# Phần lõi của hệ thống: Kernel

Thuật ngữ hệ điều hành nhìn chung được sử dụng theo 2 nghĩa sau:

* Để chỉ toàn bộ các gói phần mềm trung tâm quản lý các tài nguyên máy tính và tất cả các công cụ phần mềm đi cùng, như là trình thông dịch dòng lệnh, GUI, tiện ích với các file và các trình sửa file.
* Sâu hơn, tham chiếu vào các phần mềm trung tâm mà quản lý và cung cấp các tài nguyên máy tính (ví dụ, CPU, RAM và các thiết bị ngoại vi, v..v.).

Thuật ngữ kernel thường sử dụng có nghĩa tương tự với định nghĩa thứ hai.

Mặc dù có thể chạy các chương trình trên máy tính mà không cần kernel, sự hiện diện của kernel giúp đơn giản hóa trong việc viết và chạy các chương trình, đồng thời tăng khả năng và sự mềm dẻo cho các lập trình viên. Kernel thực hiện điều này bằng cách cung cấp một lớp phần mềm giúp quản lý và giới hạn tài nguyên của máy tính.

## Các nhiệm vụ được thực hiện bởi kernel

Kernel thực hiện các nhiệm vụ như sau:

* Lên lịch tiến trình: một máy tính có thể có một hoặc nhiều CPU. Giống như các hệ thống UNIX khác, Linux là một hệ điều hành đa nhiệm ưu tiên. Đa nhiệm ở đây nghĩa là có nhiều tiến trình có thể đồng thời được chạy cùng một lúc. Ưu tiên ở đây là các luật được xác định để quản lý tiến trình cung cấp cho nó thời gian sử dụng CPU dựa trên trình lập lịch tiến trình của kernel từ đó xác định tiến trình nào sử dụng CPU trước, tiến trình nào sau.
* Quản lý bộ nhớ: kích thước các phần mềm càng ngày càng tăng thì bộ nhớ vật lý vẫn là tài nguyên bị giới hạn mà kernel phải chia sẻ giữa các tiến trình theo một cách công bằng và hiệu quả nhất. Giống đa phần các hệ điều hành hiện đại, Linux áp dụng quản lý bộ nhớ ảo, một kỹ thuật đem lại hai lợi ích chính:
  + Tiến trình được cô lập khỏi và tiến trình khác và kernel, vậy nên một tiến trình không thể đọc hoặc thay đổi bộ nhớ của tiến trình khác hoặc của kernel.
  + Chỉ có một phần của tiến trình được lưu trong bộ nhớ, do đó làm giảm bộ nhớ yêu cầu cho mỗi tiến trình và cho phép nhiều tiến trình được tổ chức trên RAM đồng thời hơn. Điều này dẫn đến việc tối ưu CPU, bởi vì nó tăng khả năng mà tại một thời điểm, có ít nhất 1 tiến trình mà CPU có thể thực thi.
* Cung cấp hệ thống quản lý file (file system): Kernel cung cấp một file system ở trên ổ đĩa, cho phép các file có thể được tạo, lấy, cập nhật, xóa, và ..v..v.
* Tạo và kết thúc các tiến trình: Kernel có thể tải một chương trình mới vào trong bộ nhớ,cung cấp nó với tài nguyên ( như CPU, bộ nhớ, và truy cập tới các file ) mà nó cần để chạy. Một chương trình máy tính mà đang chạy thì được coi như là một tiến trình (process). Một khi tiến trình hoàn thành, kernel đảm bảo rằng tài nguyên nó sử dụng được giải phóng cho việc tái sử dụng sau đó bởi chương trình sau.
* Truy cập vào các thiết bị: các thiết bị (chuột, màn hình, bàn phím, ổ cứng, ổ đĩa và .v.v.) được kết nối với máy tính cho phép truyền tải thông tin giữa máy tính và thế giới bên ngoài, cho phép vào, ra và cả hai. Kernel cung cấp các chương trình với interface được chuẩn hóa và đơn giản cho việc truy cập tới các thiết bị.
* Mạng máy tính (networking): kernel chuyển và nhận các tin nhắn trong mạng máy tính (các gói) ở trên một nửa số tiến trình của người sử dụng. Nhiệm vụ này bao gồm định tuyến các gói tin tới hệ thống xác định.
* Cung cấp một giao diện lời gọi hệ thống: các tiến trình có thể yêu cầu kernel để thực hiện nhiều nhiệm vụ sử dụng các điểm truy cập vào kernel ( các lời gọi hệ thống ).

