A/ LINUX CƠ BẢN

1. Linux là gì?

Windows, IOS, Android là các OS (OS); còn Linux thật ra không phải là 1 OS. Linux là một kernel của OS.

Linux được tạo ra vào năm 1991 bởi Linus Torvalds, một sinh viên của đại học Helsinki, Phần Lan. Khi đó ông đã quyết định nó sẽ hoàn toàn miễn phí và là một dự án open-source, có nghĩa là ai cũng có quyền tiếp cận mã nguồn gốc của nhân Linux để sử dụng, cải thiện và phát triển ra sản phẩm của riêng họ.


Linus Torvalds


Richarh Stallman

Lúc Linux mới được đưa ra, nó chỉ có khả năng quản lý phần cứng và 1 giao diện cho các phần mềm sử dụng. Rất may mắn, lúc này có 1 nhóm nghiên cứu (đứng đầu bởi Richarh Stallman) đang thực hiện một dự án có tên GNU (GNU's Not Unix *!) nhằm tạo ra 1 OS miễn phí với những ứng dụng/tiện ích tiêu chuẩn (như đồng hồ, giao diện người dùng,..) nhưng lại thiếu 1 Kernel tốt. Sau khi hai dự án trên được kết hợp, chúng ta đã có 1 OS hoàn chỉnh với 1 kernel tốt và đầy đủ các tiện ích tiêu chuẩn mà 1OS cần có.

*Unix là 1 OS thương mại được phát triển bởi AT&T và có giá bán hàng ngàn usd /user. GNU có giao diện gần giống với Unix nhưng là 1 OS miễn phí và không chứa code của Unix. (cái tên GNU là 1 kiểu viết tắt đệ quy của GNU's Not Unix)

2. Những OS nhân Linux

Qua mấy chục năm phát triển (với việc được phép tùy ý chỉnh sửa, nâng cấp, thêm bớt tính năng từ mã nguồn của 2 dự án GNU /Linux), nhiều nhà phân phối (còn được gọi là Linux distros hay Linux distributions) đã cải thiện và phát triển thêm nhiều tính năng hơn cho các OS chạy nhân Linux (ví dụ như trình nghe nhạc /xem video, quản lý email, trình duyệt, phần mềm office,..) và đóng gói tất cả lại thành 1 OS rồi phân phối nó cho người dùng. Hai Linux distributions nổi tiếng nhất là Debian và Redhat.

Debian: Dự án Debian tập hợp những kỹ sư phần mềm với mong muốn duy trì, cải thiện và phát triển 1 OS miễn phí chất lượng và mở cho đông đảo người dùng. Rất nhiều OS đã được ra đời nhờ vào dự án Debian (như Ubuntu, Kali Linux, Parrot Linux,..)

Red Hat Enterprise Linux: Trái ngược với Debian, Red Hat là một công ty thương mại cung cấp giải pháp nền tảng Linux cho các doanh nghiệp. Do Red Hat Linux dựa hoàn toàn vào các nguồn mở và miễn phí, nên Red Hat cũng công bố source code của mình trên nền tảng của họ. Tuy nhiên, Red Hat buộc bạn phải trả tiền để được phép download file cài đặt từ máy chủ của họ cũng như để được nhận cập nhật phần mềm, hỗ trợ sửa lỗi, hoặc đề thiết lập một chức năng phức tạp nào đó. Red Hat cũng là chủ sở hữu của 2 OS khác là Fedora và CentOS.

2 OS khá nổi tiếng khác đó là **MacOS** và **Android** được phát triển dựa trên Linux kernel.

3. Ứng dụng

Các OS chạy nhân Linux thường không phổ biến với người dùng bình thường mặc dù nó miễn phí (trong khi các OS có trả phí như MacOS hay Windows thì lại thông dụng). Thường ta chỉ gặp chúng khi làm việc ở cấp chuyên nghiệp như với các hệ thống mạng, hệ thống server, các thiết bị IoT,..)

