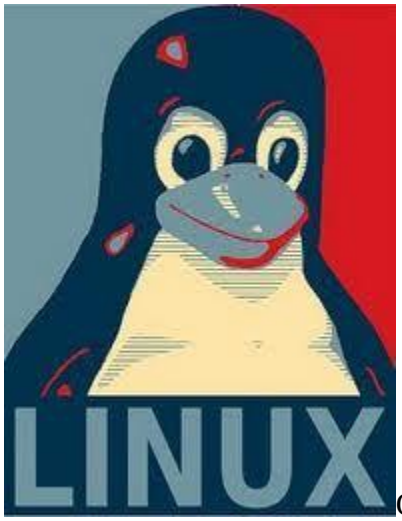
Quản lý các Shared Library trong Linux

- 1 Comment



Các gói phụ thuộc (dependency) là một thành phần thường gặp trong các chương trình máy tính ngày nay. Hôm nay, tôi sẽ đề cập tới shared library – một loại dependency cho các phần mềm trên Linux: nó là gì, quản lý nó ra sao?

1- Shared Library và Dynamic Linking

Library là file chứa các đoạn mã lệnh và dữ liệu được tổ chức thành các hàm (*subroutine*), các lớp (*class*) nhằm cung cấp dịch vụ, chức năng nào đó cho các chương trình chạy trên máy tính.

Library gồm 3 loại: **Static, Dynamic và Shared**. Thường thì các library ở dạng mã nhị phân, không phải dạng văn bản thuần túy (*plain text*) – là các ký tự mà con người có thể đọc hiểu được.

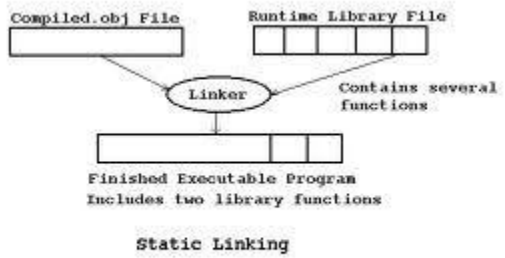
Khi biên dịch 1 chương trình đang ở dạng các file *source code* (gồm tập các câu lệnh, khai báo được viết bằng 1 ngôn ngữ lập trình cấp cao như C/C++, Java…) sang dạng *executable* (hay binary – tập các mã máy nhị phân mà chỉ có CPU mới hiểu được) thì nhiều hàm chức năng của chương trình được liên kết từ các library. Quá trình biên dịch này do bộ biên dịch (*Compiler*) đảm nhiệm.

Ví dụ, nếu chương trình của bạn có sử dụng hàm *print()*, thì bạn không cần cung cấp chi tiết mã lệnh của hàm print() này, nhưng phải đảm bảo rằng trên máy đã có sẵn 1 file library A nào đó chứa nội dung của hàm này. Khi Compiler cần liên kết đoạn mã cho hàm print(), nó tìm đoạn mã đó trong library A kia và sao chép nó vào chương trình.

Một chương trình đã được biên dịch, chứa đoạn mã trong các library và được lưu trữ ở bộ nhớ ngoài (HDD, USB…) được coi là được liên kết tĩnh (**Static Linking**) bởi vì khi chạy nó hoàn toàn độc lập, không còn phụ thuộc vào sự tồn tại của các library chứa đoạn mã đó nữa . “Chương trình được liên kết tĩnh” có 1 số điểm hạn chế như:

- Chương trình sẽ "phình to" hơn về kích thước chiếm dụng trên bộ nhớ ngoài do phải bao gồm đoạn mã của library được liên kết trong chương trình.

- Gây ra sự lãng phí bộ nhớ RAM nếu nhiều chương trình đang chạy đồng thời chứa cùng đoạn mã giống nhau trong library.

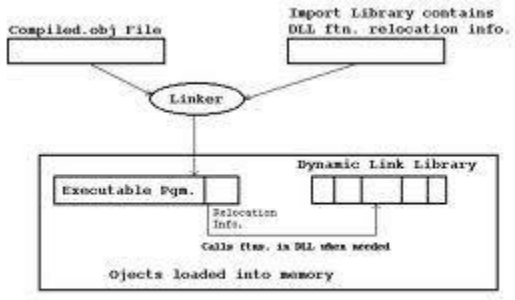


Static Linking

Để khắc phục 2 nhược điểm trên, nhiều chương trình được liên kết động (**Dynamic Linking**) tức là:

- Bản thân các chương trình này khi được lưu trữ ở bộ nhớ ngoài không chứa các đoạn mã trong library mà chỉ chứa khai báo tham khảo tới đoạn mã đó. Điều này giúp giảm kích cỡ của chương trình.