Ngoài các chức năng trên, với hệ điều hành multiuser như Linux cung cấp cho người sử dụng một lớp trừu tượng gọi là *virtual private computer*; mà mỗi người sử dụng có thể truy cập vào hệ thống và thực hiện độc lập với người sử dụng khác. Ví dụ, mỗi người sử dụng có không gian lưu trữ riêng (thư mục home). Ngoài ra, người sử dụng có thể chạy các chương trình mà chúng chia sẻ chung CPU và thực hiện trong không gian địa chỉ ảo của chúng, các chương trình này truy cập độc lập tới các thiết bị ngoại vi và vận chuyển thông tin trong mạng. Kernel giải quyết các đụng độ tiềm năng trong truy cập tài nguyên phần cứng, nên người sử dụng và các tiến trình không thể nhận thấy các đụng độ.

## Kernel mode và user mode

Các kiến trúc vi xử lý hiện đại thường cho phép CPU thực hiện ít nhất 2 chế độ khác nhau là *user mode* và *kernel mode* ( có thể được biết tới như là *supervisor mode* ). Các tập lệnh phần cứng cho phép thay đổi từ chế độ này qua chế độ khác. Tương ứng với đó, các khu vực của bộ nhớ ảo thể được đánh dấu là *“user space”* hoặc *“kernel space”.* Khi đang chạy trong user mode, CPU chỉ có thể truy cập vào phần bộ nhớ mà nó được đánh dấu là user space. Trong khi ở trong kernel mode, CPU có thể truy cập vào cả 2 vùng nhớ user space và kernel space.

Một số hoạt động chỉ có thể thực hiện ở trong kernel mode. Ví dụ như: thực hiện dừng các chương trình hệ thống, truy cập vào phần cứng bộ nhớ quản lý và khởi tạo các vào ra trong thiết bị. Với các lợi ích của việc sử dụng kernel, hệ điều hành đảm bảo cho các tiến trình của người dùng không thể truy cập vào tập lệnh và cấu trúc dữ liệu của kernel hoặc làm ảnh hưởng tới các hoạt động trong hệ thống.

# Shell

Shell là một chương trình được thiết kế với một mục đích đặc biệt là để đọc các câu lệnh được nhập vào bởi người sử dụng và thực hiện các chương trình tương ứng với các câu lệnh đó. Các chương trình như vậy có thể được biết như là một trình thông dịch lệnh (*command interpreter*).

Thuật ngữ *login shell*  được sử dụng để nói đến tiến trình mà được tạo ra để chạy một shell khi người sử dụng đăng nhập lần đầu tiên.

Trong khi trên một vài hệ điều hành, trình thông dịch lệnh là phần bên trong của kernel thì trên UNIX, shell là một tiến trình của người dùng. Có thể khởi tạo nhiều shell khác nhau cùng tồn tại bởi một người sử dụng hoặc nhiều người sử dụng ở trên cùng một máy tính. Có nhiều loại shell khác nhau như là:

* *Bourne shell (sh):* đây là shell lâu đời nhất được sử dụng rộng rãi và được viết bởi Steve Bourne. Nó là shell chuẩn cho Seventh Edition UNIX. Bourne shell bao gồm nhiều tính năng quen thuộc có trong tất cả shell: điều hướng I/O, pipe-lines, file generation, các biến, các biến môi trường, thực hiện các câu lệnh nền, các hàm.
* *C shell (csh)*: được viết bởi Bill Joy tại đại học California ở Berkeley. C shell cung cấp một vài tính năng hữu dụng mà Bourne shell không có như: lịch sử lệnh, alias, command-line editing, job control. C shell không tương thích ngược với Bourne shell.
* *Korn shell (ksh)*: Shell này
* *Bourne again shell (bash)*

Các shell không chỉ được thiết kế cho việc sử dụng tương tác trực tiếp, mà còn cho việc thông dịch các *shell scripts*, cái là các tập tin văn bản bao gồm các lệnh shell.

