**LƯU AN PHÚ**

TƯƠNG TÁC

VỚI

HỆ ĐIỀU HÀNH

LINUX

Facebook group: Cùng Nhau Học Linux Kernel

LỜI MỞ ĐẦU

Xuất phát từ nhu cầu thực tiễn của lĩnh vực Linux embedded, chúng tôi là các kỹ sư đang làm việc trong các dự án đó đã quyết định biên soạn ra bộ sách này. Sau khi nghiên cứu chương trình học của các trường đại học nổi tiếng trên thế giới. Bộ sách bao gồm 2 cuốn, trong đó cuốn một sẽ tập trung vào lý thuyết và kỹ thuật lập trình trên tầng user-space, cuốn hai sẽ về kernel-space của hệ điều hành.

Tư tưởng khi chúng tôi viết cuốn sách này là tập trung nhiều về thực hành, ngoài ra các lý thuyết cũng như thực hành đều áp dụng trên các phiên bản OS và hardware mới nhất. Việc này nhằm giúp người học giảm được sự bỡ ngỡ khi bước vào dự án thực tế.

Bộ sách được tham khảo từ 3 cuốn sách nổi tiếng đó là “Advanced programming in the unix environment”, “UnderStanding The Linux Kernel” và “Linux Device Drivers”.

Trong mỗi một chương sẽ có lý thuyết và bài tập để người học có thể thực hành. Nếu các bạn có thời gian thì nên học và thực hành từ đầu đến cuối. Đó là phương pháp học bài bản và dễ hiểu nhất. Trong trường hợp thời gian không cho phép, các bạn có thể học và thực hành luôn theo cuốn sách thứ 2 của bộ sách này.

Ngoài ra chúng tôi đã tạo ra một group facebook riêng để tập hợp tất cả mọi người trên lãnh thổ Việt Nam, những người có chung niềm đam mê về Linux embedded. Group có tên là “Cùng nhau học Linux kernel”, nếu muốn các bạn có thể join vào cùng học hỏi với chúng tôi. Ở đây người mới luôn luôn được chào đón.

Trong quá trình biên soạn không thể tránh được nhiều sai sót. Nếu thấy điểm nào cần cải tiến, các bạn hãy đặt câu hỏi cho chúng tôi thông qua group facebook ở trên hoặc gửi mail cho mình tại địa chỉ: [luuanphu@gmail.com](mailto:luuanphu@gmail.com).

Tôi muốn gửi lời cảm ơn đến bạn Trương Văn Huy, người đã đóng góp rất nhiều công sức cho bộ sách này.

Ngoài ra tôi cũng gửi lời cảm ơn đến vợ tôi, người đã tạo điều kiện rất nhiều để tôi có thể theo đuổi được đam mê của mình.

|  |  |
| --- | --- |
|  | Hà Nội, ngày 23 tháng 10 năm 2018  Tác giả  Lưu An Phú |

MỤC LỤC

[CHƯƠNG 1. TỔNG QUAN VỀ CÁC HỆ ĐIỀU HÀNH HỌ UNIX 4](#_Toc528077097)

[1.1 Các chức năng chính của hệ điều hành 4](#_Toc528077098)

[1.2 Kiến trúc của hệ điều hành 5](#_Toc528077099)

[1.3 Giao diện người sử dụng 5](#_Toc528077100)

[1.4 Chương trình và tiến trình 7](#_Toc528077101)

[1.5 Unix là một hệ điều hành đa nhiệm 7](#_Toc528077102)

[1.6 Hệ thống file system 7](#_Toc528077103)

[1.7 Bài tâp 8](#_Toc528077104)

[CHƯƠNG 2. FILE TRONG LINUX 10](#_Toc528077105)

[2.1 Giới thiệu 10](#_Toc528077106)

[2.2 Các loại tệp tin 10](#_Toc528077107)

[2.3 File descriptors 11](#_Toc528077108)

[2.4 Hàm open 11](#_Toc528077109)

[2.5 Hàm close 12](#_Toc528077110)

[2.6 Hàm lseek 12](#_Toc528077111)

[2.7 Hàm read 15](#_Toc528077112)

[2.8 Hàm write 15](#_Toc528077113)

[2.9 Hàm ioctl 15](#_Toc528077114)

[2.10 Thư mục /dev 20](#_Toc528077115)

[CHƯƠNG 3. CÁCH QUẢN LÝ HỆ THỐNG FILE TRONG LINUX 22](#_Toc528077116)

[3.1 Giới thiệu 22](#_Toc528077117)

[3.2 Giới thiệu về cây thư mục trong Linux 22](#_Toc528077118)

[3.3 Các thuộc tính của file 23](#_Toc528077119)

[3.4 Quyền truy cập file 25](#_Toc528077120)

[3.5 Hàm chmod, fchmod và fchmodat 26](#_Toc528077121)

[3.6 Quyền sở hữu của file và thưc mục mới 27](#_Toc528077122)

[3.7 Hàm access và faccessat 27](#_Toc528077123)

[3.8 Hệ thống file trong Linux 28](#_Toc528077124)

[3.9 Symbolic Link 30](#_Toc528077125)

[3.10 Tạo và đọc symbolic link 30](#_Toc528077126)

[3.11 Thư mục 31](#_Toc528077127)

[3.12 Hàm mkdir, mkdirat và hàm rmdir 31](#_Toc528077128)

[3.13 Đọc thư mục 31](#_Toc528077129)

[3.14 Hàm chdir, fchdir và hàm getcwd 32](#_Toc528077130)

[3.15 Giới thiệu về device file 32](#_Toc528077131)

[CHƯƠNG 4. GIỚI THIỆU VỀ PROCESS 33](#_Toc528077132)

[4.1 Mở đầu 33](#_Toc528077133)

[4.2 Hàm main 33](#_Toc528077134)

[4.3 Terminate một process 34](#_Toc528077135)

[4.4 Các hàm exit 34](#_Toc528077136)

[4.5 Hàm atexit 35](#_Toc528077137)

[4.6 Tham số truyền vào từ command-line 37](#_Toc528077138)

[4.7 Danh sách biến môi trường 38](#_Toc528077139)

[4.8 Biến môi trường 39](#_Toc528077140)

[4.9 Cấu trúc bộ nhớ của một chương trình 40](#_Toc528077141)

[4.10 Cấp phát bộ nhớ trong process 41](#_Toc528077142)

[4.11 Thư viện chung 42](#_Toc528077143)

# TỔNG QUAN VỀ CÁC HỆ ĐIỀU HÀNH HỌ UNIX

Ngày nay hệ điều hành đã trở lên quen thuộc với tất cả chúng ta. Tuy nhiên vào những năm 50 của thế kỷ trước, khi đó OS chưa ra đời, người ta phải nạp thẳng code vào máy tính. Mỗi máy tính tại một thời điểm chỉ chạy một chương trình và một chương trình sẽ phải điều khiển toàn bộ máy tính. Với máy tính tại thời điểm đó có kiến trúc đơn giản (không có chuột, bàn phím, màn hình, loa…) nên việc người lập trình viên quản lý toàn bộ máy tính bằng code của mình là khả thi. Tuy nhiên kiến trúc máy tính và yêu cầu tính toàn càng ngày càng phức tạp, do đó người ta cần đến một hệ thống có thể quản lý được máy tính và hỗ trợ nhiều nhất có thể đối với người lập trình viên. Từ yêu cầu thực tế đó hệ điều hành được ra đời.

Các hệ điều hành được ra đời sớm nhất có thể kể đến là GM-NAA I/O, BESYS, SOS, TENEX, Unix… Tuy nhiên thành công nhất chỉ có Unix, nó được thiết kế dựa trên rất nhiều các lý thuyết toán học. Do đó sau nửa thế kỷ trôi qua, phần thiết kế lõi của nó cũng không cần phải chỉnh sửa nhiều. Kiến trúc của Unix được áp dụng cho rất nhiều hệ điều hành phổ biến ngày nay như Android, Window, Linux, MACOS… Và chúng được gọi là các hệ điều hành họ Unix.

Trong chương này, chúng ta sẽ cùng nhau học về các tính chất chung của những hệ điều hành kế thừa từ Unix.

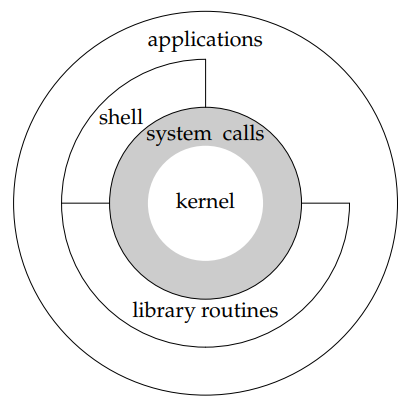
## Các chức năng chính của hệ điều hành

Là lớp vỏ bảo vệ cho hardware của hệ thống: Hiểu một cách đơn giản thì hardware của hệ thống giống như lòng trứng. Để tương tác trực tiếp với chúng thì người lập trình viên phải cẩn thận và hiểu rõ phần cứng. Tuy nhiên khi có hệ điều hành thì việc đó sẽ không cần thiết nữa. Hệ điều hành sẽ tạo ra lớp vỏ trứng bao quanh toàn bộ phần cứng. Lúc này lập trình viên thay vì tương tác với phần cứng thì sẽ tương tác với lớp vỏ là hệ điều hành, sau đó hệ điều hành sẽ là người làm việc với phần cứng.

Là đối tượng duy nhất sở hữu, quản lý và phân phối phần cứng trong hệ thống: Khi hệ thống đi vào hoạt động sẽ có rất nhiều đối tượng tồn tại trong nó – Ví dụ như các chương trình Word, Excel, Chrome, … Và hệ điều hành cũng là một đối tượng nằm trong số đó. Tuy nhiên khác với các đối tượng còn lại, hệ điều hành là đối tượng được khởi động đầu tiên trong hệ thống, nó khởi tạo toàn bộ phần cứng và chiếm luôn quyền sở hữu chúng. Sau đó nó sẽ khởi tạo các đối tượng còn lại và quản lý, phân phối phần cứng cho toàn hệ thống.