Do đặc tính mã nguồn mở và miễn phí, bất cứ nhà phát triển nào cũng có thể tiếp cận mã nguồn của Linux kernel từ đó tùy biến nó để tạo ra OS phục vụ cho mục đích riêng, nên hiện tại có khá nhiều OS chạy nhân Linux.

Các OS nhân Linux thường yêu cầu ít tài nguyên phần cứng (so với Windows), nên 1 số được tùy biến để có thể chạy trên các MicroComputers (máy tính siêu nhỏ, như Raspberry Pi, với RAM ~1GB và CPU~1GHz là có thể làm việc được).

(Raspberry Pi)

Tuy nhiên điểm bất lợi của những OS nhân Linux đó là ứng dụng, tính năng và game không nhiều và đa dạng bằng Windows. Đồng thời việc cài đặt và sử dụng các ứng dụng cũng như sửa lỗi hệ thống sẽ không đơn giản trong khi đó lại là những việc hay phải làm!

4. Security

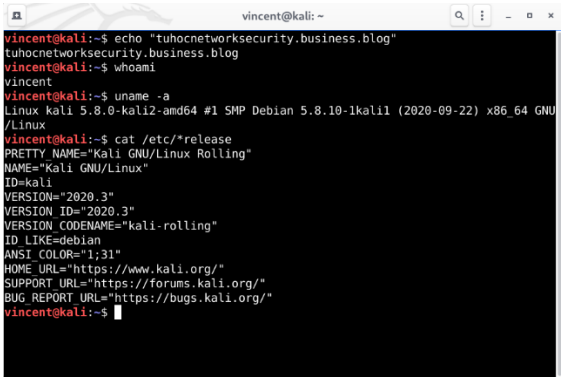
Nhiều người cho rằng kiến trúc tổ chức của OS nhân Linux (bao gồm MacOS) khiến chúng an toàn và không (hoặc khó) bị dính mã độc. Đây là quan niệm sai!

Trên thực tế Linux ít bị mã độc vì các hackers chọn mục tiêu tấn công là các OS có số lượng người dùng đông đảo và "ngây thơ", để dễ thành công và "kiếm chác" được ở mức cao nhất. Bởi vậy Windows OS là mục tiêu số 1 (và số 2 là Android). Dạo gần đây thì các hacker cũng rất ưu ái MacOS /IOS vì cho là người dùng các OS này thuộc hàng "giàu có" và cũng ngây thơ không kém. Còn người dùng các phiên bản Linux còn lại thì thường là dân chuyên nghiệp, đã "không giàu" lại "thiếu ngây thơ" nên thôi cho qua!

5. GUI và CLI

Mỗi OS thường có 2 giao diện để người dùng tương tác. Một là GUI (Graphical User Interface) và 2 là CLI (Command Line).

Giao diện GUI chính là giao diện đồ họa giống như trên Windows OS /MacOS mà bình thường mọi người vẫn hay sử dụng, GUI thân thiện và đẹp nên rất phổ biến với người dùng không chuyên kỹ thuật. Còn với giao diện dòng lệnh (CLI) thì phải tương tác chủ yếu thông qua các câu lệnh gõ từ bàn phím nên khá khó khăn với những người không chuyên.



Giao diện CLI thường được các chuyên gia hệ thống IT ưa thích, vì CLI nhẹ hơn và nhanh hơn GUI đồng thời cũng là giao diện đầu tiên được con người tạo ra để tương tác với hệ thống máy tính - chỉ dành cho các nhà nghiên cứu hay các kỹ sư.