- Khác với static linking, việc liên kết tới library diễn ra tại thời điểm biên dịch. Ở dynamic linking, việc liên kết giữa các file thực thi (ở dạng binary) của chương trình với library diễn ra tại *thời điểm chạy chương trình* (runtime). Quá trình gắn kết này do bộ *Linker* đảm nhiệm giúp cho phép nhiều chương trình sử dụng chung library trong bộ nhớ.



Dynamic Linking

Các file library được liên kết động và được dùng chung bởi nhiều ứng dụng được gọi là **shared library**. Trên Windows các file này có phần mở rộng là **.dll** , còn Linux là các file **.so**

2- Xác định các shared library của 1 chương trình

Bất kỳ chương trình nào sử dụng dynamic linking đều yêu cầu 1 vài shared library có trên hệ thống. Nếu các library cần thiết không được tìm thấy (hoặc không tồn tại), khi chạy chương trình sẽ đưa ra thông báo lỗi.

Ví dụ, nếu bạn thử chạy 1 ứng dụng viết cho môi trường đồ họa GNOME nhưng hệ thống lại chưa có bộ thư viện GTK+ thì …:”> Đơn giản, bạn chỉ cần cài đặt đủ thư viện theo yêu cầu sẽ giải quyết được vấn đề.

Tiện ích **ldd** sẽ giúp bạn xác định các library cần thiết cho 1 chương trình.

**ldd** *program1, program2…*

Lệnh trên sẽ hiển thị các shared library cho các chương trình program1, program2…. Kết quả cho biết tên của library cùng với vị trí bạn cần cần đặt nó vào cây thư mục.

VD:

**ldd** /bin/bash

/bin/bash:
libtermcap.so.2 => /lib/libtermcap.so.2 (0×40018000)

libc.so.6 => /lib/libc.so.6 (0x4001c000)

/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0×40000000)

3- Tạo chỉ mục tìm kiếm (index) cho các shared library

Khi các chương trình ở dạng executable có sử dụng dynamic linking được chạy, thì tiện ích **ld.so** sẽ chịu trách nhiệm tìm kiếm và nạp vào bộ nhớ các shared library cần thiết cho chương trình đó. Nếu ld.so không thể tìm thấy các library đó thì chương trình sẽ gặp lỗi và không thể chạy được.

Thường thì các library được đặt trong các thư mục như */lib, /usr/lib, /usr/local/lib*. Để hướng dẫn cho ld.so tìm kiếm library trong các thư mục này cũng như là các thư mục khác thì có 2 cách:

a) Đơn giản nhất, bạn thêm danh sách các thư mục đó vào biên môi trường shell là *LD_LIBRARY_PATH*. Tuy nhiên, cách nay có thể không thích hợp đối với các system library, vì có thể người dùng sẽ chỉnh sửa sai biến LD_LIBRARY_PATH này.

b) Cách còn lại là tạo index gồm tên các library và thư mục lưu trữ chúng. File **/etc/ld.so.cache** chứa thông tin index này. Đây là file nhị phân, vì thế ld.so có thể nhanh chóng đọc nội dung của file này.

Để thêm mới index của library vào file cache trên, đầu tiên bạn thêm thư mục chứa library đó vào file **/etc/ld.so.conf**, đây là file cấu hình chứa các thư mục sẽ được tạo index bởi tiện ích **ldconfig**. Sau đó, chạy lệnh ldconfig với cú pháp như sau:

**ldconfig** [options] *lib_dirs*

Chỉ có 2 option cho lệnh này

-p: chỉ hiển thị nội dung hiện tại của cache, không tạo lại cache.

-v: hiển thị quá trình thực hiện việc tạo lại cache.

ldconfig sẽ cập nhật cho cache (file ld.so.cache). Thông tin cập nhật sẽ là index của các shared library nằm trong các thư mục *lib_dirs* (được chỉ ra ở dòng lệnh), các thư mục hệ thống là /usr/lib, /lib và các thư mục có trong file /etc/ld.so.conf.

+ Ví dụ 1: Để xem nội dung của ld.so.cache:

 # ldconfig -p

144 libs found in cache '/etc/ld.so.cache'

libz.so.1 (libc6) => /usr/lib/libz.so.1

…

+ Ví dụ 2: Tìm kiếm một library trong cache

 # ldconfig -p | grep *ncurses*

libncurses.so.5 (libc6) => /usr/lib/libncurses.so.5

+ Ví dụ 3: Tạo lại cache

 # ldconfig

Mỗi khi có sự thay đổi trong các thư mục chứa library, bạn nên chạy lại lệnh ldconfig này để đảm bảo cache luôn được cập nhật.