Cung cấp môi trường hoạt động và xử lý xung đột giữa các đối tượng: Do hệ điều hành là đối tượng đầu tiên được tạo ra trong hệ thống. Sau đó tất cả các đối tượng còn lại đều được sinh ra bởi hệ điều hành, do đó nó có toàn quyền điều khiển các đối tượng còn lại. Nó có thể sinh ra một đối tượng mới, tạm dừng một đối tượng đang chạy hoặc kết thúc vòng đời của chúng. Mỗi khi trong hệ thống xuất hiện trạng thái xung đột giữa các đối tượng thì hệ điều hành sẽ đứng ra phân xử và nó sẽ trực tiếp thi hành quyết định của mình. Tất cả các đối tượng còn lại đều phải tuân theo quyết định của nó.

## Kiến trúc của hệ điều hành



Hình 1: Kiến trúc của hệ điều hành họ Unix

Có rất nhiều cách để phân chia các lớp trong hệ điều hành, không có cách nào là chính xác hoàn toàn. Việc phân chia hệ điều hành ra thành những lớp nào đều phụ thuộc vào từng góc nhìn khác nhau. Trong phạm vi cuốn sách này, chúng ta sẽ coi hệ điều hành gồm 4 lớp cơ bản như sau:

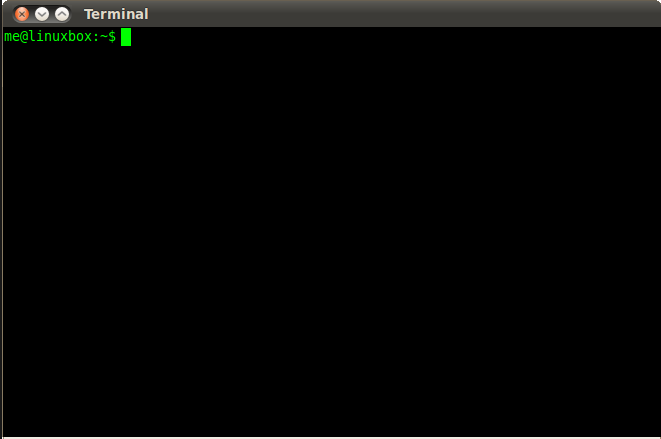
+ Lớp kernel: Đây là lớp trong cùng, nó bao ngoài phần cứng, quản lý và cung cấp những chức năng cơ bản của hệ điều hành như: Lập lịch, quản lý bộ nhớ, quản lý ngắt,…

+ Lớp system call: Do cách thiết kế hệ điều hành không cho phép các ứng dụng từ tầng application được phép truy cập thẳng vào lớp kernel (Để tránh 1 lỗi trên tầng application có thể làm sập hệ thống). Nên họ đã thiết kế 1 lớp để ngăn cách gọi là lớp system call. Nhiệm vụ của lớp system call là cung cấp các đầu hàm (Ví dụ như read(), write()) cho lớp application sử dụng.

+ Lớp application: Đây là lớp ngoài cùng của hệ điều hành, là nơi tương tác với user. Các tiến trình như word, excel... mà user sử dụng đều được chạy ở lớp này.

## Giao diện người sử dụng

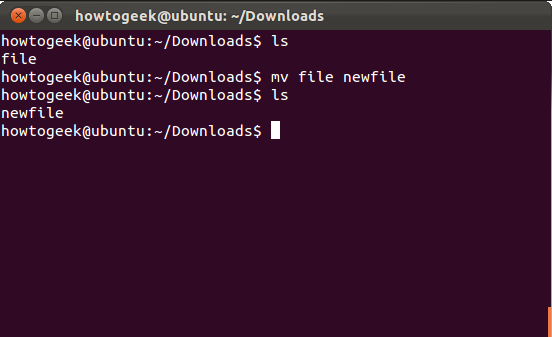
Khác với hệ điều hành như Window – hầu hết sự tương tác diễn ra thông qua giao diện cửa sổ. Trong các hệ điều hành như Linux thì giao diện dòng lệnh mới là công cụ đủ mạnh để tương tác với hệ điều hành. Có rất nhiều công việc bạn không thể sử dụng giao diện cửa sổ để xử lý mà bắt buộc phải sử dụng dòng lệnh. Dưới đây là hình ảnh giao điện bằng dòng lệnh của Linux:



Hình 2: Giao diện dòng lệnh của Linux

Trong các hệ thống embedded người ta sẽ disable chế độ graphic, do đó khi boot lên màn hình chỉ có duy nhất cửa sổ command line được hiện lên.

Các câu lệnh sẽ tuân theo cấu trúc như sau:



Hình 3: Ví dụ một câu lệnh trong Linux

Trong đó phần đầu câu lệnh (mv) sẽ là tên của chương trình thực hiện câu lệnh đó. Bản chất của tất cả các câu lệnh là các chương trình được đặt trong một thưc mục nhất định của hệ thống. Thông thường sẽ là thư mục /bin và /sbin. Khi chúng ta gọi một câu lệnh thì hệ thống sẽ run chương trình trùng tên nằm trong 2 thư mục đó. Trong trường hợp không có chương trình nào trùng tên hệ thống sẽ báo lỗi.

Phần tham số theo sau tên câu lệnh (file new file) là các chuỗi string được truyền cho chương trình thực hiện câu lệnh. Một lưu ý với các bạn là tất cả các tham số ở dạng chữ hoặc số khi truyền vào cho chương trình đều chuyển về dạng string.

Một số câu lệnh khi thực hiện cần những quyền nhất định. Ví dụ khi tương tác với các file hệ thống chúng cần quyền của người quản trị… Các quyền của một câu lệnh sẽ phụ thuộc và user nào đang thực hiện câu lệnh đó. Do vậy trong một số trường hợp, chúng ta phải tiến hành chuyển đồi user để cấp quyền cao hơn cho câu lệnh đó.

## Chương trình và tiến trình

Chúng ta sẽ đi tiếp với một khái niệm quan trọng của hệ điều hành – chương trình và tiến trình.

+ Chương trình: Là các file binary được build từ source code. Chúng nằm trên ổ cứng và không tương tác cũng như sử dụng bất cứ một tài nguyên nào của hệ thống. Do đó cho dù hệ thống có lưu trữ bao nhiêu chương trình thì hiệu năng của nó cũng sẽ không bị giảm đi.

+ Tiến trình: Là những chương trình đã được load vào hệ thống. Do đó chúng sẽ tương tác và sử dụng tài nguyên của hệ thống. Càng nhiều tiến trình chạy trong hệ thống thì hiệu năng sẽ càng bị giảm đi.

Cũng giống như việc đặt tên để định danh cho con người, hệ điều hành sẽ đánh số cho từng tiến trình để định danh chúng. Số định danh đó sẽ là số thứ tự mà tiến trình đó được load vào hệ thống. Chúng được gọi là các process id. Hệ thống sẽ tương tác với các tiến trình thông qua định danh của chúng – process id.

Input và output của tiến trình: Chúng là 2 file với file đầu là nơi tiến trình sẽ đọc dữ liệu đầu vào cho các hàm như scanf() và file thứ 2 sẽ là nơi tiến trình ghi kết quả đầu ra trong các hàm như printf(). Thông thường file input sẽ là bàn phím và file output sẽ là màn hình console.

## Unix là một hệ điều hành đa nhiệm

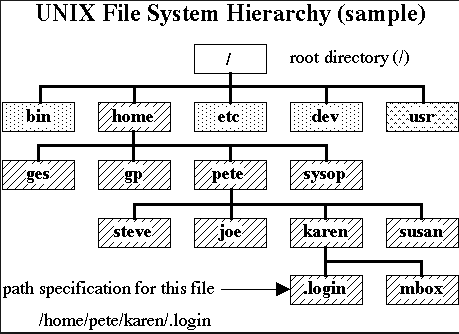
Khả năng đa nhiệm của hệ điều hành Unix nhằm mục đích tạo cho người dùng cảm giác hệ thống đang xử lý nhiều công việc một lúc, đây cũng là một tính năng làm nên tên tuổi của Unix so với những đàn anh đi trước. Đối với con người 1 2ms là rất ngắn và không thể cảm nhận được, tuy nhiên đối với máy tính thì khoảng thời gian đó đủ để làm nhiều công việc khác. Do vậy hệ thống liên tục chuyển đổi giữa các công việc khác nhau nhưng vẫn đảm bảo khả năng xử lý tức thì cho người dung.

Lấy ví dụ một người dùng đang vừa soạn thảo văn bản vừa nghe nhạc. Chương trình soạn thảo văn bản cần 1 ms để xử lý mỗi khi người dùng gõ 1 phím bất kỳ. Chương trình nghe nhạc cần chạy định kỳ 1ms mỗi 1s. Nếu hệ thống không chuyển đổi giữa các task 1 cách hợp lý thì đôi khi chương trình nghe nhạc có thể bị gián đoạn do tại thời điểm đó CPU đang chạy chương trình soạn thảo văn bản. Tuy nhiên hệ điều hành sẽ không để cho điều đó xảy ra. Nó sẽ ưu tiên cho chương trình nghe nhạc được chạy đúng thời điểm, nếu tại thời điểm đó user ấn 1 phím bất kỳ, nó sẽ không xử lý luôn mà sẽ delay 1 2ms để chương trình nghe nhạc chạy xong rồi mới xử lý việc đó. Tuy nhiên việc delay 1 2ms trong soạn thảo văn bản sẽ không gây cảm giác xử lý trễ đối với người dùng và anh ta sẽ có giảm giác hệ thống chạy đồng thời cả chương trình nghe nhạc và soạn thảo văn bản.

## Hệ thống file system

Các hệ điều hành trước Unix đã có hệ thống file system. Tuy nhiên đến lượt mình thì Unix nâng cấp chúng thêm một bậc nữa. Hệ thống sẽ coi toàn bộ các đối tượng tồn tại trong nó đều là file. Các đối tượng đó có thể là các hardware device, process, user… Từ đó hệ thống có thể quản lý tất cả các đối tượng thông qua một phương thức duy nhất đó là tương tác qua file.

