Liên kết cứng

Lệnh gọi hệ thống link() , một trong những lệnh gọi hệ thống Unix gốc và hiện được chuẩn hóa bởi POSIX, tạo ra một liên kết mới cho một tệp hiện có:

```
#include <unistd.h>

int liên kết (const char *oldpath, const char *newpath);
```

Một lệnh gọi thành công đến link( ) sẽ tạo một liên kết mới theo đường dẫn newpath cho tệp oldpath hiện có, sau đó trả về 0. Sau khi hoàn tất, cả oldpath và newpath đều tham chiếu đến cùng một tệp—trên thực tế, không có cách nào để biết đâu là liên kết "gốc".

Khi thất bại, lệnh gọi trả về -1 và đặt errno thành một trong những giá trị sau:

EACCESS

Quá trình gọi không có quyền tìm kiếm cho một thành phần trong oldpath hoặc quá trình gọi không có quyền ghi cho thư mục chứa newpath.

EEXIST

newpath đã tồn tại—link( ) sẽ không ghi đè lên mục thư mục hiện có.

EFAULT

oldpath hoặc newpath là con trỏ không hợp lệ.

EIO

Đã xảy ra lỗi I/O nội bộ (thật tệ!).

ELOOP

Có quá nhiều liên kết tượng trưng khi giải quyết oldpath hoặc newpath.

EMLINK

Inode được oldpath trỏ tới đã có số lượng liên kết tối đa trỏ tới nó.

ENAMETOOLONG

oldpath hoặc newpath quá dài.

ENOENT

Không tồn tại thành phần nào trong oldpath hoặc newpath .

ENOMEM

Không có đủ bộ nhớ để hoàn tất yêu cầu.

ENOSPC

Thiết bị chứa newpath không có chỗ cho mục nhập thư mục mới.

ENOTDIR

Một thành phần trong oldpath hoặc newpath không phải là một thư mục.

EPERM

Hệ thống tập tin chứa newpath không cho phép tạo liên kết cứng mới hoặc oldpath là một thư mục.

EROFS

newpath nằm trên hệ thống tập tin chỉ đọc.

EXDEV

newpath và oldpath không nằm trên cùng một hệ thống tập tin được gắn kết. (Linux cho phép một hệ thống tập tin duy nhất được gắn kết ở nhiều nơi, nhưng ngay cả trong trường hợp này, không thể tạo liên kết cứng qua các điểm gắn kết.)

Ví dụ này tạo một mục thư mục mới, pirate, ánh xạ tới cùng một inode (và do đó là cùng một tệp) với file privateer hiện có, cả hai đều nằm trong /home/kidd:

```
int ret;

/*
 * tạo một mục thư mục mới, * '/home/kidd/
 privateer', trỏ đến * cùng một inode với '/home/kidd/
 pirate' */

ret = liên kết ("/home/kidd/privateer", /home/kidd/pirate"); nếu
(ret)
      perror ("liên kết");
```

Liên kết tượng trưng

Liên kết tượng trưng, còn được gọi là liên kết tượng trưng hoặc liên kết mềm, tương tự như liên kết cứng ở chỗ cả hai đều trỏ đến các tệp trong hệ thống tệp. Tuy nhiên, liên kết tượng trưng khác ở chỗ nó không chỉ là một mục nhập thư mục bổ sung mà là một loại tệp đặc biệt hoàn toàn. Tệp đặc biệt này chứa tên đường dẫn cho một tệp khác , được gọi là tar-get của liên kết tượng trưng. Khi chạy, khi đang chạy, hạt nhân thay thế tên đường dẫn này cho tên đường dẫn của liên kết tượng trưng (trừ khi sử dụng các phiên bản l khác nhau của các lệnh gọi hệ thống, chẳng hạn như lstat( ), hoạt động trên chính liên kết đó chứ không phải mục tiêu). Do đó, trong khi một liên kết cứng không thể phân biệt được với một liên kết cứng khác đến cùng một tệp, thì rất dễ để phân biệt giữa liên kết tượng trưng và tệp mục tiêu của nó.

Liên kết tượng trưng có thể là tương đối hoặc tuyệt đối. Nó cũng có thể chứa thư mục dot đặc biệt đã thảo luận trước đó, đề cập đến thư mục mà nó nằm trong đó, hoặc thư mục dot-dot, đề cập đến thư mục cha của thư mục này.

Liên kết mềm, không giống như liên kết cứng, có thể mở rộng hệ thống tập tin. Trên thực tế, chúng có thể trỏ đến bất kỳ đâu!

Liên kết tượng trưng có thể trỏ đến các tệp hiện hữu (thực hành phổ biến) hoặc đến các tệp không tồn tại. Loại liên kết sau được gọi là liên kết tượng trưng lơ lửng. Đôi khi, liên kết tượng trưng lơ lửng là không mong muốn—chẳng hạn như khi mục tiêu của liên kết bị xóa, nhưng không phải liên kết tượng trưng—nhưng đôi khi, chúng là cố ý.

Hệ thống yêu cầu tạo liên kết tượng trưng rất giống với liên kết cứng:

```
#include <unistd.h>

int symlink (const char *oldpath, const char *newpath);
```

Gọi thành công tới symlink() sẽ tạo liên kết tượng trưng newpath trỏ tới tar-get oldpath, sau đó trả về 0.

Khi xảy ra lỗi, symlink() trả về -1 và đặt errno thành một trong những giá trị sau:

TRUY CẬP
: Tiến trình gọi không có quyền tìm kiếm cho một thành phần trong oldpath hoặc tiến trình gọi không có quyền ghi cho thư mục chứa newpath.

TỒN TẠI
: newpath đã tồn tại—symink() sẽ không ghi đè lên mục thư mục hiện có.

MẶC ĐỊNH
: oldpath hoặc newpath là con trỏ không hợp lệ.

EIO
: Đã xảy ra lỗi I/O nội bộ (thật tệ!).

Vòng lặp
: Có quá nhiều liên kết tượng trưng khi giải quyết oldpath hoặc newpath.

Liên kết EM
: Inode được oldpath trỏ tới đã có số lượng liên kết tối đa trỏ tới nó.

ENAMETOOLONG
: oldpath hoặc newpath quá dài.

ENOENT
: Không tồn tại thành phần nào trong oldpath hoặc newpath .

ENOMEM
: Không có đủ bộ nhớ để hoàn tất yêu cầu.

