – kết thúc êm ái

19: tạm dừng

kill – l: liệt kê tất cả các signal

Ví dụ:

kill -9 11234 (kill tiến trình có pid = 11234)

Hàm kết thúc tiến trình

Hàm void **exit(int status)** kết thúc ngay lập tức tiến trình đang gọi.

Bất cứ file nào được mở bởi tiến trình thì được đóng và bất cứ tiến trình con nào được kế thừa bởi tiến trình ban đầu và tiến trình cha được gửi một tín hiệu SIGCHILD.

Khai báo hàm exit() trong C

void exit(int status)

Tham số

status: Đây là giá trị trạng thái được trả về tới tiến trình cha.

Ví dụ:

```
#include <stdio.h>
#include
<stdlib.h> int
main ()
{
        printf("Bat dau thuc thi chuong trinh
        ...\n"); printf("Thoat chuong trinh
        ...\n");
        exit(0);
        printf("Ket thuc chuong trinh
        ...\n"); return(0);
}
```

Hàm system

Cú pháp: **`int system(const char *string);`**

Thực thi lệnh trong đối số string và trả về kết quả khi thực hiện lệnh xong. Khi gọi hàm system(string), hệ thống sẽ thực hiện lệnh sh -c string.

Giá trị trả về:

0: thành công

-127: Không khởi động shell để thực hiện lệnh

-1: lỗi khác

-1: mã trả về khi thực hiện lệnh string.

Ví dụ:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
int re;
printf("Call system to execute ls -a\n"); re=system("ls -a");
if(re != -1) printf("System call ls is done!\n");
printf("Call system to execute ps -a\n"); re=system("ps -a");
if(re != -1) printf("System call ps is done!\n");
return 0;
}
```

Thực thi chương tình mới: execve()

System call `execve()` load một chương trình mới vào bộ nhớ của process. Ở quá trình này, chương trình cũ bị bỏ đi, stack, data và heap bị thay thế bởi program mới. Ta thường sử dụng `execve()` sau khi tạo process con bằng `fork()`.

Nhiều hàm thư viện khác, có tên bắt đầu bằng exec được wrap system call này.

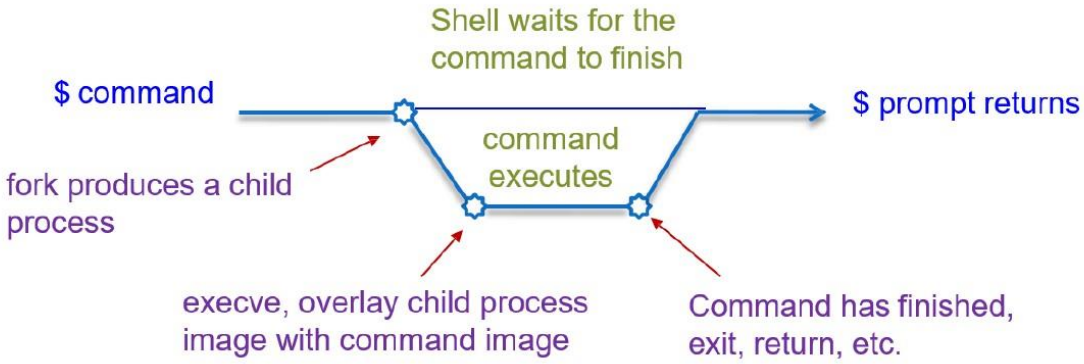
```
#include <unistd.h>
int execve(const char *pathname, char *const argv[], char *const envp[]);
```

Hàm này sẽ không trả về nếu thành công, trả về -1 nếu lỗi.

Và họ hàm được xây dựng trên `execve()`:

```
#include <unistd.h>
int execle(const char *pathname, const char *arg, ...
        /* , (char *) NULL, char *const envp[] */ );
int execlp(const char *filename, const char *arg, ...
         /* , (char *) NULL */);
int execvp(const char *filename, char *const argv[]);
int execv(const char *pathname, char *const argv[]);
int execl(const char *pathname, const char *arg, ...
        /* , (char *) NULL */);
```

Các hàm exec... Thực hiện theo cơ chế sau:



Các hàm exec… sẽ thay thế tiến trình gọi hàm bằng chương trình tương ứng trong tham số nhập của hàm. Vùng text, data, stack bị thay thế.

Chương trình được gọi bắt đầu thực thi ở hàm main, có thể nhận tham số nhập thông qua các tham số truyền.

Ví dụ:

```
#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 tập thực hành:

Bài 1:

Viết một chương trình lặp vô tận với lời gọi while(1); và thực thi nó. Đưa tiến trình này vào "background" thông qua việc gửi tín hiệu SIGTSTP bằng cách nhấn Ctrl + Z. Sử dụng lệnh ps để xác định PID của nó và sử dụng kill để kết thúc nó.

Bài 2:

Viết chương trình mà khi chạy nó sinh ra tiến trình con, để tiến trình con trở thành zombie. Tiến trình zombie này cần tồn tại trong hệ thống tối thiểu 10 giây (bằng cách dùng lời gọi sleep(10)). Sau đó dùng lệnh ps –l để xem trạng thái các tiến trình. Kết liễu zombie này bằng cách xác định PID của tiến trình cha nó và sử dụng lệnh kill.

Bài 3:

Sử dụng lời gọi system() để viết chương trình thực thi các lệnh Linux như sau:

- Tạo 1 thư mục "BaiTap" tại Desktop
- Tạo 2 thư mục "LyThuyet", "ThucHanh" trong "BaiTap"
- Tạo 1 file rỗng với tên là "test" trong "ThucHanh".
- Khi thực hiện xong các lệnh trên, nếu thực hiện thành công, hãy thực hiện thông báo đã hoàn thành.

--- HẾT ---