Trong Unix file có thể hiểu như một định danh vì nhiều khi nó đại diện cho dữ liệu nằm trên ổ cứng hoặc một thiết bị nào đó. Ví dụ như các file đại diện cho từng process chúng nằm trong thư mục /proc/process\_id. Mỗi một file sẽ có các thuộc tính ví dụ như kích thước, quyền sở hữu, ngày chỉnh sửa … Ngoài ra có 1 file dạng đặc biệt đó là thư mục. Thư mục là một file những dữ liệu bên trong nó chính là danh sách tên của các file nằm trong nó.  
 Việc tổ chức các file vào trong các thư mục và tạo các thư mục con trong thư mục cha nhằm phân cấp và sắp xếp hệ thống file người ta còn gọi chúng với cái tên cây thư mục. Cây thư mục có các node lá là file, node cành là các thư mục vào node gốc là thư mục root của hệ thống.



Hình 4: Cấu trúc của một cây thư mục

## Bài tâp

Bài 1: Hướng dẫn build chương trình trong Linux dùng gcc

Trong Ubuntu command line chúng ta dùng câu lệnh

Ví dụ: để build và chạy một chương trình hello.c đơn giản trêm command line của Ubuntu:

#include <stdio.h>

int main()

{

printf("Hello, World!");

return 0;

}

Bài 2: Ví dụ về in một file ra cửa sổ console – readfile.c:

#include <stdio.h>

#include <stdlib.h> // For exit()

int main()

{

FILE \*fptr;

char filename = "/etc/passwd";

char c;

// Open file

fptr = fopen(filename, "r");

if (fptr == NULL)

{

printf("Cannot open file \n");

exit(0);

}

// Read contents from file

c = fgetc(fptr);

while (c != EOF)

{

printf("%c", c);

c = fgetc(fptr);

}

fclose(fptr);

return 0;

}

Build và chạy chương trình bằng command line:

Bài 3: Chương trình in ra tên của process từ process id nhập từ bàn phím:

#include <stdio.h>

#include <stdlib.h> // For exit()

int main()

{

FILE \*fptr;

char processpath[100], c;

int id;

printf("Xin hay nhap process id = ");

scanf("%d", &id);

sprintf(processpath, "/proc/%d/cmdline", id);

fptr = fopen(processpath, "r");

if (fptr == NULL)

{

printf("Khong co process id = %d \n", id);

exit(0);

}

c = fgetc(fptr);

while (c != EOF)

{

printf("%c", c);

c = fgetc(fptr);

}

fclose(fptr);

return 0;

}

Bài 4: Viết 1 chương trình có khả năng hiện thị nội dung của 1 file lên console giống với lệnh cat. Đặt tên nó là MyCat và copy vào thư mục bin. Sau đó sử dụng nó giống như lệnh cat của hệ thống.

Bài 5: Viết 1 chương trình C có khả năng in ra tọa độ của chuột trên máy tính. Ví dụ: X/Y. Trong đó X, Y là hoàn độ và tung độ của con trỏ chuột trên màn hình.

# FILE TRONG LINUX

## Giới thiệu

Trong chương này chúng ta sẽ tìm hiểu chức năng của một tệp tin, bao gồm mở một tệp tin, đọc tệp tin, ghi tệp tin, v.v.. Đọc ghi tệp tin trong Linux hầu như chỉ dùng năm hàm: open, read, write, lseek, và close.

## Các loại tệp tin

Hầu hết các tệp trong Linux là tệp tin thường hoặc tệp tin thưc mục, ngoài ra còn một số loại tệp tin đặc biệt khác. Dưới đây liệt kê tất cả các loại tệp tin trong Linux:

1. Tệp tin thường (regular file). Là loại tệp tin phổ biến nhất trong Linux, nó chứa dữ liệu ở một định dạng nào đó. Kernel không phân biệt nếu dữ liệu này là text hay binary. Nội dung dữ liệu của một tệp tin thường sẽ do các ứng dụng trên tầng user sử lý. (Ngoại trừ duy nhất một loại tệp tin là tệp tin có thể thực thi. Để thực thi một tệp tin thì kernel phải hiểu được cấu trúc của nó. Tất cả các tệp tin có thể thực sẽ theo một chuẩn nào đó để kernel xác định được đâu là vùng text và vùng data của chương trình).
2. Tệp tin thư mục (directory file). Là loại tệp tin chứa tên của các các tệp tin khác cùng với các con trỏ tới thông tin của các tệp tin này. Bất kì process nếu có quyển đọc một thư mục nào đó thì nó cũng có thể đọc được nội dung dữ liệu của tệp tin thưc mục, nhưng chỉ có kernel mới có có thể ghi trực tiếp vào tệp tin thưc mục. Các process muốn thay đổi dữ liệu của tệp tin thưc mục phải dùng một số hàm đặc biệt.
3. Các loại file đặc biệt khác như block file, character file, FIFO, Socket, symbolic link. Chúng là những tệp tin không nằm trên ổ cứng:
   1. Tệp tin block (block special file). Là một loại tệp tin cho phép đọc dữ liệu từ các thiết bị theo dạng các buffer có kích thước cố định. Ví dụ như ổ đĩa
   2. Tệp tin character (character special file). Là loại tệp tin cho phép đọc dữ liệu từ các thiết bị mà không bị ép buộc về kích thước cố định của mỗi lần đọc. Tất các các thiết bị phần cứng trong hệ thống đều là một tệp tin block hoặc character.
   3. FIFO. Là một loại tệp tin dùng trong giao tiếp giữa các process với nhau.
   4. Socket. Là một loại tệp tin dùng trong giao tiếp mạng giữa các process với nhau.
   5. Tệp tin liên kết (symbolic link). Là một loại tệp tin trỏ đến một tệp tin khác.

## File descriptors

Đối với kernel thì tất cả các tệp tin đã mở được xác định bởi file descriptor. File descriptor là một số nguyên không âm. Khi chúng ta mở hoặc tạo một tệp tin mới, kernel sẽ trả về một file descriptor cho process. Khi đọc hoặc ghi một file, chúng ta xác định file bằng file descriptor được trả về từ hàm open hoặc create, và dùng file descriptor này như là một tham số truyền vào hàm read hoặc write.

Theo quy ước, Các chương trình thường liên kết file descriptor 0 với thiết bị nhập chuẩn của một tiến trình (process), file descriptor 1 với thiết bị xuất chuẩn, và file descriptor 2 với thiết bị xuất lỗi chuẩn. Quy ước này được sử dụng bởi shell và nhiều chương trình khác nhau; nó không phải là một tính năng của kernel. Tuy nhiên, nếu một chương trình không theo chuẩn này có thể gây phát sinh lỗi khi chạy.

Những giá trị file descriptor trên được chuẩn hóa trong POSIX.1, những số magic 0, 1, và 2 được thay thế bằng những hằng số tượng trưng STDIN\_FILENO, STDOUT\_FILENO, và STDERR\_FILENO để giúp việc đọc code được dễ dàng hơn. Các hằng số này được định nghĩa trong thư viện <unistd.h>.

## Các hàm cơ bản để thao tác với file

### Hàm open

Một file được mở hoặc tạo bằng cách giọi ra một trong hai hàm open.



Tham số cuối cùng là … để chỉ ra rằng số lượng và kiểu của các tham số còn lại có thể thay đổi. Với hàm trên, tham số cuối cùng chỉ được sử dụng khi tạo một file mới.

Tham số path là tên của file cần được mở hoặc tạo. Hai hàm này có vô số tùy chọn được xác định bởi tham số oflag. oflag được tạo bởi một hoặc hơp của nhiều hằng số được định nghĩ trong thư viện <fcntl.h>:

|  |  |
| --- | --- |
| O\_RDONLY | Mở để chỉ đọc |
| O\_WRONLY | Mở để chỉ ghi |
| O\_RDWR | Mở để đọc và ghi |
|  | Để tương thích với các chương trình cũ thì linux định nghĩa O\_RDONLY là 0, O\_WRONLY là 1, và O\_RDWR là 2. |
| O\_EXEC | Mở để chỉ thực thi |
| O\_SEARCH | Mở để chỉ tìm kiếm (dùng cho thử mục) |

Một và chỉ một trong năm hằng số bên trên là cần thiết phải đưa ra. Các hằng số tiếp sau đây là tùy chọn:

|  |  |
| --- | --- |
| O\_APPEND | Ghi vào cuối tệp tin |
| O\_CLOEXEC | Bật FD\_CLOEXEC trong cờ file descriptor |
| O\_CREAT | Tạo một file nếu nó chưa tồn tại. Tùy chọn này yêu cầu truyền vào tham số thứ ba trong hàm open (tham số thứ tư trong hàm openat) |
| O\_DIRECTORY | Trả về lỗi nếu tham số path không phải là thư mục |
| O\_EXCL | Trả về lỗi nếu dùng chung với tùy chọn O\_CREAT và file đã tồn tại. Nó dùng để kiểm tra sự tồn tại của một tệp tin. |
| O\_NOCTTY |  |
| O\_NOFOLLOW | Trả về lỗi nếu tham số path là symbolic link. |
| O\_NONBLOCK | Nếu path là một FIFO, một loại tệp tin block hoặc character đặc biệt thì tùy chọn này sẽ cho phép mở và đọc ghi file không bị block |
| O\_SYNC | Mỗi quá trình đọc ghi sẽ chờ cho dữ liệu được đồng bộ phần cứng. |
| O\_TRUNC | Nếu một file đã tồn tài và mở thành công thì xóa toàn bộ dữ liệu của tệp tin |
| O\_TTY\_INIT | Dùng khi mở một thiết bị nhập xuất |

File descriptor trả về bởi hàm open luân luân là số nhỏ nhất chưa được sử dụng.

### Hàm close

Để đóng một file chúng ta dùng hàm close

Đóng một tệp tin sẽ đồng thời giải phóng tất cả các khóa (record locks) mà một tiến trình sử dụng trên tệp tin. Khi một tiến trình bị đóng, tất cả các tệp tin mở bởi tiến trình sẽ bị đóng một cách tự động bởi kernel. Chình vì thế mà thực tế nhiều chương trình không trực tiếp đóng tệp tin khi sử dụng xong.