# Users và Groups

Mỗi người sử dụng ở trong hệ thống được định danh một cách đơn nhất, và người sử dụng có thể thuộc vào các nhóm.

## Users

Mỗi người sử dụng của hệ thống có *login name* và một số sô *user ID (UID)*. Đối với mỗi người sử dụng, chúng được định nghĩa bởi một dòng trong tập tin hệ thống password, */etc/passwd*, cái mà bao gồm các thông tin sau:

* *GroupID*: mã ID của nhóm mà người dùng là thành viên
* *Thư mục home*: Đường dẫn khởi tạo mà người dùng truy cập vào sau khi đăng nhập.
* *Login shell*: tên của chương trình mà được thực hiện để thông dịch câu lệnh của người dùng.

Tập tin *password* cũng có thể lưu lại mật khẩu của người sử dụng được mã hóa. Tuy nhiên vì lý do bảo mật, mật khẩu của người dùng thường được lưu trữ ở trong *password file,* cái mà có thể đọc chỉ bởi người dùng có đặc quyền.

## Groups

Đối với việc truy cập vào các tập tin và các tài nguyên hệ thống khác – khi người dùng được tổ chức thành các nhóm sẽ đem lại hiệu quả tốt hơn. Ví dụ, người dùng thuộc một nhóm làm việc chung với nhau trong một dự án, do đó việc chia sẽ một tập các file chung cho tất cả các thành viên trong nhóm là việc cần thiết. Mỗi nhóm được định danh bởi một dòng trong *group file* của hệ thống, */etc/group*, bao gồm các thông tin sau:

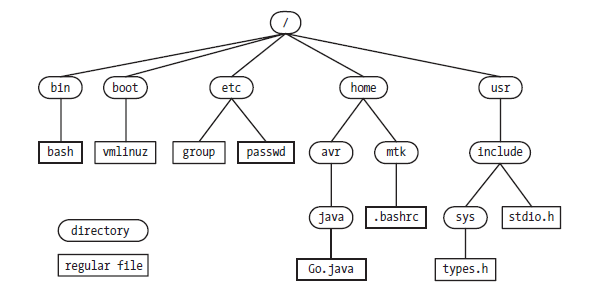
* *Group name:* Tên của nhóm ( duy nhất).
* *Group ID*: một số để định danh được liên kết với nhóm.
* *User list*: một danh sách được ngăn cách bởi dấu phẩy gồm các *login name* của người dùng mà là thành viên của nhóm.

## Superuser

Một người dùng, được biết như là *superuser*, có các quyền đặc biệt trong hệ thống. Tài khoản superuser có mã ID là 0, và thường truy cập với tên là *root*. Trong hệ thống UNIX thông thường, superuser được cấp cho tất cả các quyền kiểm tra trong hệ thống. Do đó ví dụ, *superuser* có thể truy cập vào bất cứ file nào trong hệ thống bất kể quyền truy cập của file đó, và có thể gửi các signal tới bất kì tiến trình của người dùng trong hệ thống.

# Phân cấp thư mục đơn, các thư mục, liên kết và các file

Kernel duy trì một cấu trúc thư mục phân cấp đơn để tổ chức tất cả các file trong hệ thống. (Điều này ngược lại với một số hệ điều hành như Microsoft Windows, khi mà mỗi ổ đĩa có hệ thống thư mục phân cấp của riêng nó). Tại điểm bắt đầu của hệ thống phân cấp này là thư mục *root*, được đặt tên là “/”. Tất cả các file và đường dẫn là con hoặc xa hơn là các cháu và hậu duê của thư mục *root*. Hình dưới chỉ ra ví dụ của cấu trúc phân cấp này.



## Các loại file

Trong hệ thống file, mỗi file sẽ được đánh dấu là một trong các loại: file chứa dữ liệu nguyên thủy (hay được gọi là *regular file* hoặc *plain file* để phân biệt với các loại khác), device file, pipe, socket, directory, symbolic link.

Thuật ngữ file nhìn chung được sử dụng để chỉ bất cứ loại tập tin nào, không chỉ là một *regular file*.

## Thư mục và liên kết

Một thư mục là một file đặc biệt mà nội dung của nó chứa một bảng các filename giúp cho việc tham chiếu tới các file tương ứng. Cái mà có một entry point tới file kết hợp với filename được gọi là một liên kết, và các file có nhiều liên kết và do đó có nhiều tên ở trong cùng hoặc các thư mục khác nhau.