ENOSPC
: Thiết bị chứa newpath không có chỗ cho mục nhập thư mục mới.

ENOTDIR
: Một thành phần trong oldpath hoặc newpath không phải là một thư mục.

EPERM
: Hệ thống tập tin chứa newpath không cho phép tạo liên kết tượng trưng mới.

EROFS
: newpath nằm trên hệ thống tập tin chỉ đọc.

Đoạn mã này giống với ví dụ trước của chúng tôi, nhưng nó tạo ra /home/kidd/pirate như một liên kết tượng trưng (khác với liên kết cứng) đến /home/kidd/privateer:

```
int ret;

/*
 * tạo một liên kết tượng trưng,
```

```
 * '/home/kidd/privateer', trỏ đến '/home/
 *
   kidd/pirate' */

ret = liên kết tượng trưng ("/home/kidd/privateer", "/home/kidd/pirate"); nếu
(ret)
      perror ("liên kết tượng trưng");
```

Ngược lại

với liên kết là hủy liên kết, tức là xóa tên đường dẫn khỏi hệ thống tập tin.

Một lệnh gọi hệ thống duy nhất, unlink( ) , xử lý tác vụ này:

```
#include <unistd.h>

int hủy liên kết (const char *tên đường dẫn);
```

Một lệnh gọi thành công tới unlink() sẽ xóa pathname khỏi hệ thống tệp và trả về 0. Nếu tên đó là tham chiếu cuối cùng tới tệp, tệp sẽ bị xóa khỏi hệ thống tệp. Tuy nhiên, nếu một tiến trình mở tệp, hạt nhân sẽ không xóa tệp khỏi hệ thống tệp cho đến khi tiến trình đó đóng tệp. Khi không có tiến trình nào mở tệp, tệp sẽ bị xóa.

Nếu pathname tham chiếu đến một liên kết tượng trưng thì liên kết đó, chứ không phải mục tiêu, sẽ bị hủy.

Nếu pathname tham chiếu đến một loại tệp đặc biệt khác, chẳng hạn như thiết bị, FIFO hoặc ổ cắm, thì tệp đặc biệt đó sẽ bị xóa khỏi hệ thống tệp, nhưng các tiến trình có tệp đang mở vẫn có thể tiếp tục sử dụng tệp đó.

Khi xảy ra lỗi, unlink() trả về -1 và đặt errno thành một trong các mã lỗi sau:

EACCESS

Tiến trình gọi không có quyền ghi vào thư mục cha của pathname hoặc tiến trình gọi không có quyền tìm kiếm thành phần trong pathname.

Đường

dẫn EFAULT là một con trỏ không hợp lệ.

EIO

Đã xảy ra lỗi I/O (thật tệ!).

Đường

dẫn EISDIR đề cập đến một thư mục.

ELOOP

Có quá nhiều liên kết tượng trưng khi duyệt pathname.

Đường dẫn

ENAMETOOLONG quá dài.

ENOENT

Một thành phần trong pathname không tồn tại.

ENOMEM

Không có đủ bộ nhớ để hoàn tất yêu cầu.

ENOTDIR

Một thành phần trong pathname không phải là một thư mục.

EPERM

Hệ thống không cho phép hủy liên kết tập tin.

Đường

dẫn EROFS nằm trong hệ thống tập tin chỉ đọc.

unlink() không xóa thư mục. Đối với điều đó, các ứng dụng nên sử dụng rmdir( ) mà chúng ta đã thảo luận trước đó (xem “Xóa thư mục”).

Để giảm thiểu việc phá hủy tùy ý bất kỳ loại tệp nào, ngôn ngữ C cung cấp hàm remove() :

```
#include <stdio.h>

int xóa (const char *path);
```

Gọi thành công remove() sẽ xóa path khỏi hệ thống tập tin và trả về 0. Nếu path là một tập tin, remove() sẽ gọi unlink( ); nếu path là một thư mục, remove() sẽ gọi rmdir( ).

Khi xảy ra lỗi, remove() trả về -1 và đặt errno thành bất kỳ mã lỗi hợp lệ nào được đặt bởi unlink() và rmdir( ), tùy trường hợp.

Sao chép và di chuyển tập tin

Hai trong số các tác vụ thao tác tệp cơ bản nhất là sao chép và di chuyển tệp, thường được thực hiện thông qua các lệnh cp và mv . Ở cấp độ hệ thống tệp, sao chép là hành động sao chép nội dung của tệp đã cho dưới một tên đường dẫn mới. Điều này khác với việc tạo liên kết cứng mới đến tệp ở chỗ các thay đổi đối với một tệp sẽ không ảnh hưởng đến tệp kia—tức là, hiện có hai bản sao riêng biệt của tệp, dưới (ít nhất) hai mục nhập thư mục khác nhau. Ngược lại , di chuyển là hành động đổi tên mục nhập thư mục mà tệp được đặt. Hành động này không dẫn đến việc tạo ra một

sao chép.

Sao chép Mặc

dù có thể gây ngạc nhiên cho một số người, Unix không bao gồm lệnh gọi hệ thống hoặc thư viện để tạo điều kiện thuận lợi cho việc sao chép tệp và thư mục. Thay vào đó, các tiện ích như cp hoặc trình quản lý tệp Nautilus của GNOME thực hiện các tác vụ này theo cách thủ công.

Khi sao chép một tệp src vào một tệp có tên dst, các bước thực hiện như sau:

1. Mở src.
2. Mở dst, tạo nó nếu nó không tồn tại và cắt bớt nó thành độ dài bằng không nếu nó tồn tại

hiện hữu.

3. Đọc một đoạn src vào bộ nhớ.
4. Ghi khối vào dst.
5. Tiếp tục cho đến khi toàn bộ src đã được đọc và ghi vào dst.
6. Đóng dst.
7. Đóng src.

Nếu sao chép một thư mục, thư mục riêng lẻ và mọi thư mục con sẽ được tạo thông qua mkdir( ); sau đó mỗi tệp trong đó sẽ được sao chép riêng lẻ.