### Hàm lseek

Mỗi tệp tin đều có một giá trị offset, thường là một số nguyên không âm, dùng để tính số byte tính từ đầu tệp tin đến vị trí hiện tại của đầu đọc. Các hàm đọc ghi thường bắt đầu từ vị trí offset này của tệp tin và làm cho giá trị offset tăng lên bằng số số byte được đọc hoặc ghi. Mặc định, giá trị của offset được khởi tạo là 0 khi một tệp tin được mở, trừ khi chúng ta dùng tùy chọn O\_APPEND.

Giá trị offset của tệp tin có thể được gán trực tiếp bằng hàm lseek

  
Tham số whence để chỉ điểm xuất phát để tính offset. Thông thường nó được gán với 1 trong 3 macro: SEEK\_SET, SEEK\_CUR, SEEK\_END. Trong đó:

* Nếu whence là SEEK\_SET, giá trị offset của tệp tin sẽ là offs tính từ đầu tệp tin.
* Nếu whence là SEEK\_CUR, giá trị offset của tệp tin sẽ là giá trị hiện tại của nó cộng thêm offs. offs có thể là số dương hoặc âm.
* Nếu whence là SEEK\_END, giá trị offset của tệp tin sẽ là kích thước của tệp tin cộng thêm offs. offs có thể là số dương hoặc âm.

Vì giá trị trả về của hàm lseek là giá trị offset mới của tệp tin, chúng ta có thể tính được giá trị offset hiện tại của tệp tin bằng cách dịch chuyển 0 byte từ vị trí hiên tại.

off\_t currpos;

currpos = lseek(fd, 0, SEEK\_CUR);

Ví dụ: Chương trình dưới đọc một file từ vị trí bất kì

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <fcntl.h>

#include <stdlib.h>

int main(int argc, char \*\*argv)

{

int offset, file, fsize;

char buff[1024];

if (argc < 3)

{

printf("Cach nhap tham so: ./a.out <file> <offset>\n");

return -1;

}

file = open(argv[1], O\_RDONLY);

if (file == -1)

{

printf("Khong the mo file!\n");

return -1;

}

offset = atoi(argv[2]);

if (offset < 0)

{

printf("Gia tri offset khong hop le!");

return -1;

}

fsize = (int)lseek(file, offset, SEEK\_SET);

while (1)

{

fsize = (int)read(file, buff, 1024);

if (fsize > 0)

{

if (fsize < 1024)

buff[fsize] = 0;

printf("%s", buff);

}

else

{

printf("\n");

return 0;

}

}

return 0;

}

Khi chạy chương trình trên chúng ta sẽ thu được kết quả:



### Hàm read

Dữ liệu trong một tệp tin đã mở được đọc ra bằng hàm read.

Nếu đọc thành công, hàm trả về số byte đọc được. Nếu đọc đến cuối tệp tin hàm trả về 0. Trong một số trường hợp số byte đọc được sẽ ít hơn số byte yêu cầu:

* Khi đọc một tệp tin bình thường, nếu đọc đến cuối tệp tin mà chưa đủ số byte yêu cầu. Ví dụ: nếu một tệp tin có 30 byte mà ta yêu cầu đọc 100 byte thì giá trị trả về sẽ là 30. Trong lần đọc tiếp theo, giá trị trả về sẽ là 0.

Hàm đọc bắt đầu từ vị trí offset của tệp tin. Trước khi hàm đọc trả về thành công thì giá trị offset sẽ được tăng lên một khoảng bằng số byte đọc được.

### Hàm write

Dữ liệu được ghi vào tệp tin đã mở bằng hàm read

Giá trị trả về luân bằng tham số truyền vào nbytes; nếu không đúng thì tức là đã xảy ra lỗi. Lỗi sinh ra khi dùng hàm write thường là do ổ đĩa đã bị đầy hoặc đã đạt tới giới hạn bộ nhớ của một process.

Với một tệp tin thông thường, việc ghi sẽ bắt đầu từ vị trí offset hiện tại của tệp tin. Nếu ta khai báo tùy chọn O\_APPEND khi mở tệp tin thì offset sẽ ở vị trí cuối tệp tin mỗi lần ghi. Sau khi ghi thành công, giá trị offset sẽ tăng lên một khoảng bằng số byte đã được ghi.

### Hàm ioctl

ioctl là một hàm đa chức năng. Tất cả những gì không làm được khi sử dụng các hàm trên thì thường sẽ làm được khi dùng hàm ioctl. Thiết bị vào ra chuẩn là ví dụ điển hình về việc sử dụng hàm ioctl. Để xử lý được hàm ioctl thì trong driver của thiết bị phải khai báo hàm handle.

ioctl là một system call dùng để điều khiển các thiết bị bằng cách truyền dữ liệu xuống driver của thiết bị đó. Tham số đầu tiên fd là file descriptor của thiết bị cần điều khiển, nó được trả về từ hàm open. Tham số thứ hai là một số nguyên, nó là mã lệnh điều khiển thiết bị. Dấu ba chấm cuối cùng để chỉ ra rằng có thể còn nhiều tham số nữa. Thông thường thì ta chỉ sử dụng thêm một tham số, nó thường là con trỏ trỏ đến biến hoặc struct.

Trong kernel driver, hàm sử lý cho ioctl sẽ giống như sau:

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/version.h>

#include <linux/fs.h>

#include <linux/errno.h>

#include <linux/device.h>

#include <linux/uaccess.h>

#include <linux/init.h>

#include <linux/printk.h>

#include "mystruct.h"

#define DEVICE\_NAME "myioctl"

#define CLASS\_NAME "myioctl\_device"

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Truong Van Huy");

static struct class \*myioctl\_class;

static struct device \*myioctl\_device;

static int major\_number;

static int status = 1, dignity = 3, ego = 5;

static long my\_ioctl(struct file \*i, unsigned int cmd, unsigned long arg)

{

my\_variables v;

pr\_info("%s: func called, cmd = %u", \_\_func\_\_, cmd);

switch (cmd) {

case QUERY\_GET\_VARIABLES:

v.status = status;

v.dignity = dignity;

v.ego = ego;

if (copy\_to\_user((my\_variables \*)arg, &v,

sizeof(my\_variables)))

return -EACCES;

break;

case QUERY\_CLR\_VARIABLES:

status = 0;

dignity = 0;

ego = 0;

break;

case QUERY\_SET\_VARIABLES:

if (copy\_from\_user(&v, (my\_variables \*)arg,

sizeof(my\_variables)))

return -EACCES;

status = v.status;

dignity = v.dignity;

ego = v.ego;

break;

default:

return -EINVAL;

}

return 0;

}

const struct file\_operations query\_fops = {

.owner = THIS\_MODULE,

.unlocked\_ioctl = my\_ioctl

};

static int \_\_init myioctl\_init(void)

{

major\_number = register\_chrdev(0, DEVICE\_NAME, &query\_fops);

if (major\_number < 0) {

pr\_info("%s: Khong cap phat duoc major number\n", \_\_func\_\_);

return major\_number;

}

myioctl\_class = class\_create(THIS\_MODULE, CLASS\_NAME);

if (IS\_ERR(myioctl\_class)) {

pr\_info("%s: Khong dang ky duoc device classs\n", \_\_func\_\_);

return PTR\_ERR(myioctl\_class);

}

myioctl\_device = device\_create(myioctl\_class, NULL,

MKDEV(major\_number, 0), NULL, DEVICE\_NAME);

if (IS\_ERR(myioctl\_device)) {

pr\_info("%s: Khong tao duoc device\n", \_\_func\_\_);

class\_destroy(myioctl\_class);

unregister\_chrdev(major\_number, DEVICE\_NAME);

return PTR\_ERR(myioctl\_device);

}

pr\_info("%s: Module created\n", \_\_func\_\_);

return 0;

}

static void \_\_exit myioctl\_exit(void)

{

device\_destroy(myioctl\_class, MKDEV(major\_number, 0));

class\_unregister(myioctl\_class);

class\_destroy(myioctl\_class);

unregister\_chrdev(major\_number, DEVICE\_NAME);

pr\_info("%s: driver exit\n", \_\_func\_\_);

}

module\_init(myioctl\_init);

module\_exit(myioctl\_exit);

Bên trên là hàm ioctl handle của một driver cho phép đọc và ghi một struct đơn giản, ta không thể thực hiện được việc này khi dùng hàm read hoặc write thông thường. Dưới đây là ví dụ về việc xử lý ioctl trên tầng user:

#include <stdio.h>

#include <sys/ioctl.h>

#include <sys/types.h>

#include <fcntl.h>

#include <string.h>

#include "mystruct.h"

char \*filename = "/dev/myioctl";

int fd;

void get\_vars(int fd)

{

my\_variables v;

printf("\n\nioctl read variables\n");

if (ioctl(fd, QUERY\_GET\_VARIABLES, &v) == -1) {

perror("get\_vars ioctl get");

}

else {

printf("Status\t: %d\n", v.status);

printf("Dignity\t: %d\n", v.dignity);

printf("Ego\t: %d\n", v.ego);

}

}

void set\_vars(int fd)

{

int t;

my\_variables v;

printf("Enter Status\t: ");

scanf("%d", &t);

getchar();

v.status = t;

printf("Enter Dignity\t: ");

scanf("%d", &t);

getchar();

v.dignity = t;

printf("Enter Ego\t: ");

scanf("%d", &t);

getchar();

v.ego = t;

if (ioctl(fd, QUERY\_SET\_VARIABLES, &v) == -1)

perror("set\_vars ioctl set");

}

int main(void)

{

fd = open(filename, O\_RDWR);

if (fd == -1) {

perror("main open");

return -1;

}

set\_vars(fd);

get\_vars(fd);

return 0;

}

## Thư mục /dev

/dev là nơi chứa các tệp tin đặc biệt hoặc các device file. Trong Linux, tất cả mọi thứ đều được mô tả bằng file. Nhìn vào các file và thự mục trong đường dẫn này chúng ta sẽ thấy có các file sda1, sda2, … chúng đại diện cho các phân vùng trên ổ đĩa đầu tiên của chúng ta. /dev/cdrom và /dev/fd0 đại diện cho ổ đĩa CD và ổ đĩa mềm. Điều này có vẻ lạ, nhưng khi chú ý kỹ ta sẽ nhận thấy thuộc tính của một file sẽ giống với một thiết bị là có thể đọc ghi. Chúng ta có thể open, read và write xuống một device file như các file bình thường, nhưng dữ liệu này sẽ được chuyển xuống driver của device đó. Ví dụ , khi ta đọc ghi dữ liệu vào file /dev/ttyS0 chính là bạn đang giao tiếp với một device có tên ttyS0.