Các thư mục bao gồm cả 2 loại liên kết tới các file và tới các thư mục khác. Các liên kết giữa các thư mục được thiết lập trong phân cấp thư mục chỉ ra ở hình trên.

Mỗi thư mục bao gồm ít nhất 2 file là: . (chấm), liên kết tới bản thân nó, và .. (chấm-chấm) , liên kết tới thư mục cha của nó. Mọi thư mục, ngoại trừ thư mục *root*, đều có cha. Đối với thư mục *root*, file chấm-chấm là một liên kết tới bản thân nó (do đó, /.. bằng với /).

## Symbolic link

Giống như một liên kết bình thường, *symbolic link* cung cấp một cái tên khác cho file. Nhưng trong khi liên kết bình thường là một file name kết hợp với một entry point trỏ vào file thì một *symbolic link* là một file bao gồm tên của một file khác. File khác ở đây hay được gọi là mục tiêu (*target*) của *symbolic link,* và nó nhìn chung thì *symbolic link* trỏ tới hoặc tham chiếu tới file mục tiêu. Khi được dẫn được xác định trong một lời gọi hệ thống, kernel tự động thay thế mỗi *symbolic link* trong hệ thống với filename mà nó trỏ tới. Nếu một *symbolic link*  tham chiếu tới một file không tồn tại, nó được gọi là *dangling link*.

Thông thường thuật ngữ *hard link* và *soft link* được sử dụng để phân biệt giữa liên kết thông thường với *symbolic link*.

## Filenames

Trong đa phần hệ thống quản lý file Linux, filename có thể lên tới độ dài 255 kí tự. Filename có thể bao gồm bất kỳ kí tự nào ngoại trừ dấu chéo ( / ) và kí tự null ( \0 ). Tuy nhiên, nó được khuyến khích sử dụng chỉ các chữ cái và số, và dấu chấm ( . ), dấu gạch chân ( \_ ), và dấu gạch ngang ( - ). 65 kí tự này được viết trong SUSv3 với tên *portable filename character set*.

## Pathname

Một *đường dẫn (pathname)*  là một chuỗi gồm có thể gồm các dấu chéo ( / ) ở đầu chuỗi và sau đó là chuỗi các filename được chia bởi dấu chéo. Tất cả các filename này trừ file name cuối cùng đều là các thư mục. Filename cuối cùng có thể là bất cứ loại file nào. Một *đường dẫn* mô tả địa chỉ của một file trong hệ thống phân cấp thư mục đơn:

* Đường dẫn tuyệt đối: bắt đầu với dấu chéo ( / ).
* Đường dẫn tương đối: xác định địa của tương đối của file so với thư mục hiện tại đang làm việc của tiến trình.

## Quyền sở hữu file và các quyền hạn

Mỗi file có một user ID và group ID được liên kết mà định nghĩa người sở hửu file và nhóm mà file đó thuộc về. Quyền sở hữu một file thường được xác định bằng khả năng truy cập của người dùng tới file.

# Mô hình I/O ( Input/Output)

Giống với phương châm trên hệ thống Unix: “Trong một hệ thống Unix, mọi thứ đều là file. Nếu không phải là file thì nó là tiến trình”. Vì vậy, khi thực hiện các hành động I/O ( Input/Output) trong Linux dựa vào các file. Các lời gọi hệ thống (như *open(), read(), write(), close(), và v.v..* )đều được thực hiện ở trên file.

## File descriptor

Các I/O trên hệ thống được sử dụng với việc mở ra một file descriptor. Đây là một file đặc biệt giúp truy cập vào các tài nguyên input và output trong hệ thống.

Thông thường trong một tiến trình sẽ có 3 file descriptor khi nó được bắt đầu bởi shell.

* Descriptor 0 là *standard input*, sử dụng khi tiến trình yêu cầu nhập vào
* Descriptor 1 là *standard output*, sử dụng khi tiến trình yêu cầu xuất ra
* Descriptor 2 là *standard error*, sử dụng để tiến trình thông báo lỗi, ghi ra các trạng thái bất thường.