---

https://www.guru99.com/components-of-operating-system.html
https://afteracademy.com/blog/what-is-kernel-in-operating-system-and-what-are-the-various-types-of-kernel
https://www.softwaretestinghelp.com/unix-vs-linux/#:~:text=Unix%20is%20not%20free.,Hence%2C%20UNIX%20is%20extremely%20expensive.
https://www.debian.org/intro/about
https://www.quora.com/How-does-Red-Hat-make-money

B/ LẬP TRÌNH C TRÊN LINUX

1. Trình dịch C trên Linux

Cài gói chương trình dịch: **sudo apt-get install build-essential**

Ta cần nắm rõ các vị trí chứa trình biên dịch, tập tin thư viện, các tập tin header, cũng như các tập tin chương trình sau khi dịch:

- Trình biên dịch gcc: thường được đặt trong thư mục **/usr/bin** hoặc **/usr/local/bin** (kiểm tra bằng lệnh which gcc).

- Các tập tin header: /usr/include hay /usr/local/include

- Các tập tin thư viện liên kết: **/lib** hoặc **/usr/local/lib**

- Các thư viện chuẩn của gcc: **/usr/lib/gcc-lib**

2. Các tập tin header:

Nếu ta có các tập tin header của riêng mình trong một thư mục khác thư mục mặc định thì có thể chỉ rõ đường dẫn đến thư mục khi biên dịch bằng tùy chọn –I, ví dụ: ***$ gcc –I /usr/mypro/include test.c –o test***

Khi chúng ta sử dụng một hàm nào đó của thư viện hệ thống trong chương trình C, ngoài việc phải biết khai báo nguyên mẫu của hàm, chúng ta cần phải biết hàm này được định nghĩa trong tập tin header nào. Trình man sẽ cung cấp cho chúng ta các thông tin này rất chi tiết. Ví dụ, với hàm kill(), man sẽ chỉ ra 2 header file là types.h và signal.h.

3. Các tập tin thư viện

Muốn tạo ra chương trình thực thi ta cần phải có các tập tin thư viện. Trong Linux, các tập tin thư viện tĩnh của C có phần mở rộng là .a, .so, .sa và bắt đầu bằng tiếp đầu ngữ lib. Ví dụ libutil.a hay libc.so là tên các thư viện liên kết trong Linux.

- Linux có 2 loại liên kết: tĩnh (static) và động (dynamic). Thư viện liên kết động trên Linux thường có phần mở rộng là .so, ta có thể dùng lệnh ls /usr/lib hoặc ls /lib để xem các thư viện hệ thống đang sử dụng. Khi biên dịch, thông thường trình liên kết (ld) sẽ tìm thư viện ở /usr/lib và /lib. Để chỉ định tường minh một thư viện nào đó, ta làm như sau: **$ gcc test.c –o test /usr/lib/libm.a** . Bởi vì thư viện

bắt buộc phải có tiếp đầu ngữ lib và có phần mở rộng là .a hoặc .so, trình dịch cho phép sử dụng tùy chọn **–l** ngắn gọn như sau: **$ gcc test.c –o test -lm** , ta sẽ thấy rằng gcc sẽ mở rộng –l thành tiếp đầu ngữ lib và tìm libm.a hoặc libm.so trong thư mục chuẩn để liên kết.

- Nếu thư viện của chúng ta nằm ở một thư mục khác, chúng ta có thể chỉ định gcc tìm kiếm trực tiếp với tùy chọn –L như sau: **$ gcc test.c –o test –L /usr/myproj/lib -ltool** (Lệnh này cho phép liên kết với thư viện libtool.a hoặc libtool.so trong thư mục/usr/myproj/lib).

4. Thư viện liên kết trên Linux

- Hình thức đơn giản nhất của thư viện là tập hợp các tập tin **.o** do trình biên dịch tạo ra ở bước biên dịch với tùy chọn **–c**. Ví dụ **$gcc –c helloworld.c** trình biên dịch chưa tạo ra tập tin thực thi mà tạo ra tập tin đối tượng **helloworld.o**. Tập tin này chứa các mã máy của chương trình đã được sắp xếp lại. Nếu muốn tạo ra tập tin thực thi, chúng ta gọi trình biên dịch thực hiện bước liên kết: **$gcc helloworld.o –o helloworld.**