Phần lớn các device là block hoặc character device. Thông thường block device là thiết bị lưu trữ dữ liệu, còn character device là thiết bị có thể gửi và truyền dữ liệu. Ví dụ như ổ đĩa mềm, ổ đĩa cứng, CD là các block device, còn cổng giao tiếp serial, chuột, cổng máy in là các character devie. Để có thể hiện device file trong thư mục dev và giao tiếp với nó thì linux phải được cài đặt driver của device này trước đó.

Ví dụ: Chương trình đọc vị trí con trỏ chuột trên màn hình từ device file

#include <stdio.h>

#include <sys/time.h>

#include <unistd.h>

#include <stdlib.h>

#include <fcntl.h>

#include <stdlib.h>

struct input\_event {

struct timeval time;

unsigned short type;

unsigned short code;

unsigned int value;

};

void main(void)

{

int file, fsize;

char buff[128];

struct input\_event \*temp;

unsigned int x, y;

x = y = 0;

file = open("/dev/input/event6", O\_RDONLY);

while (1)

{

fsize = (int)read(file, buff, 1024);

if (fsize > 0)

{

temp = (struct input\_event \*)buff;

if (temp->code == 0)

x = temp->value;

else

y = temp->value;

}

printf("x=%u y=%u\n", x, y);

}

}

Khi chạy trương trình ta sẽ thu được kết quả như sau:



# CÁCH QUẢN LÝ HỆ THỐNG FILE TRONG LINUX

## Giới thiệu

Trong bài trước chúng ta đã tìm hiều về các loại file và các hàm đọc ghi dùng cho các file thông thường – như mở, đọc hoặc ghi một file. Trong chương này, chúng ta sẽ tiếp tục tìm hiểu thêm các tính năng nâng cao hơn của hệ thống file system và các thuộc tính của một file, thư mục. Chúng ta bắt đầu bằng việc tìm hiểu về cấu trúc của cây thư mục trong Linux, tiếp theo là hàm stat và các thuộc tính của cấu trúc stat, ý nghĩa của từng thuộc tính của một file trong cấu trúc stat. Tiếp theo chúng ta sẽ tìm hiểu về các hàm dùng để thay đổi các thuộc tính này của file và thư mục.

## Giới thiệu về cây thư mục trong Linux

Hệ thống tệp tin (file system) là tập hợp tất cả các file có liên kết với nhau và nằm trên cùng một phân vùng trong ổ đĩa. Một phân vùng là nơi chứa dữ liệu và có kích thước có thể mở rộng bằng kích thước ổ đĩa cứng. Ổ đĩa cứng có thể có nhiều phân vùng và thường chỉ chứa một cây thư mục duy nhất.

Linux sử dụng cây thư mục có cấu trúc kế thừa, giống như một cái cây lộn ngược, điểm bắt đầu của cây thư mục là root (/) và tất cả các thư mục khác là thư mục khác đều bắt đầu từ root. Cây thư mục trong Linux là tập hợp các file và thư mục có tính chất sau:

* Nó có thư mục gốc root (/) chứa tất cả các file và thư mục khác.
* Mỗi file hoặc thư mục trong cùng một thư mục mẹ phải có tên khác nhau.
* Các cây thư mục trên cùng một ổ đĩa thì độc lập với nhau, chúng không có bất kì liên hệ nào.

Mục đích của thư mục là để chứa các tệp tin cùng loại với nhau để dễ dàng tìm kiếm. Dưới đây là các thư mục phổ biến trong Linux:

1. **/** Thư mục gốc root chỉ lên chứa các thư mục quan trong trong của hệ thống
2. **/bin** nơi đây lưu giữ những tập lệnh cơ bản của linux, những tập lệnh này có thể dược chạy bởi bất kỳ user nào trên hệ thống (không phải như là việc chỉ mỗi root có thể chạy trong /sbin). Các chương trình khác có thể được tìm thấy trong /usr/bin.
3. **/etc** trong thư mục này chứa các tập tin tuỳ biến của cả hệ thống. Những tập tin trong này điều khiển cả quá trình khởi động máy, quản lí users, quản lí mạng ...
4. **/lib** Nơi chứa đựng một vài thư viện dùng chung hoặc là đường dẫn tượng trưng (symbolic links) đến các thư mục dùng chung đó. Những thư viện này sẽ được sử dụng đến cho việc chạy một số các programs nhất định. Trong thư mục /lib /modules chứa đựng những kernel modules, chúng được bật và tắt nếu cần thiết.
5. **/boot** Nơi chứa những thông tin bootmanager cần có (thông dụng hiện nay là grub) và một số bản của kernel.
6. **/home** Trong thư mục này chứa đựng những home directories của người dùng.
7. **/mnt** để mount các hệ thống tệp tin phụ như CD-Rom, ổ đĩa mềm.
8. **/proc** Những thư mục con trong thư mục này chứa đựng những tiến trình đang được chạy trên hệ thống, các tệp tin trong thư mục này không phải là các tệp tin thực sự.
9. **/tmp** Chứa các tệp tin tạm thời.
10. **/usr** Thư mục này được dùng với nhiều mục đích khác nhau bởi nhiều người dùng.
11. **/var** Thường chứa các file có kích thước thay đổi, như các file log có chứa nhiều thông tin khác nhau.

## Các thuộc tính của file

Trọng tâm của bài này sẽ xoay quanh bốn hàm stat và dữ liệu mà chúng trả về:

Khi chúng ta cung cấp đường dẫn pathname, thì hàm stat sẽ trả về một cấu trúc chứa thông tin về file đó. Hàm fstat thì trả về thông tin của một file đã mở có file descriptor là fd. Hàm lstat thì tương tự như hàm stat, nhưng đường dẫn là một symbolic link, hàm lstat trả về thông tin của symbolic link chứ không phải thông tin của file mà nó trỏ đến.

Hàm fstatat cho phép chúng ta lấy thông tin của file nằm trong thư mục đã được mở có file descriptor là fd. Tham số flag dùng để xác định xem hàm này có tìm kiếm trong symbolic link hay không.

Tham số buf của bốn hàm trên là một con trỏ trỏ đến cấu trúc stat mà chúng ta đã cấp phát trước đó. Các hàm stat sẽ điền thông tin vào các trường của cấu trúc stat này. Tùy từng phiên bản Linux khác nhau mà cấu trúc stat có thể khác nhau đôi chút, nhưng về cơ bản thì cấu trúc stat sẽ giống như sau:



Cấu trúc timespec định nghĩa thời gian theo giây hoặc nano giây. Nó gồm ít nhất hai trường:

time\_t tv\_sec;

long tv\_nsec;

Câu lệnh ls –l là ví dụ điển hình về việc sử dụng hàm stat để kiểm tra thông tin của file.

Ví dụ dưới đây trả về mode, user id, group id, và kích thước của một file bất kì:

#include <fcntl.h>

#include <stdio.h>

#include <sys/stat.h>

int main(int argc, char \*argv[])

{

struct stat st;

if (argc != 2)

printf("usage: a.out <pathname>");

if (!stat(argv[1], &st))

{

printf("File information:\n");

printf("Mode: %u\n", (unsigned int)st.st\_mode);

printf("User id: %u\n", (unsigned int)st.st\_uid);

printf("Group id: %u\n", (unsigned int)st.st\_gid);

printf("File size: %u\n", (unsigned int)st.st\_size);

}

else

printf("Cannot access file %s\n", argv[1]);

return 0;

}

Khi chạy chương trình trên ta sẽ thu được kết quả như sau:



## Quyền truy cập file

Thuộc tính st\_mode chứa các bit dùng để xác định quyền truy cập của file – quyền truy cập ở đây là quyền đọc, quyền ghi và quyền thực thi của một file. Các file ở đây bao gồm toàn bộ các file được miêu tả trong bài trước: các file thông thường, file thư mục và các file đặc biệt. Tất cả các file này đều có quyền truy cập. Có chín bit quyền truy cập khác nhau của một file, và nó được chia ra làm ba loại:

|  |  |
| --- | --- |
| st\_mode mask | Ý nghĩa |
| S\_IRUSR  S\_IWUSR  S\_IXUSR | user-read user-write user-execute |
| S\_IRGRP  S\_IWGRP  S\_IXGRP | group-read group-write group-execute |
| S\_IROTH  S\_IWOTH  S\_IXOTH | other-read other-write other-execute |

Bảng 3. 1: Chín bit quyền truy cập được định nghĩa trong <sys/stat.h>

Từ user ở ba hàng đầu tiên dùng để chỉ chủ sở hữu của file. Group dùng để chỉ người người trong cùng group với chủ sở hữu, other dùng để chỉ những người còn lại. Câu lệnh chmod thường được dùng để thay đổi chín bit quyền truy cập này.

Mỗi khi user mở một file, kernel sẽ tiến hành kiểm qua quyền truy cập của user đối với file đó thông qua việc so sánh user-id và group-id. Quá trình kiểm tra quyền truy cập trong kernel diễn ra như sau:

1. Nếu user-id của process là 0 (tức là supperuser), thì được phép truy cập. Nó cho phép superuser tự do truy cập bất kì file nào trong hệ thống.
2. Nếu user-id của process trùng với user-id của file, thì cho phép truy cập file nếu các bit user- (trong hình 1) được bật. Nếu các bit user- không được bật thì từ chối quyền truy cập file. Thích hợp ở đây có nghĩa là, nếu process muốn mở file để đọc thì user-read bit phải được bật. Nếu process muốn mở file để ghi thì user-write bit phải đươc bật. Nếu process muốn mở file để thực thi thì user-execute bit phải được bật.
3. Nếu group ID của process trùng với group ID của file, thì quyền truy cập được xác định bằng các bit truy cập thích hợp trong group- bit (hình 1). Nếu bit này không bật thì từ chối quyền truy cập.
4. Các trường hợp còn lại xác định thông qua other- bit (hình 1), nếu bit tương ứng được bật thì cho phép truy cập. Ngược lại thì từ chối truy cập.