Di chuyển

Không giống như sao chép, Unix cung cấp lệnh gọi hệ thống để di chuyển tệp. Chuẩn ANSI C đã giới thiệu lệnh gọi tệp và POSIX đã chuẩn hóa lệnh này cho cả tệp và thư mục:

```
#include <stdio.h>

int đổi tên (const char *oldpath, const char *newpath);
```

Một lệnh gọi thành công đến lệnh đổi tên () sẽ đổi tên đường dẫn oldpath thành newpath. Nội dung và inode của tệp vẫn giữ nguyên. Cả oldpath và newpath phải nằm trên cùng một hệ thống tệp*; nếu không, lệnh gọi sẽ không thành công. Các tiện ích như mv phải xử lý trường hợp này bằng cách sao chép và hủy liên kết.

Khi thành công, lệnh đổi tên() trả về 0 và tệp đã từng được tham chiếu bởi oldpath giờ được tham chiếu bởi newpath. Khi thất bại, lệnh gọi trả về -1, không chạm đến oldpath hoặc newpath và đặt errno thành một trong các giá trị sau:

TRUY CẬP

Quá trình gọi thiếu quyền ghi cho phần tử cha của oldpath hoặc newpath, quyền tìm kiếm cho một thành phần của oldpath hoặc newpath hoặc quyền ghi cho oldpath trong trường hợp oldpath là một thư mục. Trường hợp cuối cùng là một vấn đề vì đổi tên () phải cập nhật .. trong oldpath nếu đó là một thư mục.

EBUSY

oldpath hoặc newpath là điểm gắn kết.

EFAULT

oldpath hoặc newpath là con trỏ không hợp lệ.

EINVAL

newpath nằm trong oldpath, do đó, đổi tên một trong hai sẽ biến oldpath thành một thư mục con của chính nó.

EISDR

newpath tồn tại và là một thư mục, nhưng oldpath không phải là một thư mục.

* Mặc dù Linux cho phép bạn gắn một thiết bị tại nhiều điểm trong cấu trúc thư mục, bạn vẫn không thể đổi tên từ một trong những điểm gắn kết này sang điểm gắn kết khác, ngay cả khi chúng được hỗ trợ bởi cùng một thiết bị.

ELOOP

Khi giải quyết oldpath hoặc newpath, có quá nhiều liên kết tượng trưng được phát hiện.

EMLINK

oldpath đã có số lượng liên kết tối đa đến chính nó hoặc oldpath là một thư mục và newpath đã có số lượng liên kết tối đa đến chính nó.

ENAMETOOLONG

oldpath hoặc newpath quá dài.

ENOENT

Thành phần trong oldpath hoặc newpath không tồn tại hoặc chỉ là một liên kết tượng trưng lơ lửng.

ENOMEM

Không có đủ bộ nhớ hạt nhân để hoàn tất yêu cầu.

ENOSPC

Thiết bị không đủ dung lượng để hoàn tất yêu cầu.

ENOTDIR

Một thành phần (ngoại trừ thành phần cuối cùng) trong oldpath hoặc newpath không phải là một thư mục, hoặc oldpath là một thư mục, còn newpath tồn tại nhưng không phải là một thư mục.

ENOTEMPTY

newpath là một thư mục và không trống.

EPERM

Có ít nhất một đường dẫn được chỉ định trong các đối số tồn tại, thư mục cha có bit cố định được đặt, ID người dùng hiệu quả của quy trình gọi không phải là ID người dùng của tệp cũng không phải của tệp cha và quy trình không được cấp đặc quyền.

EROFS

Hệ thống tập tin được đánh dấu là chỉ đọc.

EXDEV

oldpath và newpath không nằm trên cùng một hệ thống tập tin.

Bảng 7-1 xem xét kết quả của việc di chuyển đến và đi từ các loại tệp khác nhau.

Bảng 7-1. Tác động của việc di chuyển đến và đi từ các loại tệp khác nhau

	Điểm đến là một tập tin	Điểm đến là một thư mục	Điểm đến là một liên kết	Điểm đến không tồn tại
Nguồn là một tập tin	Điểm đến được ghi đè bởi nguồn.	Thất bại với EISDIR.	Tập tin được đổi tên và đích đến bị ghi đè.	Tập tin đã được đổi tên.
Nguồn là một thư mục	Thất bại với ENOTDIR.	Nguồn được đổi tên thành quốc gia đích nếu quốc gia đích trống; nếu không thì sẽ không có ENOTEMPTY .	Thư mục được đổi tên và đích đến bị ghi đè.	Thư mục đã được đổi tên.

Bảng 7-1. Tác động của việc di chuyển đến và đi từ các loại tệp khác nhau (tiếp theo)

	Điểm đến là một tập tin	Điểm đến là một thư mục	Điểm đến là một liên kết	Điểm đến không tồn tại
Nguồn là một liên kết	Liên kết được đổi tên và đích đến bị ghi đè.	Thất bại với EISDIR.	Liên kết được đổi tên và đích đến bị ghi đè.	Liên kết đã được đổi tên.
Nguồn không tồn tại	Thất bại với ENOENT.	Thất bại với ENOENT.	Thất bại với ENOENT.	Thất bại với ENOENT.

Đối với tất cả các trường hợp này, bất kể loại nào, nếu nguồn và đích nằm trên các hệ thống tệp khác nhau, lệnh gọi sẽ không thành công và trả về EXDEV.

Các nút thiết bị

Các nút thiết bị là các tệp đặc biệt cho phép các ứng dụng giao tiếp với trình điều khiển thiết bị. Khi một ứng dụng thực hiện Unix I/O thông thường—mở, đóng, đọc, ghi, v.v.—trên một nút thiết bị, thì hạt nhân không xử lý các yêu cầu đó như I/O tệp thông thường. Thay vào đó, hạt nhân chuyển các yêu cầu đó đến trình điều khiển thiết bị. Trình điều khiển thiết bị xử lý hoạt động I/O và trả về kết quả cho người dùng. Các nút thiết bị cung cấp tính trừu tượng của thiết bị để các ứng dụng không cần phải quen thuộc với các thông số kỹ thuật của thiết bị hoặc thậm chí là làm chủ các giao diện đặc biệt. Thật vậy, các nút thiết bị là cơ chế tiêu chuẩn để truy cập phần cứng trên các hệ thống Unix. Các thiết bị mạng là ngoại lệ hiếm hoi và trong suốt lịch sử của Unix, một số người đã lập luận rằng ngoại lệ này là một sai lầm. Thật vậy, có một vẻ đẹp tao nhã trong việc thao tác tất cả phần cứng của máy bằng các lệnh gọi read( ), write( ) và mmap() .