Trình biên dịch sẽ gọi tiếp trình liên kết **ld** tạo ra định dạng tập tin thực thi cuối cùng. Ở đây, nếu chúng ta không sử dụng tùy chọn **–c**, trình biên dịch sẽ thực hiện cả hai bước đồng thời.

a) Thư viện liên kết tĩnh

- Là các thư viện khi liên kết trình dịch sẽ lấy toàn bộ mã thực thi của hàm trong thư viện đưa vào chương trình chính. Chương trình sử dụng thư viện liên kết tĩnh chạy độc lập với thư viện sau khi biên dịch xong. Nhưng khi nâng cấp và sửa đổi, muốn tận dụng những chức năng mới của thư viện thì chúng ta phải biên dịch lại chương trình.

Ví dụ sử dụng liên kết tĩnh:

```
/* cong.c */
int cong( int a, int b ) {
  return a + b;
}
/* nhan.c */
long nhan( int a, int b ) {
  return a * b;
}
```

Thực hiện dịch để tạo ra hai tập tin thư viện đối tượng **.o : $ gcc –c cong.c nhan.c**

Để một chương trình nào đó gọi được các hàm trong thư viện trên, ta cần tạo 1 tập tin header **.h** khai báo các nguyên mẫu hàm :

```
/* lib.h */
int cong( int a, int b );
long nhan( int a, int b );
```

Cuối cùng, tạo ra chương trình chính **program.c** gọi hai hàm này.

```
/* program.c */
#include <stdio.h>
#include "lib.h"
int main () {
  int a, b;
  printf( "Nhap vào a : " ); scanf( "%d", &a );
  printf("Nhap vào b : " ); scanf( "%d", &b );
  printf( "Tổng %d + %d = %d\n", a, b, cong( a, b ) );
  printf( "Tich %d * %d = %ld\n", a, b, nhan( a, b ) );
  return ( 0 );
}
```

Ta biên dịch và liên kết với chương trình chính như sau:

```
$ gcc –c program.c
$ gcc program.o cong.o nhan.o –o program
```

Sau đó thực thi chương trình: **$ ./program**

Ở đây **.o** là các tập tin thư viện đối tượng. Các tập tin thư viện **.a** là chứa một tập hợp các tập tin **.o**. Tập tin thư viện **.a** thực ra là 1 dạng tập tin nén được tạo ra bởi chương trình **ar**. Chúng ta hãy yêu cầu **ar** đóng **cong.o** và **nhan.o** vào **libfoo.a**

$ ar cvr libfoo.a cong.o nhan.o

Sau khi đã có được thư viện **libfoo.a**, chúng ta liên kết lại với chương trình theo cách sau:

$ gcc program.o –o program libfoo.a

Ta có thể sử dụng tùy chọn **–l** để chỉ định thư viện khi dịch thay cho cách trên. Tuy nhiên **libfoo.a** không nằm trong thư mục thư viện chuẩn, cần phải kết hợp với tùy chọn **–L** để chỉ định đường dẫn tìm kiếm trong thư mục hiện hành. Đây là cách dịch: **$ gcc program.c –o program –L. –lfoo**

Ta có thể sử dụng lệnh **nm** để xem các hàm đã biên dịch sử dụng trong tập tin chương trình, tập tin đối tượng **.o** hoặc tập tin thư viện **.a**. Ví dụ: **$ nm cong.o**

b) Thư viện liên kết động

Khuyết điểm của thư viện liên kết tĩnh là nhúng mã nhị phân kèm theo chương trình khi dịch, do đó tốn không gian đĩa và khó nâng cấp. Thư viện liên kết động được dùng để giải quyết vấn đề này. Các hàm trong thư viện liên kết động không trực tiếp đưa vào hương trình lúc dịch và liên kết, trình liên kết chỉ lưu thông tin tham chiếu đến các hàm trong thư viện liên kết động. Vào lúc chương trình nhị phân thực thi, OS sẽ nạp các chương trình liên kết cần tham chiếu vào bộ nhớ. Như vậy, nhiều chương trình có thể sử dụng chung các hàm trong 1 thư viện duy nhất.