# Program

Chương trình tồn tại thông thường theo 2 dạng là *source code* và *binary machine-language instructions*. *Source code* là văn bản dạng chữ mà con người có thể đọc và được viết tuân theo quy tắc của một ngôn ngữ lập trình nào đó ví dụ như C. Để chương trình có thể được thực hiện thì cần chuyển *source code* sang dạng thứ 2, *binary machine-language instructions*, bao gồm một tập các chỉ thị mà máy tính có thể hiểu được. Công việc chuyển đổi phải trải qua các bước như *dịch* (compile) và *liên kết* (linking).

# Tiến trình

Hiểu một cách đơn giản, một tiến trình là một chương trình đang chạy. Khi một chương trình được thực thi, kernel tải các đoạn mã của chương trình vào trong bộ nhớ ảo, cấp phát bộ nhớ cho các biến, lưu lại các thông tin (như process ID, trạng thái kết thúc, userIDs v.v. ) về tiến trình.

Với góc nhìn của kernel, tiến trình là các thực thể mà trong đó kernel phải chia sẽ các nguồn tài nguyên của máy tính. Vì tài nguyên là có hạn, nên kernel sẽ chịu trách nhiệm cung cấp, điều chỉnh nguồn tài nguyên sao cho hợp lí. Khi tiến trình kết thúc, kernel sẽ giải phóng nguồn tài nguyên cấp cho tiến trình để có thể tiếp tục cung cấp cho các tiến trình khác.

## Lớp bộ nhớ của tiến trình

Bộ nhớ của một tiến trình được chia làm nhiều phần, được biết như là các *đoạn* (segment) :

* *Text*: các chỉ thị của chương trình
* *Data*: các biến tĩnh sử dụng bởi chương trình
* *Heap*: nơi mà chương trình cung cấp các bộ nhớ động
* *Stack*: thường được cung cấp cho các biến địa phương, các thông tin lưu trữ cần thiết với các lời gọi hàm.

## Tạo tiến trình và thực thi tiến trình

Một tiến trình có thể tạo ra một tiến trình mới nhờ lời gọi hệ thống *fork()*. Tiến trình mà gọi đến fork() là tiến trình cha và tiến tình mới được tạo ra là tiến trình con. Kernel tạo ra một tiến trình con bằng cách tạo một bản sao của tiến trình cha. Tiến trình con sẽ sao chép toàn bộ dữ liệu, stack, heap của tiến trình cha.

Tiến trình con có thể thực hiện tập lệnh khác với tiến trình cha bằng cách goi đến lời gọi hệ thống *execve()* để tải chương trình mới vào tiến trình con.

## Process ID và parent process ID

Mỗi tiến trình đều có một *process identifier* (PID) duy nhất. Mỗi tiến trình cũng có thuộc tính *parent process identifier* (PPID), các giá trị này giúp định danh tiến trình khi kernel tạo ra một tiến trình mới.

### Kết thúc tiến trình và trạng thái kết thúc của tiến trình

Một tiến trình có thể được kết thúc theo 2 cách:

* Sử dụng lời gọi hệ thống *\_exit()* để kết thúc
* Có một signal kết thúc tiến trình được gửi tới

Khi tiến trình kết thúc dù bằng cách nào trong 2 cách thì nó đều trả về một *trạng thái kết thúc*, là một số nguyên không âm nhỏ. Tiến trình cha có thể kiểm tra *trạng thái kết thúc* bằng lời gọi hệ thống wait().

Theo quy ước, trạng thái kết thúc 0 chỉ ra tiến trình thực hiện thành công và trạng thái khác 0 chỉ ra tiến trình xảy ra lỗi.

## Process user và group identifiers

Mỗi tiến trình đều liên kết với một số lượng các user IDs (UIDs) và group IDs (GIDs).

* *Real user ID* và *real group ID*: là những người dùng và nhóm người dùng sở hữu tiến trình. Mỗi tiến trình con được sinh ra đều kế thừa *real user ID* và *group user ID* từ tiến trình cha.
* *Effective user ID* và *effective group ID*: đây là những người dùng và nhóm người dùng mà tiến trình có thể tác động vào. Dùng để xác định quyền của tiến trình tới các tài nguyên của các người dùng và các nhóm người dùng.
* *Supplemetary group IDs*: các ID này dùng để xác định các nhóm thêm vào ngoài *Real user ID* và *Real group ID* mà tiến trình thuộc về. Một tiến trình con cũng sẽ kế thừa toàn bộ *Supplementary group ID* của tiến trình cha.