Bốn bước trên được thực hiện một cách tuần tự.

## Hàm chmod, fchmod và fchmodat

Các hàm chmod, fchmod và fchmodat cho phép chúng ta thay đổi quyền truy cập của một file.



Hàm chmod tác động đến một file bất kì có đường dẫn là pathname, còn hàm fchmod thì tác động đến một file đã được mở có file descriptor là fd. Để thay đổi quyền truy cập của file thì process chạy câu lệnh trên phải có user-id trùng với chủ sở hữu của file hoặc process phải có quyền superuser.

Tham số mode được xác định bằng phép hoặc trên bit của các hằng số trong bảng sau:

|  |  |
| --- | --- |
| *mode* | Description |
| S\_ISUID  S\_ISGID  S\_ISVTX | set-user-ID on execution set-group-ID on execution saved-text (sticky bit) |
| S\_IRWXU  S\_IRUSR  S\_IWUSR  S\_IXUSR | read, write, and execute by user (owner) read by user (owner) write by user (owner) execute by user (owner) |
| S\_IRWXG  S\_IRGRP  S\_IWGRP  S\_IXGRP | read, write, and execute by group read by group write by group execute by group |
| S\_IRWXO  S\_IROTH  S\_IWOTH  S\_IXOTH | read, write, and execute by other (world) read by other (world) write by other (world) execute by other (world) |

Bảng 3. 2: Tham số mode truyền vào của hàm chmod được định nghĩa trong <sys/stat.h>

## Quyền sở hữu của file và thưc mục mới

Khi người dùng tạo ra một file hoặc thư mục mới thì thông thường user-id của file/thư mục mới sẽ được gán bằng user-id của process tạo thư mục đó. POSIX.1 cho phép chọn một trong các cách sau để xác định group ID của một file mới:

1. Group-id của file mới có thể là groud-id của process.
2. Group-id của file mới có thể là group-id của thưc mục chứa file được tạo đó

Dùng cách thứ hai – kế thừa group-id của thư mục mẹ - đảm bảo rằng tất cả các file và thực mục được tạo ra trong một thư mục sẽ có chung group ID như thư mục mẹ. Các này được dùng phổ biến trong Linux, ví dụ như thư mục /var/mail

## Hàm access và faccessat

Như chúng ta đã biết ở trên, khi mở một file, kernel sẽ tiến hành kiểm tra quyền truy cập dựa vào user-id và group-id của process. Tuy nhiên, đôi khi trong chương trình chúng ta muốn kiểm tra quyền truy cập của một file trước khi mở nó. Hàm access và faccessat cho phép chúng ta kiểm tra quyền truy cập của một file bất kì:



Tham số truyền vào mode là cờ dùng để xác định xem kiểu kiểm tra của hàm là kiểm gì. Cờ mode có thể là F\_OK dùng để kiểm tra xem file có tồn tại hay không hoặc bất kì cờ nào trong các cờ sau, chúng ta có thể dung phép OR để kết hợp nhiều cờ với nhau.

|  |  |
| --- | --- |
| *mode* | Ý nghĩa |
| R\_OK  W\_OK  X\_OK | Kiểm tra quyền đọc Kiểm tra quyền ghi Kiểm tra quyền thực thi |

Bảng 3. 3: Cờ mode xác định cách kiểm tra của hàm access

Ví dụ dưới đây dùng để kiểm tra quyền truy cập của một file bất kỳ

#include <unistd.h>

#include <fcntl.h>

#include <stdio.h>

int main(int argc, char \*argv[])

{

if (argc != 2)

printf("usage: a.out <pathname>");

if (access(argv[1], R\_OK) < 0)

printf("access error for %s\n", argv[1]);

else

printf("read access OK\n");

if (open(argv[1], O\_RDONLY) < 0)

printf("open error for %s\n", argv[1]);

else

printf("open for reading OK\n");

return 0;

}

Khi chạy chương trình chúng ta sẽ thu được kết quả như sau:



## Hệ thống file trong Linux

Trong phần này chúng ta sẽ tìm hiểu kỹ hơn về cách xây dựng hệ thống file trong Linux. Hiểu được sự khác nhau giữa i-node và cách directory entry trỏ đến i-node.

Chúng ta đã biết rằng, một ổ đĩa cứng được chia ra làm nhiều phân vùng (partitions) khác nhau. Mỗi phân vùng có thể chữa một hệ thống file, như trong hình 4 dưới đây.

|  |
| --- |
|  |
| Hình 3. 1: Ổ đĩa, phân vùng, và hệ thống file |

Như chúng ta đã biết tất cả mọi thứ trong Linux đều được biểu diễn dươi dạng một file – từ các file thông thường, đến các thư mục hay các thiết bị phần cứng… đều là các file. Còn mỗi file đều được xác định bằng một inode. Vậy inode là gì?. inode là một cấu trúc dữ liệu, dùng cho các hệ thống file truyền thống như UFS hay ext3, nó bao gồm các trường mô tả thông tin về file như: Kiểu file, quyền truy cập, chủ sở hữu, kích thước, ngày tạo, ngày truy cập gần nhất, có lượng link đến file (soft link và hard link)… Hàm stat mà chúng ta tìm hiểu ở phần trên lấy thông tin từ inode của file rồi gán cho các trường của cấu trúc stat. Trong mỗi phân vùng bộ nhớ thường có một vùng để chứa các inode, tập hợp các inode này tạo thành một mảng các phần tử, số thứ tự của inode trong mảng này được gọi là chỉ số inode (inode index), như mô tả trên hình 4.

Mỗi i-inode sẽ chứa các thuộc tính và vị trí các khối dữ liệu của file trên ổ đĩa:

|  |
| --- |
|  |
| Hình 3. 2: Liên hệ giữa i-node và dữ liệu trong file |

Một khai niệm nữa mà chúng ta cần nắm rõ là directory entry. Directory entry cũng là một cấu trúc nhưng nó chỉ chứa chỉ số inode và tên file.

Từ hình 5 trên ta xác định được rằng:

* Hai directory entry có thể cùng trỏ đến một inode – tức là chúng có cùng chỉ số inode. Mỗi i-node có một trường dùng để đếm số link trỏ đến i-node này. Khi giá trị số đếm link này giảm về 0 thì file đó mới bị xóa (giải phóng toàn bộ các khối dữ liệu mà file trỏ đến). Chính vì thề mà việc xóa liên kết đến một file không phải lúc nào cũng là xóa các khối dữ liệu của file. Trong cấu trúc stat của file, số đếm link là thuộc tính của biến st\_nlink, nó có kiểu nlink\_t. Kiểu link này được gọi là hard link.
* Chỉ số i-node lưu trong directory entry phải trỏ đến inode trên cùng một hệ thống file, một directory entry không thể trỏ đến inode trên hệ thống file khác. Điều này giải thích tại sao chúng ta không thể tạo hard link của một file từ phân vùng này trên một phân vùng khác.
* i-node chứa tất cả các thông tin về file: kiểu file, các bit quyền truy cập file, kích thước file, các con trỏ trỏ đến khối dữ liệu của file…

## Symbolic Link

Symbolic link là một con trỏ gián tiếp đến một file, không giống với hard link được miêu tả bên trên – hard link trỏ trực tiếp đến i-node của file.

* Hard link thường yêu cầu link và file phải cùng nằm trên một file system.
* Chỉ có superuser mới có thể tạo một hard link đến một thư mục.

Không có giới hạn nào đối với symbolic link và nơi mà nó trỏ đến, tất cả mọi người dùng đều có quyền tạo symbolic link.

## Tạo và đọc symbolic link

Một symbolic link được tạo bằng hàm symlink hoặc symlinkat:

Một directory entry mới, sympathy, được tạo ra và trỏ tới actualpath. Hàm này không bắt buộc actualpath tồn tại khi tạo symbolic link.

Hàm symlinkat tương tự như hàm symlink, nhưng sympathy là đường dẫn tương đối tính từ thư mục có file descriptor fd.

Khi đọc symbolic link bằng hàm open thì nó sẽ đọc file mà symbolic link trỏ đến. Do đó, để đọc nội dung và tên của symbolic link – không phải của file mà nó trỏ đến – thì chúng ta dùng hàm readlink và readlinkat:

Hàm này gộp chức năng của ba hàm open, read và close. Nếu thành công hàm trả về số byte đọc được gửi trong buf. Nội dung của symbolic link không kết thúc bằng giá trị null.

## Thư mục

Trong bài trước chúng ta đã biết thư mục cũng là một file thông thường, nhưng dữ liệu của nó chứa tên của các file nằm trong nó. Trong phần này, chúng ta sẽ tìm hiểu cách tạo, xóa, mở và đọc một thư mục:

## Hàm mkdir, mkdirat và hàm rmdir

Thư mục được tạo bằng hàm mkdir và mkdirat, và được xòa bằng hàm rmdir.

Hai hàm trên sẽ tạo ra một thư mục rỗng mới, nó gồm hai thưc mục là chấm và chấm-chấm được tạo một cách tự động. Tham số quyền truy cập file, mode, sẽ bị thay đổi bởi mode của process.

Một sai lầm thường gặp là chúng ta đặt mode giống như cho file: chỉ có quyền đọc và ghi. Nhưng đồi với một thư mục, chúng ta cần ít nhất là bit quyền thực thì phải được bật, để truy cập đến các file trong thư mục này.

User Id và group ID của process của thư mục được xác định giống như trong phần quyền truy cập file phía bên trên.

Một thư mục được xóa bằng hàm rmdir. Nhớ rằng, một thư mục rỗng là thư mục có chứa hai mục chấm và chấm-chấm.



## Đọc thư mục

Thư mục có thể được đọc bởi bất kì ai có quyền đọc thư mục đó. Nhưng chỉ có kernel mới có quyền ghi vào file thư mục. Nhắc lại rằng, bit quyền ghi và bit quyền thực thi cho một thư mục dùng để xác định xem chúng ta có thể tạo file mới trong thư mục và xóa file từ thư mục đó – Chúng không cho phép chúng ta ghi vào file thư mục. Dươi đây là liệt kê các hàm đọc thư mục:

Hàm fdopendir giúp chúng ta chuyển từ file descriptor sang cấu trúc DIR để dùng trong các hàm handle.