Tạo thư viện liên kết động: Khi biên dịch tập tin đối tượng để đưa vào thư viện liên kết động, chúng ta phải thêm tùy chọn **–fpic** (PIC- Position Independence Code – mã lệnh vị trí độc lập). Ví dụ: biên dịch lại 2 tập tin **cong.c** và **nhan.c**

$ gcc –c –fpic cong.c nhan.c

Để tạo ra thư viện liên kết động, chúng ta không sử dụng trình **ar** như với thư viện liên kết tĩnh mà dùng lại **gcc** với tùy chọn **–shared**. **$ gcc –shared cong.o nhan.o –o libfoo.so**

Nếu tập tin **libfoo.so** đã có sẵn trước thì không cần dùng đến tùy chọn ***–o $ gcc –shared cong.o nhan.o libfoo.so*** Bây giờ chúng ta đã có thư viện liên kết động **libfoo.so**. Biên dịch lại chương trình như sau***: $ gcc program.c –o program –L. –lfoo***

Sử dụng thư viện liên kết động: Khi OS chương trình **program**, nó cần tìm thư viện **libfoo.so** ở đâu đó trong hệ thống. Ngoài các thư mục chuẩn, Linux còn tìm thư viện liên kết động trong đường dẫn của biến môi trường **LD_LIBRARY_PATH**. Do **libfoo.so** đặt trong thư mục hiện hành, không nằm trong các thư mục chuẩn nên ta cần đưa thư mục hiện hành vào biến môi trường **LD_LIBRARY_PATH**: **$ LD_LIBRARY_PATH=.: $ export LD_LIBRARY_PATH.** Kiểm tra xem OS có thể tìm ra tất cả các thư viện liên kết động mà chương trình sử dụng hay không: **$ ldd program** rồi chạy chương trình sử dụng thư viện liên kết động này: **$./program**

Một khuyết điểm của việc sử dụng thư viện liên kết động đó là thư viện phải tồn tại trong đường dẫn để OS tìm ra khi chương trình được gọi. Nếu không tìm thấy thư viện, OS sẽ chấm dứt ngay chương trình cho dù các hàm trong thư viện chưa được sử dụng. Ta có thể chủ động nạp và gọi các hàm trong thư viện liên kết động mà không cần nhờ vào OS bằng cách gọi hàm liên kết muộn.

5. Ví dụ

- Tạo source code: Dùng bất kì trình soạn thảo nào tạo file hello.c (có thể dùng vim: ***vim hello.c***)

```
#include <stdio.h>
int main() {
```

printf("Hello World \n");
return 0;
}
- Dịch: ***gcc -o hello hello.c***
- Chạy thử : ***./hello***
- Debug:

gcc -g -o hello hello.c *(// thêm option [-g] để cho phép chạy debug)*
gdb hello *(// vào console của chương trình debug GDB)*

- Trong console của GDB, ta có thể thực hiện các lệnh của linux: (gdb) shell [lenh_linux]
Ví dụ: (gdb) shell clear / (gdb) shell ls
- Để đặt break point ở một vị trí nào đó: (gdb) break [so_dong] hoặc (gdb) break [ten_ham]
Ví dụ: đặt breakpoint tại dòng thứ 9: (gdb) break 9 hoặc đặt breakpoint tại hàm main: (gdb) break main
- Để xóa breakpoint: (gdb) delete [so_thu_tu_break_point]
Ví dụ: Xóa breakpoint thứ 2: (gdb) delete 2
- Bắt đầu chạy debug: (gdb) run
- Chạy tới dòng lệnh : (gdb) next [so_dong]
- Chạy vào bên trong thân hàm: (gdb) step
- Chạy đến breakpoint kế: (gdb) continue
- Xem giá trị biến: (gdb) display [ten_bien]
- In giá trị biến ra console: (gdb) print [ten_bien] // In địa chỉ biến: (gdb) print &[ten_bien]
- Hiển thị kiểu dữ liệu của biến: (gdb) ptype [ten_bien] hoặc (gdb) whatis [ten_bien]
- Gán giá trị cho 1 biến: (gdb) set variable [ten_bien] = [value]

C/ XỬ LÝ PROCESS (PROCESS) TRONG LINUX

Khái quát

- Một trong những đặc điểm nổi bật của Linux là khả năng chạy đồng thời nhiều chương trình. Một chương trình có thể bao gồm nhiều process kết hợp với nhau.