## Privileged processes

Thông thường trong hệ thống UNIX, một *privileged process* là một tiến trình mà có *effective user* là 0 (superuser). Các tiến trình như vậy có thể bỏ qua mọi giới hạn truy cập được tạo dựng bởi kernel. Ngược lại, ta có thuật ngữ *unprivileged process* là các tiến trình và có *effective user* khác 0 và phải chịu sự hạn chế về quyền được thiết lập bởi kernel.

## Capabilities

Từ phiên bản kernel 2.2, Linux chia privileges của super user thành một tập các đơn vị riêng rẽ gọi là *capabilities*. Mỗi một hành động privileges được liên kết với một *capability* cụ thể, và tiến trình có thể thực hiện hành động privileges chỉ khi nó có *capability* tương ứng. Một tiến trình superuser tất nhiên sẽ có toàn bộ các *capabilities*.

Dựa vào việc cung cấp các *capability* mà có thể kiểm soát tiến trình thực hiện các hành động xác định với quyền của superuser và cấm các hành động khác.

## Tiến trình *init*

Khi khởi động vào hệ thống, kernel tạo ra một tiến trình đặc biệt gọi là *init*, “cha của mọi tiến trình”, được chạy từ file /sbin/init. Tất cả các tiến trình trong hệ thống đều được tạo bởi tiến trình init hoặc các hậu duệ của nó. Tiến trình *init* luôn có PID là 1 và luôn chạy với quyền superuser. Tiến trình này không thể kết thúc kể cả khi có quyền superuser, nó chỉ kết thúc khi hệ thống bị tắt đi. Nhiệm vụ chính của tiến trình *init* này là giám sát các tiến trình khác trong hệ thống.

## Deamon process

Tiến trình *deamon* là một tiến trình được sinh ra với mục đích đặc biệt. Nó được khởi tạo và điều khiển bởi hệ thống giống như với tiến trình khác, nhưng nó có các tính chất đặc biệt sau:

* Nó tồn tại xuyên suốt với hệ thống. Thường là từ lúc hệ thống được khởi động cho đến khi đến thống bị tắt.
* Nó chạy ngầm và không có controlling terminal nào mà từ đó có thể đọc vào hay ghi ra.

Ví dụ của các *deamon* process như là: *syslogd* lưu trữ các tin nhắn trong system log, hoặc *httpd* để phục vụ các trang web thông qua giao thức Hypertext Transfer (HTTP).

## Environment list

Mỗi process đều có một *environment list*. Trong đó, *environment list* chứa một tập các *environment variable* mà được duy trì ở trong bộ nhớ user-space của tiến trình. Mỗi *environment variable* bao gồm một tên và một giá trị tương ứng đi với nó. Khi một tiến trình mới được tạo thông qua lệnh *fork()*, nó kế thừa toàn bộ *environment list* của tiến trình cha. Do đó, *environment list* cung cấp một cơ chế giúp cho tiến trình cha giao tiếp với tiến trình con.

## Giới hạn tài nguyên

Mỗi tiến trình tiêu thụ nhiều nguyền tài nguyên của hệ thống, như là các file, bộ nhớ, thời gian CPU. Sử dụng lời gọi hàm *setrlimit()*, một tiến trình có thể thiết lập giới hạn tiêu thụ tài nguyên máy tính tối đa. Có 2 giá trị được sử dụng với mỗi giới hạn là :

* *Soft limit* : đưa ra các giới hạn về số lượng tài nguyên có thể tiêu thụ
* *Hard limit* : chứa một giá trị cận trên mà đựa vào đó để điều chỉnh *soft limit*

Một tiến trình unprivileged có thể thay đổi *soft limit* cho từng tài nguyên cụ thể trong khoảng từ 0 cho đến *hard limit*. Giá trị *soft limit* không thể vượt *hard limit*.