Làm thế nào để hạt nhân xác định trình điều khiển thiết bị mà nó phải chuyển giao yêu cầu? Mỗi nút thiết bị được gán hai giá trị số, được gọi là số chính và số phụ. Các số chính và số phụ này ánh xạ tới một trình điều khiển thiết bị cụ thể được tải vào hạt nhân. Nếu một nút thiết bị có số chính và số phụ không tương ứng với trình điều khiển thiết bị trong hạt nhân—điều này đôi khi xảy ra, vì nhiều lý do khác nhau—một yêu cầu open() trên nút thiết bị trả về -1 với errno được đặt thành ENODEV. Chúng ta nói rằng các nút thiết bị như vậy nằm trước các thiết bị không tồn tại.

Các nút thiết bị đặc biệt Có một

số nút thiết bị trên tất cả các hệ thống Linux. Các nút thiết bị này là một phần của môi trường phát triển Linux và sự hiện diện của chúng được coi là một phần của ABI Linux.

Thiết bị null có số chính là 1 và số phụ là 3. Thiết bị này nằm ở /dev/null.
Tệp thiết bị phải do root sở hữu và có thể đọc và ghi được bởi tất cả người dùng. Kernel sẽ âm thầm loại bỏ tất cả các yêu cầu ghi vào thiết bị. Tất cả các yêu cầu đọc vào tệp sẽ trả về end-of-file (EOF).

Thiết bị zero nằm tại /dev/zero và có major là 1 và minor là 5. Giống như thiết bị null, kernel sẽ âm thầm loại bỏ các lệnh ghi vào thiết bị zero. Đọc từ thiết bị sẽ trả về một luồng byte null vô hạn.

Thiết bị đầy đủ, với major là 1 và minor là 7, nằm tại /dev/full. Cũng giống như thiết bị zero, các yêu cầu đọc trả về ký tự null (\0). Tuy nhiên, các yêu cầu ghi luôn kích hoạt lỗi ENOSPC , biểu thị rằng thiết bị cơ bản đã đầy.

Các thiết bị này có nhiều mục đích khác nhau. Chúng hữu ích để kiểm tra cách ứng dụng xử lý các trường hợp góc và vấn đề—ví dụ như hệ thống tệp đầy đủ. Vì các thiết bị null và zero bỏ qua các lần ghi, chúng cũng cung cấp một cách không tốn chi phí để loại bỏ I/O không mong muốn.

Máy phát số ngẫu nhiên

Các thiết bị tạo số ngẫu nhiên của hạt nhân nằm ở /dev/random và /dev/urandom.
Chúng có số chính là 1, số phụ là 8 và 9.

Bộ tạo số ngẫu nhiên của hạt nhân thu thập nhiễu từ trình điều khiển thiết bị và các nguồn khác, và hạt nhân nối lại với nhau và băm một chiều nhiễu thu thập được. Sau đó, kết quả được lưu trữ trong một nhóm entropy. Hạt nhân giữ ước tính về số bit entropy trong nhóm.

Đọc từ /dev/random trả về entropy từ nhóm này. Kết quả phù hợp để gieo hạt cho các trình tạo số ngẫu nhiên, thực hiện tạo khóa và các tác vụ khác yêu cầu entropy mạnh về mặt mật mã.

Về mặt lý thuyết, một đối thủ có thể lấy đủ dữ liệu từ nhóm entropy và phá vỡ thành công hàm băm một chiều có thể có được kiến thức về trạng thái của phần còn lại của nhóm entropy. Mặc dù một cuộc tấn công như vậy hiện chỉ là một khả năng lý thuyết—không có cuộc tấn công nào như vậy được công khai biết đến là đã xảy ra—hạt nhân phản ứng với khả năng này bằng cách giảm ước tính của nó về lượng entropy trong nhóm với mỗi yêu cầu đọc. Nếu ước tính đạt đến số không, lệnh đọc sẽ bị chặn cho đến khi hệ thống tạo ra nhiều entropy hơn và ước tính entropy đủ lớn để đáp ứng lệnh đọc.

/dev/urandom không có thuộc tính này; các lần đọc từ thiết bị sẽ thành công ngay cả khi ước tính entropy của hạt nhân không đủ để hoàn tất yêu cầu. Vì chỉ những ứng dụng an toàn nhất—chẳng hạn như tạo khóa để trao đổi dữ liệu an toàn trong GNUPrivacy Guard—mới quan tâm đến entropy mạnh về mặt mật mã, nên hầu hết các ứng dụng nên sử dụng /dev/urandom chứ không phải /dev/random. Các lần đọc vào cái sau có khả năng bị chặn trong một thời gian rất dài nếu không có hoạt động I/O nào xảy ra cung cấp dữ liệu cho nhóm entropy của hạt nhân. Điều này không phải là hiếm trên các máy chủ không có đĩa, không có đầu.

Giao tiếp ngoài băng tần

Mô hình tệp Unix rất ấn tượng. Chỉ với các thao tác đọc và ghi đơn giản, Unix tóm tắt hầu như mọi hành động có thể thực hiện trên một đối tượng. Tuy nhiên, đôi khi, các lập trình viên cần giao tiếp với một tệp bên ngoài luồng dữ liệu chính của nó. Ví dụ, hãy xem xét một thiết bị cổng nối tiếp. Đọc từ thiết bị sẽ đọc từ phần cứng ở đầu xa của cổng nối tiếp; ghi vào thiết bị sẽ gửi dữ liệu đến phần cứng đó. Một quy trình sẽ đọc một trong các chân trạng thái đặc biệt của cổng nối tiếp như thế nào, chẳng hạn như tín hiệu thiết bị đầu cuối dữ liệu sẵn sàng (DTR)? Hoặc, một quy trình sẽ thiết lập tính chẵn lẻ của cổng nối tiếp như thế nào?

Câu trả lời là sử dụng lệnh gọi hệ thống ioctl() . ioctl( ) là viết tắt của I/O control, cho phép giao tiếp ngoài băng tần:

```
#include <sys/ioctl.h>

int ioctl (int fd, int yêu cầu, ...);
```

Lệnh gọi hệ thống yêu cầu hai tham số:

fd
: Mô tả tập tin của một tập tin.

yêu cầu
: Giá trị mã yêu cầu đặc biệt, được xác định trước và thống nhất bởi hạt nhân và quy trình, biểu thị thao tác nào sẽ thực hiện trên tệp được fd tham chiếu.