Cấu trúc DIR dùng trong bảy hàm trên để xác định thư mục đang được đọc. Con trỏ DIR được trả về bởi hàm opendir và fdopendir sẽ được sử dụng bởi năm hàm còn lại. Hàm opendir sẽ khởi tạo mọi thứ, sau đó hàm readdir sẽ trả về phần tử đầu tiên của thư mục. Khi cấu trúc DIR tạo ra bởi hàm fdopendir, thì phần tử đầu tiên trả về bởi hàm readdir phụ thuộc vào trường offset trong file descriptor truyền vào hàm fdopendir. Chú ý rằng, cách sắp xếp các phần từ trong thư mục sẽ phụ thuộc vào xem nó có xắp xếp theo thư tự tăng dần hoặc giảm dần.

## Hàm chdir, fchdir và hàm getcwd

Tất cả các process đều có thư mục làm việc hiện thời của nó. Thư mục này là nơi bắt đầu tìm kiếm cho tất cả các file có đường dẫn tương đối. Khi người dùng đăng nhập vào hệ thống UNIX, thì thư mục làm việc hiện thời thường bắt đầu ở thư mục được khai bào ở dòng thứ 6 trong file /etc/passwd – là thư mục home của người dùng. Thư mục làm việc hiện thời là một tham số được truyền vào trong hàm main khi chạy một process. Chúng ta có thể thay đổi thư mục làm việc hiện thời bằng cách gọi hàm chdir hoặc fchdir:

Chúng ta có thể chỉ rõ thư mục làm việc mới bằng pathname hoặc thông qua một file descriptor.

## Giới thiệu về device file

Trong hệ điều hành Linux thì device file là một loại file đặc biệt, nó là giao diện của một device driver trong hệ thống. Các device file này cho phép các chương trình giao tiếp với các thiết bị thông qua device driver bằng cách đọc ghi file thông thường.

Device file có hai loại chính là character file và block file. Điểm khác biệt giữa hai loại file này là cách đọc ghi dữ liệu. Character file có thể đọc ghi từng ký tự một, còn block file cần phải đọc ghi với một kích thước cố định gọi là buffer.

Trong Linux, hầu hết các device file đều được đặt trong thư mục /dev, mỗi file này có thể đang được sử dụng bởi một deamon nào đó để điều khiển hardware. Trong chương 2 chúng ta đã lấy ví dụ về việc đọc vị trí con trỏ chuột thông qua device file của chuột là /dev/input/event6.

# GIỚI THIỆU VỀ PROCESS

## Mở đầu

Tất cả các chương trình trong Linux thực chất đều là các processes. Terminal bạn chạy, vim, hay bất cứ lệnh nào bạn gõ vào terminal. Process chính là đơn vị cấu thành nên Linux. Nó chính là một instance của chương trình bạn viết ra. Nói cách khác mỗi dòng code của bạn, sẽ được thực thi trên một process.

Linux cung cấp tool ps để list ra tất cả các process đang chạy trên hệ thống:

 Trong bài này, chúng ta sẽ tìm hiểu về cách thức hoạt động của một process thông thường. Từ điểm bắt đầu của một process là hàm main, cách thức chúng nhận tham số từ command-line truyền vào như thế nào, cấu trúc bộ nhớ điển hình của một process sẽ như thế nào, cách cấp phát bộ nhớ trong một process, và các cách thức để terminate một process. Cuối cùng, chùng ta sẽ tìm hiểu cách mà kernel phân quyền cho một process và làm một số ví dụ cơ bản về lập trình với process trong môi trường Linux.

## Hàm main

Một chương trình viết bằng ngôn ngữ C sẽ bắt đầu thực thi từ hàm main. Nguyên mẫu của hàm main là:

Trong đó tham số argc là số tham số truyền vào từ command-line và argv là mảng của các con trỏ.

Khi một chương trình C được thực thi bởi kernel – bằng các gọi hàm exec – một loạt các quá trình khỏi tạo sẽ diễn ra trước khi hàm main được thực thi. Khi biên dịch chương trình bằng compiler thì compiler sẽ chỉ rõ địa chỉ bắt đầu của process khi nó được chạy. Quá trình khởi tạo sẽ nhận giá trị bên ngoài truyền vào – từ command-line hoặc các biến môi trường.

## Terminate một process

Có khoảng tám cách để terminate một process, năm cách terminate thông thường đầu tiên gồm:

1. Khi kết thúc hàm main.
2. Khi gọi hàm exit.
3. Khi gọi hàm \_exit hoặc \_Exit.
4. Khi toàn bộ các thread của một process kết thúc.
5. Khi gọi hàm pthread\_exit

Ba cách không bình thường để thoát khỏi process bao gồm:

1. Gọi hàm abort
2. Khi nhận được một signal
3. Thread bị cancel.

Trong bài này chúng ta chỉ tìm hiểu về ba cách terminate đầu tiên.

## Các hàm exit

Ba hàm terminate process một cách thông thường là: \_exit và \_Exit, hai hàm này lập tức kết thúc chương trình rồi trở về kernel, với hàm exit, hàm này sẽ tiến hành một số quá trình dọn dẹp trước khi trở về kernel.



Tất cả các hàm trên đều nhận một tham số truyền vào là một số nguyên, nó là trạng thái trả về của chương trình. Trong Linux shell, chúng ta có thể kiểm tra giá trị trả về của chương trình chạy trước đó bằng lệnh echo $?. Nếu hàm main có kiểu trả về là void hoặc hàm main return mà không khai báo giá trị thì trạng thái trả về của chương trình là không xác định. Trả về giá trị bằng lênh return trong hàm main cũng tương đương với việc gọi hàm exit với cùng giá trị của return. Ví dụ

exit(0);

tương đương với

return 0;

trong hàm main.

Ví dụ: giá trị trả về của hàm main không khai báo kiểu trả về

#include <stdio.h>

main()

{

printf("hello, world\n");

}

Khi compile và chạy chương trình trên, chúng ta sẽ thấy giá trị trả về của chương trình là ngẫu nhiên. Nếu chúng ta compile chương trình này trên các máy khác nhau thì giá trị trả về náy thường sẽ khác nhau, nó phụ thuộc vào nội dung của bộ nhớ stack và các thanh ghi tại thời điểm hàm main kết thúc:



## Hàm atexit

Theo chuẩn ISO C, một process có thể khai báo lên đến 32 handler hàm mà chúng sẽ được tự động gọi khi process bị terminate. Các hàm này được gọi là hàm exit handler, và được khai báo bằng hàm atexit.



Từ nguyên mẫu hàm trên, chúng ta có thể thấy tham số truyền vào là con trỏ hàm void. Khi hàm handler này được gọi, nó không nhận bất cứ tham số truyền vào nào và cũng không trả về bất kì giá trị nào. Hàm exit gọi các hàm handler này theo thứ tự ngược lại với thứ tự mà chúng được khai báo. Các hàm handler được gọi ra bằng số lần mà chúng được khai báo, nếu một hàm handler đươc khai bào nhiều lần thì cũng sẽ được gọi ra bấy nhiêu lần lần.

Theo chuẩn ISO C và POSIX.1, hàm exit trước tiên sẽ gọi ra các hàm exit handler sau đó nó sẽ đóng tất cả các stream đang mở của process thông qua hàm fclose (ví dụ như đóng các file đang được mở). Hình dưới đây mô tả quá trình bắt đầu và kết thúc của một chương trình C.

|  |
| --- |
|  |
| Hình 4. 1: Vòng đời của một process |

Cách duy nhất một chương trình có thể chạy bởi kernel là gọi ra một trong hàm exec. Một process chỉ tự động đóng khi nó gọi ra hàm \_exit hoặc \_Exit, một cách trực tiếp hoặc gián tiếp qua hàm exit. Một process cũng có thể bị đóng bởi một signal.

Ví dụ về việc sử dụng hàn exit handler

#include <stdio.h>

#include <stdlib.h>

static void my\_exit1(void);

static void my\_exit2(void);

int main(void)

{

if (atexit(my\_exit2) != 0)

printf("can’t register my\_exit2");

if (atexit(my\_exit1) != 0)

printf("can’t register my\_exit1");

if (atexit(my\_exit1) != 0)

printf("can’t register my\_exit1");

printf("main is done\n");

return(0);

}

static void my\_exit1(void)

{

printf("first exit handler\n");

}

static void my\_exit2(void)

{

printf("second exit handler\n");

}

Khi chạy chương trình, chúng ta thu được kết quả như sau:

Hàm exit handler được gọi ra bằng số lần mà nó được khai bào, như kết quả bên trên, hàm exit handler đầu tiên được khai báo hai lần lên nó được gọi ra hai lần. Chú ý rằng chương trình bên trên chúng ta không dùng hàm exit mà gọi ra lệnh return, hai hàm này tương đương nhau.

## Tham số truyền vào từ command-line

Khi một chương trình được khỏi chạy từ một process nào đó bằng hàm exec thì process có thể truyền tham số từ command-line vào chương trình mới này. Đây là một phần của các hệ thống Unix/Linux shell. Chúng ta đã dùng tính năng truyền tham số này trong các ví dụ trước đây.

Ví dụ dưới đây in ra các tham số truyền vào của một chương trình:

#include <stdio.h>

int main(int argc, char \*argv[])

{

int i;

for (i = 0; i < argc; i++) /\* echo all command-line args \*/

printf("argv[%d]: %s\n", i, argv[i]);

return(0);

}

Compile và chạy chương trình trên, chúng ta sẽ thu được kết quả như sau:



## Danh sách biến môi trường

Mỗi chương trình cũng sẽ được truyền vào một danh sách các biến môi trường. Giống như tham số truyền vào từ command-line, danh sách biến môi trường là một mảng của những con trỏ, mỗi con trỏ là một chuỗi kí tự kết thưc bằng NULL. Địa chỉ của mảng này nằm trong biến global environ:

Dưới đây là ví dụ về biến môi trường gồm năm phần tử, chúng đều được kết thúc bằng kí tự null.

|  |
| --- |
|  |
| Hình 4. 2: Danh sách biến môi trường |

Chúng ta sẽ gọi biến environ là environment pointer, nó là con trỏ trỏ đến danh sách biến môi trường, và mỗi phần từ là một chỗi có tên environment strings.