- Các lệnh của Linux là những lệnh có khả năng kết hợp và truyền dữ liệu giữa các process 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 process (IPC – Inter Process Comunication). Xây dựng ứng dụng trong môi trường đa process như Linux là công việc khó khăn. Không như môi trường đơn nhiệm, trong môi trường đa nhiệm process có tài nguyên rất hạn hẹp - khi hoạt động phải luôn ở trạng thái tôn trọng và sẵn sàng nhường quyền xử lý CPU cho các process khác ở bất kỳ thời điểm nào khi hệ thống có yêu cầu. Nếu process của chúng ta được xây dựng không tốt, khi đổ vỡ và gây ra lỗi nó có thể làm treo các process khác trong hệ thống hay thậm chí phá vỡ (crash) OS.

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). Process của Linux có các trạng thái: + Đang chạy (running) : là lúc process chiếm quyền sứ dụng CPU để thực hiện các công việc của mình. + Chờ (waiting) : process bị OS tước quyền sứ dụng CPU, và chờ đến lược cấp phát khác. + Tạm dừng (suspend) : OS tạm dừng process, đưa vào trạng thái ngủ (sleep). Khi cần thiết và có nhu cầu, OS sẽ đánh thức (wake up) hay nạp lại mã lệnh của process vào bộ nhớ và cấp phát tài nguyên CPU để process tiếp tục hoạt động.

- Trên dòng lệnh, thay vì dùng lệnh Ctrl+z, ta có thể sử dụng lệnh **`bg`** để đưa một process vào hoạt động phía hậu trường. Chúng ta cũng có thể yêu cầu 1 process chạy nền bằng cú pháp **`&`**. Ví dụ: **`$ls –R & .`** Lệnh **`fg`** sẽ đem process trở về hoạt động ưu tiên phía

trước. Trong chương trình, ta có thể dùng lệnh **fork()** để nhân bản process mới từ process cũ. Hoặc dùng lệnh **system()** để gọi 1 process của OS. Hàm **exec()** cũng có khả năng tạo ra process mới khác.

Cấu trúc tiến trình

Ta hãy xem OS quản lý process như thế nào? Nếu có 2 người dùng: **user1** và **user2** cùng đăng nhập vào chạy chương trình **grep** đồng thời, thực tế, OS sẽ quản lý và nạp mã của chương trình **grep** vào hai vùng nhớ khác nhau và gọi mỗi phân vùng như vậy là process. Hình sau cho thấy cách phân chia chương trình **grep** thành hai process cho 2 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**