# Memory Mappings

Lời gọi hệ thống *mmap()* tạo ra một *memory mapping*  mới trong không gian địa chỉ ảo của tiến trình gọi đến nó. Mapping được chia làm 2 loại:

* *File mapping* ánh ra một đoạn dữ liệu của file vào trong bộ nhớ ảo của tiến trình gọi đến nó. Một khi ánh xạ thành công, nội dung của file có thể được truy cập bằng cách thực hiện ở trên các byte của vùng nhớ ảo ánh xạ tương ứng.
* *Anonymous mapping* không có một file để ánh xa. Thay vào đó, các trang địa chỉ được ánh xạ sẽ khởi tạo thành 0.

Bộ nhớ trong một tiến trình có thể chia sẻ với tiến trình khác bằng cách thực hiện ánh xạ tới cùng khu vực của file. Các tiến trình có thể nhìn thấy và thay đổi nội dung của bộ nhớ của tiến trình khác thông qua việc chia sẻ và quyền của chúng phụ thuộc vào việc chia sẻ ở trạng thái *private* hay *shared*. Khi ở *private*, các thay đổi nội dung không thể được nhìn thấy hay tác động tới các tiến trình khác. Còn khi ở *shared* thì các thay đổi về mặt nội dung trong bộ nhớ chia sẻ sẽ tác động tới các tiến trình còn lại.

# Static and Shared Libraries

Một *object library* là một fle bao gồm compiled object code chứa một tập các hàm có thể được gọi từ các chương trình ứng dụng. Việc đặt các đoạn mã của một tập hợp các hàm ở trong một single object library sẽ giúp giảm nhẹ công việc cho các chương trình. Hệ thống Linux hiện đại cung cấp 2 loại thư viện: *static libraries* và *shared libraries.*

## Thư viện Static

Vào thời điểm ban đầu của hệ thông Unix, thư viện *static* là loại thư viện duy nhất. Thư viện *static* cung cấp các nhóm được cấu trúc trong các compiled module. Để sử dụng các hàm từ một thư viện *static* cần phải xác định vị trí của thư viện bằng cách sử dụng các câu lệnh liên kết khi build một chương trình. Sau khi đã định được vị trí của thư viện, linker sao chép các object module từ thư viện và sao chép chúng vào trong file thực thi. Chúng ta nói rằng một chương trình như vậy là *statically linked*.

Thư viện Static cũng tồn tại một số nhược điểm do tính chất phải sao chép các object module vào file thực thi.

* Tốn bộ nhớ: Do việc phải sao chép vào file thực thi nên mỗi lần liên kết sẽ có một object module riêng và có thể xảy ra việc trùng lặp trong cùng một chương trình dẫn đến việc tốn bộ nhớ không đáng có. Ngoài ra, nhiều chương trình có thể liên kết đến cùng bộ thư viện nên cũng sinh ra việc tiêu hao bộ nhớ.
* Phải liên kết lại thư viện khi thư viện có sự thay đổi: do phải sao chép các object module nên khi có sự thay đổi trong thư viện gốc thì chương trình phải được liên kết lại.

## Thư viện Shared

Thư viện shared được thiết kế nhằm giải quyết các vấn đề của thư viện static.

Nếu một chương trình liên kết tới một thư viện shared, thì linker chỉ lưu lại thông tin giúp cho file thực thi biết vị trí của để khởi chạy thư viện shared. Khi file thực thi được tải vào trong bộ nhớ, chương trình sẽ gọi đến *dynamic linker* để đảm bảo thư viện shared tồn tại và gọi tới các hàm trong thư viện shared.

Tại một thời điểm bất kỳ, chỉ có duy nhất một bản sao chép của thư viện shared ở trong bộ nhớ, tất cả các chương trình sẽ sử dụng chung bản sao chép này.

Điều này sẽ vừa giúp tiết kiệm bộ nhớ và khi muốn cập nhật thư viện thì chỉ cần cập nhật lại duy nhất bản sao chép trong bộ nhớ RAM.

# Interprocess Communication (IPC) và đồng bộ

Trong một hệ thống Linux bao gồm một số lượng lớn các tiến trình, rất nhiều trong số chúng được thực hiện độc lập với nhau. Tuy nhiên, nhiều tiến trình cần kết hợp với nhau để thực hiện. Các tiến trình này cần phải giao tiếp với nhau và đồng bộ các hành động của chúng.

Các tiến trình có thể giao tiếp với nhau thông qua việc đọc và ghi thông tin lên file. Tuy nhiên, đối với nhiều chương trình, giải pháp này quá chậm và không mềm dẻo.