Theo quy ước, biến môi trường gồm những chỗi có dạng:

*name=value*

Hầu hết name là những kí tự được viết hoa. Ví dụ dươi đây in ra tất cả các biến môi trường mà chương trình nhận được

#include <stdio.h>

extern char \*\*environ;

int main(int argc, char \*argv[])

{

int i;

for (i = 0; environ[i] != NULL; i++)

printf("environ[%d]: %s\n", i, argv[i]);

return(0);

}

Compile và chạy trương trình trên, chúng ta thu được kết quả sau:



## Biến môi trường

Như chúng ta đã biết ở trên, một chỗi biến môi trường có dạng:

*name=value*

Linux kernel không quan tâm địch dạng của chỗi này như thế nào mà nó phải được xử lý bởi các chương trình trên tầng user. Ví dụ như chương trình shell, chúng sử dụng rất nhiều biến môi trường. Một số sẽ tự động đươc khởi tạo khi đăng nhập như biến HOME và USER, còn các biến khác sẽ tùy thuộc vào người dùng và các chương trình khác. Theo chuẩn ISO C, để lấy giá trị từ biến môi trường ta dùng hàm getenv:



Hàm hàm trả về con trỏ đến giá trị value của một chuỗi biến môi trường name=value. Chúng ta lên sử dụng hàm này để lấy giá trị thay vì trực tiếp đọc từ biến environ.

Ví dụ chương trình dưới đây in ra giá trị của biến môi trường HOME và USER

#include <stdlib.h>

#include <stdio.h>

int main(int argc, char \*\*argv)

{

char \*value;

value = getenv("HOME");

printf("HOME: %s\n", value);

value = getenv("USER");

printf("USER: %s\n", value);

return 0;

}

Compile và chạy trương trình, chúng ta sẽ thu được kết quả sau đây:



## Cấu trúc bộ nhớ của một chương trình

Thông thường, cấu trúc bộ nhớ của một chương trình C được tạo bởi các phần như sau:

* Vùng Text, là vùng chứa mã lệnh mà CPU thực thi. Vùng này thường có thể chia sẻ được nên chỉ cần một vùng cố định đối với những chương trình thường xuyên được thực thi, ví dụ như trình soạn thảo văn bản, của sổ dòng lệnh shell, chúng ta có thể mở nhiều của sổ cùng một lúc nhưng chúng chia sẻ cùng một vùng text. Vùng text là vùng chỉ đọc để ngăn chương trình tự thay đổi mã lệnh của nó.
* Vùng dữ liệu đã được khởi tạo, thường được gọi ngắn gọn là vùng data, nó chứa các biến mà đã được khỏi tạo khi biên dịch chương trình. Ví dụ như khai báo biến sau trong C:

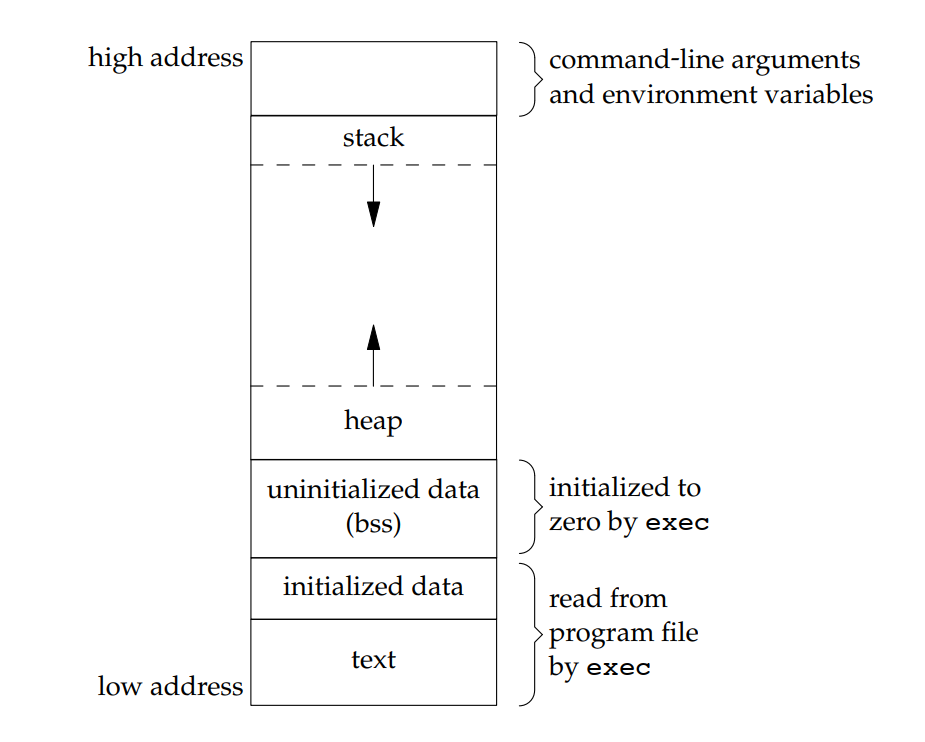


* Vùng dữ liệu chưa được khỏi tạo, thường được gọi là vùng bss. Dữ liệu của vùng này được kernel gán cho giá trị 0 hoặc con trỏ null trước khi chương trình được thực thi. Ví dụ như một biến, mảng, hoặc con trỏ được khai báo nằm ngoài tất cả các hàm:



* Vùng stack, nơi các biến tự động được lưu trữ, cùng với dữ liệu lưu lại mỗi lần gọi hàm. Khi một hàm được gọi, nó sẽ được cấp phát vùng nhớ dùng cho các biến cục bộ của nó trên vùng stack… Ngoài ra vùng stack còn chứa thông tin về một số biến môi trường và giá trị của một số thanh ghi của máy tính.
* Vùng heap, dùng cho việc cấp phát động bộ nhớ.

Hình dưới đây mô tả cấu trúc bộ nhớ của một chương trình:



Hình 4. 3: Cấu trúc vùng nhớ của một process

Trong shell chúng ta có thể dùng lênh size để xem cấu trúc vùng nhớ của một chương trình bất kì:



## Cấp phát bộ nhớ trong process

ISO C quy định 3 hàm dùng để cấp phát bộ nhớ:

1. malloc, hàm này dùng để cấp phát bộ nhớ theo kích thước byte tùy ý. Giá trị khỏi tạo của vùng nhớ là không xác định.
2. calloc, hàm này dùng để cấp phát một số nguyên lần kích thước của một đối tượng. Giá trị của các byte trong vùng nhớ là 0.
3. realloc, hàm này dùng để tăng hoặc giảm kích thước của một vùng nhớ đã cấp phát trước đó. Khi kích thước của vùng nhớ tăng, thì vùng nhớ cũ có thể sẽ được di chuyển đến vị trí khác để có đủ bộ nhớ cần cấp phát vào cuối, vùng nhớ mới được cấp phát thêm có giá trị không xác định.

Kiểu trả về của ba hàm cấp phát bộ nhớ là con trỏ void \*, khi gán con trỏ này cho loại con trỏ khác, chúng ta không cần phải dùng phép ép kiểu.

Hàm free giải phóng bộ nhớ trỏ đến bởi con trỏ ptr. Bộ nhớ bị giải phóng này sẽ được để vào một bể bộ nhớ trống và có thể sẽ được cấp phát lại bằng ba hàm trên.

Hàm realloc giúp chúng ta thay kích thước của một vùng đã cấp phát trước đó, thường dùng để tăng kích thước vùng nhớ. Ví dụ, nếu chúng ta đã cấp phát bộ nhớ cho một mảng gồm 512 phần tử và dùng hết vùng nhớ này, khi muốn có thêm phân phần tử trống chúng ta dùng hàm realloc. Chú ý rằng khi dùng hàm realloc để tăng kích thước vùng nhớ thì vùng nhớ đó có thể bị di chuyển đến vị trí khác để có đủ chỗ trống, vì thế lên con trỏ trỏ đến vùng nhớ có thể bị thay đổi.

Tham số cuối cùng của hàm realloc là kích thước của vùng nhớ mới, nó không phải là kích thước chênh lệch của vùng nhớ mới và cũ. Nếu con trỏ ptr là null thì hàm realloc sẽ hoạt động giống như hàm malloc và cấp phát một vùng nhớ mới có kích thước newsize.

Khi chúng ta ghi dữ liệu tràn ra biên trên và dưới của vùng nhớ được cấp phát động thì chúng ta có thể sẽ hư hại dữ liệu của vùng nhớ khác đang được sử dụng, ví chúng cũng có thể đang được cấp phát động cho biến khác. Khi ghi tràn vùng nhớ cấp phát động xảy ra thì việc xác định nguyên nhân lỗi của chương trình là rất khó khăn.

Một lỗi nguy hiểm khác có thể xảy ra là chúng ta giải phóng bộ nhớ đã được giải phóng trước đó hoặc giải phóng bộ nhớ không được cấp phát bằng các hàm alloc. Nếu process gọi hàm malloc nhưng quên gọi hàm free thì bộ nhớ của process sẽ tăng lên, nó được gọi là rò rỉ bộ nhớ (leakage).

## Thư viện chung

Hầu hết các hệ thống Unix/Linux ngày nay đều hỗ trợ việc sử dụng các thư viện chung (shared libaries). Thư viện chung này là các thư viện code được chia sẻ và dùng chung bởi nhiều process. Thư viện chung này thay thế các thư viện thường được build trong code của các process. Thay vì mỗi process đều có một bản sao chép của cùng một thư viện thì hệ thống sẽ đặt thư viện này ở một ví trí xác định và các process có thể cùng dùng thư viện này mà không cần phải copy trong file thực thi. Nó làm giảm kích thước của file thực thi, nhưng sẽ làm chậm quá trình khởi tạo chương trình khi nó chạy lần đầu tiên. Một ưu điểm của thư viện chung là khi thư viện chung có update phiên bản mới thì chương trình dùng thư viện đó sẽ không cần phải biên dịch lại.