PID **101**
Code
Data s=**abc**
Library
filede

s
file1

user2
$grep cde file2

PID **102**
Code
Data s=**cde**
Library
filede

s
file2

2 người dùng **user1** và **user2** có thể ở 2 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 OS 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 process mà OS đang kiểm soát, Ví dụ: **$ps -af**
Mỗi process 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 process mới yêu cầu khởi động, OS sẽ chọn lấy một số (chưa bị process nào đang chạy chiếm giữ) trong khoảng số nguyên trên và cấp phát cho process mới. và khi process chấm dứt, OS sẽ thu hồi số **PID**. **PID** bắt đầu từ giá trị 2 bởi vì giá trị 1 được dành cho process đầu tiên gọi là **init**. Process **init** được và chạy ngay khi ta khởi động OS. **init** là process quản lý và tạo ra mọi process con khác. Ở ví dụ trên, chúng ta thấy lệnh **ps -af** sẽ hiển thị 2 process **grep** chạy bởi **user1** và **user2** với số **PID** lần lượt là **101** và **102**.
- Trừ mã lệnh và thư viện có thể chia sẻ, còn dữ liệu thì không thể chia sẻ bởi các tiến trình. Mỗi process sở hữu phân đoạn dữ liệu riêng. Ví dụ process **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 process 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 process khởi động, thường OS 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**. OS cấp phát số mô tả file cho mỗi process và lưu riêng chúng trong bảng mô tả file của process đó.
- Ngoài ra, mỗi process có riêng ngăn xếp stack để lưu biến cục bộ và các giá trị trả về sau lời gọi hàm. Process 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
- OS lưu giữ một cấu trúc danh sách bên trong hệ thống gọi là process table để quản lý tất cả thông tin chi tiết về các process đang chạy. Ví dụ khi chúng ta gọi lệnh **ps**, Linux đọc thông tin trong bảng process này và hiển thị những lệnh hay tên process được gọi: thời gian chiếm giữ CPU của process, tên người sử dụng process, …

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

4. Tạo lập tiến trình

a) Gọi process mới bằng hàm `system()`

- Chúng ta có thể gọi một process khác bên trong một chương trình đang thực thi bằng hàm **`system()`**. Có nghĩa là chúng ta có thể tạo ra một process mới từ một process 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

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 ta được sử dụng để gọi lệnh "**`ps -ax`**" của OS.

b) Thay thế process hiện hành với các hàm `exec`

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

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

- Thay thế process đôi khi bất lợi với chúng ta. Đó là process mới chiếm giữ toàn bộ không gian của process cũ và chúng ta sẽ không có khả năng kiểm soát cũng như điều khiển tiếp process 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 process. **`Fork()`** là 1 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 process hiện hành 2 process (dĩ nhiên OS phải cấp phát thêm không gian bộ nhớ để process mới hoạt động). Process ban đầu gọi là process cha (parent process) trong khi process mới gọi là process con (child process). Process con sẽ có một số định danh **`PID`** riêng biệt. ngoài ra, process con còn mang thêm một định danh **`PPID`** là số định danh **`PID`** của process cha.
- Sau khi tách tiến trình, mã lệnh thực thi ở cả hai process được sao chép là hoàn toàn giống nhau. Chỉ có 1 dấu hiệu để chúng ta có thể hận dạng process cha và process con, đó là trị trả về của hàm **`fork()`**.

Bên trong process con, hàm **`fork()`** sẽ trả về trị 0. Trong khi bên trong process cha, hàm **`fork()`** sẽ trả về trị số nguyên chỉ là **`PID`** của process con vừa tạo. Trường hợp không tách được process, **`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 process con đặt ở đây
        break;
default: printf( "Day la tien trinh cha" ); // mã lệnh dành cho process cha đặt ở đây
        break;
}
```

d) Kiểm soát và đợi process con

- Khi **fork()** tách process chính thành hai process cha và con, trên thực tế cả 2 process đều hoạt động độc lập. Đôi lúc process cha cần phải đợi process con thực hiện xong tác vụ thì mới tiếp tục thực thi. Ở ví dụ trên, khi thực thi, chúng ta sẽ thấy rằng process cha đã kết thúc mà process con vẫn in thông báo và cả 2 đều tranh nhau gởi kết quả ra màn hình. Ta không muốn điều này - khi process cha kết thúc thì process 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 process cha dừng lại chờ process con kết thúc trước khi thực hiện tiếp các lệnh điều khiển trong process cha. **wait()** làm cho sự liên hệ giữa process cha và process con trở nên tuần tự. Khi process con kết thúc, hàm sẽ trả về số **PID** tương ứng của process con. Nếu ta truyền thêm đối số **stat_loc** khác **NULL** cho hàm thì **wait()** cũng sẽ trả về trạng thái mà process con kết thúc trong biến **stat_loc**. Chúng ta có thể sử dụng các macro khai báo sẵn trong **sys/wait.h**