Vì vậy, Linux cung cấp một lượng lớn các cơ chế cho việc *interprocess communication (IPC – giao tiếp giữa các tiến trình)*:

* *signals*: được sử dụng để chỉ ra khi có “sự kiện” diễn ra
* *pipes* và *FIFOs* : được sử dụng để trao đổi dữ liệu giữa các tiến trình.
* *sockets* : sử dụng để vẫn chuyển dữ liệu giữa các tiến trình trên cùng một host hoặc khác host thông qua mạng.
* *file locking* : cho phép một tiến trình khóa một số vị trí của file để ngăn chặn việc đọc hoặc cập nhật nội dung file.
* *message queues* : trao đổi các thông điệp giữa các tiến trình.
* *semaphores* : để đồng bộ các hành động của các tiến trình.
* *shared memory* : cho phép hai hoặc nhiều tiến trình chia sẻ một phần bộ nhớ. Khi một tiến trình thay đổi nội dung trong vùng nhớ này, tất cả các tiến trình khác cũng sẽ nhìn thấy sự thay đổi.

# Signals

Signal thường được coi như các “ngắt phần mềm”. Khi signal đến, nó thông báo cho tiến trình rằng có một vài “sự kiện” hoặc trạng thái ngoại lệ xảy ra. Có rất nhiều loại signal, mỗi loại sẽ định danh một “sự kiện” hoặc trạng thái khác nhau.

Signal được gửi tới tiến trình bởi kernel hoặc tiến trình hoặc bởi bản thân tiến trình. Ví dụ, kernel có thể gửi một signal tới tiến trình khi xảy ra các sự kiện sau:

* Người sử dụng gõ kí tự *ngắt* ( thường là *Control-C* ) ở trên bàn phím.
* Một tiến trình con của nó kết thúc;
* Tiến trình cố gắng truy cập vào bộ nhớ không hợp lệ;

Trong shell, câu lệnh kill có thể được sử dụng để gửi signal tới một tiến trình.

Khi một tiến trình nhận được 1 signal, nó thực hiện một trong số các hành động sau, phụ thuộc vào signal:

* Bỏ qua signal
* Kết thúc bởi signal
* Người cho đến khi nhận được signal đặc biệt báo hiệu tiếp tục

Đối với hầu hết các loại signal, thay vì thực hiện các hành động mặc định, một chương trình có thể bỏ qua signal, hoặc thiết lập một *signal handler*. Một *signal handler* là một hàm do lập trình viên định nghĩa, sẽ tự động gọi đến khi signal được cung cấp tới tiến trình.

Trong khoảng thời gian giữa lúc signal được sinh ra cho tới lúc nó tới được tiến trình gọi là thời gian chờ xử lý. Thông thường một signal đang chờ xử lý sẽ được gửi tới tiến trình ngay khi tiến trình lập lịch cho hành động tiếp theo hoặc ngay lập tức nếu tiến trình đang chạy. Tuy nhiên một signal có thể bị chặn lại bằng cách thêm nó vào *signal mask* của tiến trình. Nếu signal được sinh ra trong khi nó bị chặn, nó tiếp tục đợi xử lý cho đến khi nó không bị chặn nữa.

# Luồng ( Thread )

Trong Linux, mỗi tiến trình có thể có nhiều luồng. Có thể coi các luồng như là một tập các tiến trình mà chia sẻ chung bộ nhớ ảo cũng như một lượng lớn các thuộc tính khác. Mỗi luồng thực hiện mã chương trình giống nhau và chia sẻ chung vùng dữ liệu và heap. Tuy nhiên, mỗi luồng cũng có stack của riêng nó để chứa các biến địa phương và thông tin liên kết ghi gọi hàm.

Các luồng có thể giao tiếp với nhau thông qua biến toàn cục. Các luồng có thể giao tiếp hoặc đồng bộ với nhau thông qua IPC hoặc các cơ chế động bộ với tiến trình mô tả ở trên.

Thuận lợi chính của việc sử dụng luồng là chúng dễ dàng chia sẻ dữ liệu với nhau và với một vài giải thuật sử dụng đa luồng sẽ đem lại hiệu quả cao hơn và dẽ dàng thiết lập hơn so với đa tiến trình.