Nó cũng có thể nhận một hoặc nhiều tham số tùy chọn không có kiểu (thường là số nguyên không dấu hoặc con trỏ) để truyền vào hạt nhân.

Chương trình sau đây sử dụng yêu cầu CDROMEJECT để đẩy khay phương tiện ra khỏi thiết bị CD-ROM, mà người dùng cung cấp làm đối số đầu tiên trên dòng lệnh của chương trình. Do đó, chương trình này hoạt động tương tự như lệnh đẩy chuẩn :

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <linux/cdrom.h>
#include <stdio.h>

int chính (int argc, char *argv[]) {

        int fd, ret;

        nếu (argc < 2)
                { fprintf (stderr,
                           "cách sử dụng: %s <thiết bị cần đẩy
                           ra>\n",
                argv[0]); trả về 1;
        }
```

```
        /*
         *
            Mở thiết bị CD-ROM, chỉ đọc. O_NONBLOCK * cho kernel biết rằng
         chúng ta muốn mở thiết bị ngay cả khi không có phương tiện nào trong ổ
         *
            đĩa. */

        fd = mở (argv[1], O_RDONLY | O_NONBLOCK); nếu (fd
        < 0) { perror
                  ("mở"); trả về 1;

        }

        /* Gửi lệnh đẩy đĩa ra khỏi thiết bị CD-ROM. */ ret = ioctl
        (fd, CDROMEJECT, 0); if (ret) { perror
        ("ioctl");
                  return 1;

        }

        ret = đóng (fd);
        nếu (ret)
                  { perror ("đóng");
                  trả về 1;
        }

        trả về 0;
}
```

Yêu cầu CDROMEJECT là một tính năng của trình điều khiển thiết bị CD-ROM của Linux. Khi ker-nel nhận được yêu cầu ioctl () , nó sẽ tìm hệ thống tệp (trong trường hợp tệp thực) hoặc trình điều khiển thiết bị (trong trường hợp nút thiết bị) chịu trách nhiệm cho mô tả tệp được cung cấp và chuyển yêu cầu để xử lý. Trong trường hợp này, trình điều khiển thiết bị CD-ROM nhận được yêu cầu và đẩy ổ đĩa ra.

Ở phần sau của chương này, chúng ta sẽ xem xét ví dụ ioctl() sử dụng tham số tùy chọn để trả về thông tin cho tiến trình yêu cầu.

Giám sát sự kiện tệp Linux cung cấp một

giao diện, inotify, để giám sát các tệp—ví dụ, để xem khi nào chúng được di chuyển, đọc từ, ghi vào hoặc xóa. Hãy tưởng tượng rằng bạn đang viết một trình quản lý tệp đồ họa, chẳng hạn như Nautilus của GNOME. Nếu một tệp được sao chép vào một thư mục trong khi Nautilus đang hiển thị nội dung của nó, chế độ xem thư mục của trình quản lý tệp sẽ trở nên không nhất quán.

Một giải pháp là liên tục đọc lại nội dung của thư mục, phát hiện các thay đổi và cập nhật màn hình hiển thị. Điều này gây ra chi phí định kỳ và không phải là giải pháp sạch sẽ. Tệ hơn nữa, luôn có một cuộc đua giữa thời điểm tệp được xóa khỏi hoặc thêm vào thư mục và thời điểm trình quản lý tệp đọc lại thư mục.

Với inotify, kernel có thể đẩy sự kiện đến ứng dụng ngay khi sự kiện xảy ra. Ngay khi một tệp bị xóa, kernel có thể thông báo cho Nautilus. Nautilus, để phản hồi, có thể ngay lập tức xóa tệp đã xóa khỏi màn hình đồ họa của thư mục.

Nhiều ứng dụng khác cũng quan tâm đến các sự kiện tệp. Hãy xem xét một tiện ích sao lưu hoặc một công cụ lập chỉ mục dữ liệu. inotify cho phép cả hai chương trình này hoạt động theo thời gian thực: ngay khi một tệp được tạo, xóa hoặc ghi vào, các công cụ có thể cập nhật kho lưu trữ sao lưu hoặc chỉ mục dữ liệu. inotify

thay thế dnotify, một cơ chế giám sát tệp trước đó bằng một giao diện dựa trên tín hiệu cồng kềnh. Các ứng dụng luôn phải ưu tiên inotify hơn dnotify. inotify, được giới thiệu với kernel 2.6.13, linh hoạt và dễ sử dụng vì các hoạt động giống nhau mà các chương trình thực hiện trên các tệp thông thường—đáng chú ý là select() và poll( )—hoạt động với inotify. Chúng tôi chỉ đề cập đến inotify trong cuốn sách này.

Khởi tạo inotify Trước khi

một tiến trình có thể sử dụng inotify, tiến trình phải khởi tạo nó. Lệnh gọi hệ thống inotify_init() khởi tạo inotify và trả về một mô tả tệp biểu diễn thể hiện đã khởi tạo:

```
#include <inotify.h>

int inotify_init (không có giá trị);
```

Khi xảy ra lỗi, inotify_init() trả về -1 và đặt errno thành một trong các mã sau:

TỆP EM

Đã đạt đến giới hạn số lượng tối đa các phiên bản inotify cho mỗi người dùng.

ENFILE

Đã đạt đến giới hạn toàn hệ thống về số lượng tối đa các mô tả tệp.

ENOMEM

Không có đủ bộ nhớ để hoàn tất yêu cầu.

Hãy khởi tạo inotify để có thể sử dụng nó trong các bước tiếp theo:

```
số nguyên fd;

fd = inotify_init ();
nếu (fd == -1)
        { lỗi ("inotify_init");
        thoát (EXIT_FAILURE);
}
```

Đồng hồ

Sau khi một tiến trình khởi tạo inotify, nó sẽ thiết lập các watch. Watch, được biểu diễn bằng một watch descriptor, là một đường dẫn Unix chuẩn và một watch mask liên quan, cho biết kernel những sự kiện nào mà tiến trình quan tâm—ví dụ, đọc, ghi hoặc cả hai. inotify có thể theo dõi cả tệp và thư mục. Nếu theo dõi một thư mục, inotify sẽ báo cáo các sự kiện xảy ra trên chính thư mục đó và trên bất kỳ tệp nào nằm trong thư mục đó (nhưng không phải trên các tệp trong các thư mục con của thư mục được theo dõi—watch không đệ quy).

Thêm đồng hồ mới

Lệnh gọi hệ thống inotify_add_watch( ) sẽ thêm chức năng theo dõi sự kiện hoặc các sự kiện được mô tả bởi mask trên đường dẫn tệp hoặc thư mục tới phiên bản inotify được biểu diễn bởi fd:

```
#include <inotify.h>

int inotify_add_watch (int fd,
                       const char *path,
                       mặt nạ uint32_t);
```

Khi thành công, lệnh gọi trả về một mô tả đồng hồ mới. Khi thất bại, inotify_add_watch( ) trả về -1 và đặt errno thành một trong những giá trị sau:

TRUY CẬP

Không được phép truy cập đọc vào tệp được chỉ định bởi đường dẫn . Quy trình gọi phải có khả năng đọc tệp để thêm chế độ theo dõi vào đó.

EBADF

Mô tả tệp fd không phải là phiên bản inotify hợp lệ.

EFAULT

Đường dẫn con trỏ không hợp lệ.

EINVAL

Mặt nạ theo dõi, mặt nạ, không chứa sự kiện hợp lệ nào.

ENOMEM

Không có đủ bộ nhớ để hoàn tất yêu cầu.

ENOSPC

Đã đạt đến giới hạn về tổng số lượng giám sát inotify cho mỗi người dùng.

Xem mặt nạ

Mặt nạ theo dõi là OR nhị phân của một hoặc nhiều sự kiện inotify, được <inotify.h> định nghĩa:

IN_ACCESS
Tệp đã được đọc từ.

IN_MODIFY
Tệp đã được ghi vào.

IN_ATTRIB

Siêu dữ liệu của tệp (ví dụ: chủ sở hữu, quyền hoặc thuộc tính mở rộng) đã bị thay đổi.

IN_CLOSE_WRITE

Tệp đã được đóng và mở để ghi.

IN_CLOSE_NOWRITE

Tệp đã bị đóng và chưa được mở để ghi.

IN_OPEN

Tệp đã được mở.

IN_MOVED_FROM

Một tệp đã được di chuyển khỏi thư mục được theo dõi.

ĐÃ_DI_ĐẾN

Một tập tin đã được di chuyển vào thư mục được theo dõi.

IN_CREATE

Một tệp đã được tạo trong thư mục được theo dõi.

IN_DELETE

Một tập tin đã bị xóa khỏi thư mục được theo dõi.

IN_DELETE_SELF

Đối tượng được theo dõi đã bị xóa.

IN_MOVE_SELF

Đối tượng được theo dõi đã được di chuyển.

Các sự kiện sau đây cũng được định nghĩa, nhóm hai hoặc nhiều sự kiện thành một giá trị duy nhất:

IN_ALL_EVENTS Tất

cả các sự kiện hợp pháp.

IN_CLOSE

Tất cả các sự kiện liên quan đến việc đóng (hiện tại, cả IN_CLOSE_WRITE và IN_CLOSE_NOWRITE).

IN_MOVE

Tất cả các sự kiện liên quan đến di chuyển (hiện tại, cả IN_MOVED_FROM và IN_MOVED_TO).

Bây giờ, chúng ta có thể xem xét việc thêm một watch mới vào phiên bản inotify hiện có:

```
int wd;

wd = inotify_add_watch (fd, "/etc", ĐÃ TRUY CẬP | ĐÃ CHỈNH SỬA); nếu
(wd == -1) { lỗi
          ("inotify_add_watch"); thoát (THẤT
          BẠI_THOÁT);
}
```

Ví dụ này thêm một watch cho tất cả các lần đọc hoặc ghi trên thư mục /etc. Nếu bất kỳ tệp nào trong /etc được ghi vào hoặc đọc từ, inotify sẽ gửi một sự kiện đến mô tả tệp inotify, fd, cung cấp mô tả watch wd. Hãy xem cách inotify biểu diễn những
sự kiện.

Sự kiện inotify

Cấu trúc inotify_event , được định nghĩa trong <inotify.h>, biểu diễn các sự kiện inotify:

```
#include <inotify.h>

struct inotify_event { int
          wd; /* xem mô tả */ uint32_t mask; /* mặt nạ của các
          sự kiện */ uint32_t cookie; /* cookie duy nhất */
          uint32_t len; /* kích thước của trường 'name' */
          char name[]; /* tên kết thúc bằng null */

};
```

wd xác định mô tả giám sát, như được lấy bởi inotify_add_watch( ) và mask rep-resents các sự kiện. Nếu wd xác định một thư mục và một trong những sự kiện được giám sát đã xảy ra trên một tệp trong thư mục đó, name cung cấp tên tệp tương đối với thư mục. Trong trường hợp này, len khác không. Lưu ý rằng len không giống với độ dài chuỗi của name ; name có thể có nhiều hơn một ký tự null theo sau đóng vai trò là phần đệm để đảm bảo rằng inotify_event tiếp theo được căn chỉnh đúng cách. Do đó, bạn phải sử dụng len, chứ không phải strlen( ), để tính toán độ lệch của cấu trúc inotify_event tiếp theo trong một mảng.

Ví dụ, nếu wd biểu diễn /home/rlove và có mặt nạ là IN_ACCESS, và tệp /home/rlove/canon được đọc từ đó, thì name sẽ bằng canon và len sẽ ít nhất là 6.
Ngược lại, nếu chúng ta đang theo dõi /home/rlove/canon trực tiếp với cùng một mặt nạ, len sẽ là 0 và name sẽ là zero-length—bạn không được chạm vào nó. cookie được sử dụng

để liên kết hai sự kiện có liên quan nhưng không giao nhau. Chúng ta sẽ giải quyết vấn đề này trong phần tiếp theo.

Đọc sự kiện inotify

Việc lấy sự kiện inotify rất dễ dàng: bạn chỉ cần đọc từ mô tả tệp được liên kết với phiên bản inotify. inotify cung cấp một tính năng được gọi là slurping, cho phép bạn đọc nhiều sự kiện—nhiều nhất có thể trong bộ đệm được cung cấp cho read( )—với một yêu cầu đọc duy nhất. Do trường tên có độ dài thay đổi , đây là cách phổ biến nhất để đọc sự kiện inotify.

Ví dụ trước của chúng tôi đã khởi tạo một thể hiện inotify và thêm một watch vào thể hiện đó. Bây giờ, hãy đọc các sự kiện đang chờ xử lý:

```
char buf[BUF_LEN]_attribute_((aligned(4)));
ssize_t len, i = 0;

/* đọc BUF_LEN byte sự kiện */ len = đọc (fd,
buf, BUF_LEN);

/* lặp qua mọi sự kiện đọc cho đến khi không còn sự kiện
nào nữa */ while (i
          < len) { struct inotify_event
                   *event = (struct inotify_event *) &buf[i];
```

```
            printf ("wd=%d mask=%d cookie=%d len=%d dir=%s\n", event-
                    >wd, event->mask, event-
                    >cookie, event->len, (event-
                    >mask & IN_ISDIR) ? "có" : "không");

            /* nếu có tên, in ra */ if (event->len)
            printf
                   ("name=%s\n", event->name);

            /* cập nhật chỉ mục đến điểm bắt đầu của sự kiện tiếp theo
            */ i += sizeof (struct inotify_event) + event->len;
    }
```

Bởi vì mô tả tệp inotify hoạt động giống như một tệp thông thường, nên các chương trình có thể theo dõi nó thông qua select( ), poll( ) và epoll( ). Điều này cho phép các tiến trình ghép kênh các sự kiện inotify với các tệp I/O khác từ một luồng duy nhất.

Sự kiện inotify nâng cao. Ngoài các sự kiện tiêu chuẩn, inotify có thể tạo ra các sự kiện khác sự kiện:

IN_IGNORED

Đồng hồ được biểu thị bằng wd đã bị xóa. Điều này có thể xảy ra vì người dùng đã xóa đồng hồ theo cách thủ công hoặc vì đối tượng được theo dõi không còn tồn tại nữa.

Chúng ta sẽ thảo luận về sự kiện này ở phần sau.

IN_ISDIR

Đối tượng bị ảnh hưởng là một thư mục. (Nếu không được đặt, đối tượng bị ảnh hưởng là một tệp.)

IN_Q_OVERFLOW Hàng

đợi inotify bị tràn. Kernel giới hạn kích thước của hàng đợi sự kiện để ngăn chặn việc sử dụng bộ nhớ kernel không giới hạn. Khi số lượng sự kiện đang chờ xử lý đạt đến một ít hơn mức tối đa, kernel sẽ tạo sự kiện này và thêm nó vào đuôi của hàng đợi. Không có sự kiện nào được tạo thêm cho đến khi hàng đợi được đọc, giảm kích thước của nó xuống dưới giới hạn.

IN_UNMOUNT

Thiết bị hỗ trợ đối tượng được theo dõi đã bị hủy gắn kết. Do đó, đối tượng không còn khả dụng nữa; hạt nhân sẽ xóa đồng hồ và tạo IN_IGNORED sự kiện.

Bất kỳ đồng hồ nào cũng có thể tạo ra những sự kiện này; người dùng không cần phải thiết lập chúng một cách rõ ràng.

Các lập trình viên phải coi mặt nạ như một mặt nạ bit của các sự kiện đang chờ xử lý. Do đó, không kiểm tra các sự kiện bằng cách sử dụng các thử nghiệm trực tiếp về tính tương đương:

```
    /* KHÔNG được làm điều này! */

    if (event->mask == IN_MODIFY)
            printf ("Tệp đã được ghi vào!\n"); else
    if (event->mask == IN_Q_OVERFLOW) printf
            ("Ồ, hàng đợi đã tràn!\n);
```

Thay vào đó, hãy thực hiện các thử nghiệm bitwise:

```
if (event->mask & IN_ACCESS)
        printf ("Tệp đã được đọc từ!\n"); if
(event->mask & IN_UNMOUNTED)
        printf ("Thiết bị sao lưu của tệp đã bị hủy gắn kết!
\n); if (event->mask &
        IN_ISDIR) printf ("Tệp là một thư mục!\n");
```

Liên kết các sự kiện di chuyển

Các sự kiện IN_MOVED_FROM và IN_MOVED_TO mỗi sự kiện chỉ biểu diễn một nửa của một lần di chuyển: sự kiện trước biểu diễn việc di chuyển khỏi một vị trí nhất định, trong khi sự kiện sau biểu diễn việc đến một vị trí mới. Do đó, để thực sự hữu ích cho một chương trình đang cố gắng theo dõi các tệp một cách thông minh khi chúng di chuyển xung quanh hệ thống tệp (hãy xem xét một trình lập chỉ mục với mục đích không lập chỉ mục lại các tệp đã di chuyển), các quy trình cần có khả năng liên kết hai sự kiện di chuyển với nhau.

Nhập trường cookie vào cấu trúc inotify_event .

Trường cookie , nếu khác không, sẽ chứa một giá trị duy nhất liên kết hai sự kiện với nhau. Hãy xem xét một tiến trình đang theo dõi /bin và /sbin. Giả sử /bin có mô tả theo dõi là 7 và / sbin có mô tả theo dõi là 8. Nếu tệp /bin/compass được di chuyển đến /sbin/compass, hạt nhân sẽ tạo ra hai sự kiện inotify.

Sự kiện đầu tiên sẽ có wd bằng 7, mask bằng IN_MOVED_FROM và tên la bàn. Sự kiện thứ hai sẽ có wd bằng 8, mask bằng IN_MOVED_TO và tên la bàn. Trong cả hai sự kiện, cookie sẽ giống nhau—ví dụ: 12.

Nếu một tệp được đổi tên, hạt nhân vẫn tạo ra hai sự kiện. wd giống nhau cho cả hai.

Lưu ý rằng nếu một tệp được di chuyển từ hoặc đến một thư mục không được theo dõi, quy trình sẽ không nhận được một trong các sự kiện tương ứng. Chương trình sẽ lưu ý rằng sự kiện thứ hai có cookie khớp sẽ không bao giờ đến.

Tùy chọn xem nâng cao

Khi tạo đồng hồ mới, bạn có thể thêm một hoặc nhiều giá trị sau vào mặt nạ để kiểm soát hành vi của đồng hồ:

IN_DONT_FOLLOW Nếu
: giá trị này được đặt và nếu mục tiêu của đường dẫn hoặc bất kỳ thành phần nào của nó là liên kết tượng trưng, liên kết sẽ không được theo dõi và inotify_add_watch() sẽ không thành công.

IN_MASK_ADD
: Thông thường, nếu bạn gọi inotify_add_watch( ) trên một tệp mà bạn có watch hiện tại, watch mask sẽ được cập nhật để phản ánh mask mới được cung cấp. Nếu cờ này được đặt trong mask, các sự kiện được cung cấp sẽ được thêm vào mask hiện tại.

IN_ONESHOT

Nếu giá trị này được đặt, kernel sẽ tự động xóa watch sau khi tạo sự kiện đầu tiên đối với đối tượng đã cho. Watch, trên thực tế, là "one shot".

IN_ONLYDIR

Nếu giá trị này được đặt, watch chỉ được thêm vào nếu đối tượng được cung cấp là một thư mục. Nếu path biểu diễn một tệp, không phải thư mục, inotify_add_watch() sẽ không thành công.

Ví dụ, đoạn mã này chỉ thêm chức năng theo dõi trên /etc/init.d nếu init.d là một thư mục và nếu cả /etc và /etc/init.d đều không phải là liên kết tượng trưng:

```
int wd;

/*
 * Theo dõi '/etc/init.d' để xem nó có di chuyển không, nhưng chỉ khi đó là một * thư
 mục và không có phần nào trong đường dẫn của nó là liên kết tượng trưng. */

wd = inotify_add_watch (fd,
                        "/etc/init.d",
                        IN_MOVE_SELF |
                        IN_ONLYDIR |
                        IN_DONT_FOLLOW);
nếu (wd ==
        -1) lỗi ("inotify_add_watch");
```

Xóa một Watch inotify

Như được hiển thị trong ví dụ này, bạn có thể xóa một đồng hồ khỏi một phiên bản inotify bằng lệnh gọi hệ thống inotify_rm_watch( ):

```
#include <inotify.h>

int inotify_rm_watch (int fd, uint32_t wd);
```

Một cuộc gọi thành công đến inotify_rm_watch( ) sẽ xóa watch được biểu diễn bởi mô tả watch wd khỏi thể hiện inotify (được biểu diễn bởi mô tả tệp) fd và trả về 0.

Ví dụ:

```
int ret;

ret = inotify_rm_watch (fd, wd);
nếu
    (ret) lỗi ("inotify_rm_watch");
```

Khi thất bại, lệnh gọi hệ thống trả về -1 và đặt errno thành một trong hai tùy chọn sau:

EBADF

fd không phải là phiên bản inotify hợp lệ.

EINVAL

wd không phải là mô tả giám sát hợp lệ trên phiên bản inotify đã cho.

Khi xóa một watch, kernel tạo ra sự kiện IN_IGNORED . Kernel gửi sự kiện này không chỉ trong quá trình xóa thủ công mà còn khi hủy watch như một tác dụng phụ của một hoạt động khác. Ví dụ, khi một tệp đã theo dõi bị xóa, mọi watch trên tệp đó đều bị xóa. Trong tất cả các trường hợp như vậy, kernel sẽ gửi IN_IGNORED. Hành vi này cho phép các ứng dụng hợp nhất việc xử lý việc xóa watch của chúng tại một nơi duy nhất: trình xử lý sự kiện cho IN_IGNORED. Điều này hữu ích cho những người dùng inotify nâng cao quản lý các cấu trúc dữ liệu phức tạp hỗ trợ cho mỗi watch inotify, chẳng hạn như cơ sở hạ tầng tìm kiếm Beagle của GNOME.

Lấy kích thước của hàng đợi sự kiện Kích thước

của hàng đợi sự kiện đang chờ có thể được lấy thông qua FIONREAD ioctl trên mô tả tệp của phiên bản inotify. Đối số đầu tiên của yêu cầu nhận kích thước của hàng đợi theo byte, dưới dạng số nguyên không dấu:

```
int không dấu queue_len;
int ret;

ret = ioctl (fd, FIONREAD, &queue_len); nếu
(ret < 0)
        perror ("ioctl");
khác
        printf ("%u byte đang chờ trong hàng đợi\n", queue_len);
```

Lưu ý rằng yêu cầu trả về kích thước của hàng đợi theo byte, chứ không phải số sự kiện trong hàng đợi. Một chương trình có thể ước tính số sự kiện từ số byte, bằng cách sử dụng kích thước đã biết của cấu trúc inotify_event (thu được qua sizeof( )) và ước tính kích thước trung bình của trường tên . Tuy nhiên, điều hữu ích hơn là số byte đang chờ xử lý cung cấp cho quy trình kích thước lý tưởng để đọc.

Tiêu đề <sys/ioctl.h> định nghĩa hằng số FIONREAD .

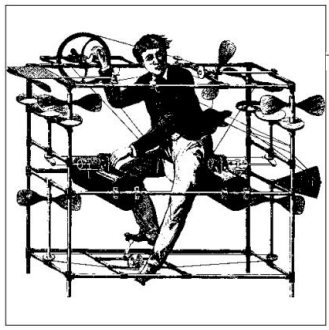
Hủy một Instance inotify

Việc hủy một phiên bản inotify và bất kỳ watch nào liên quan cũng đơn giản như đóng mô tả tệp của phiên bản đó:

```
int ret;

/* 'fd' được lấy thông qua inotify_init( ) */
ret = close (fd);
if (fd == -1)
        perror ("close");
```

Tất nhiên, giống như bất kỳ trình mô tả tệp nào, hạt nhân sẽ tự động đóng trình mô tả tệp và dọn dẹp tài nguyên khi tiến trình thoát.



CHƯƠNG 8

Quản lý bộ nhớ

Bộ nhớ là một trong những tài nguyên cơ bản nhất nhưng cũng thiết yếu nhất có sẵn cho một quy trình. Chương này đề cập đến việc quản lý tài nguyên này: phân bổ, thao tác và giải phóng bộ nhớ cuối cùng.

Động từ allocate— thuật ngữ thông dụng để chỉ việc có được bộ nhớ—có phần gây hiểu lầm, vì nó gợi lên hình ảnh phân bổ một nguồn tài nguyên khan hiếm mà cầu vượt quá cung. Chắc chắn, nhiều người dùng sẽ thích có thêm bộ nhớ. Tuy nhiên, trên các hệ thống hiện đại, vấn đề thực sự không phải là chia sẻ quá ít cho quá nhiều người, mà là sử dụng và theo dõi đúng cách phần thưởng.

Trong chương này, bạn sẽ tìm hiểu về tất cả các phương pháp phân bổ bộ nhớ trong các vùng khác nhau của chương trình, bao gồm ưu điểm và nhược điểm của từng phương pháp. Chúng ta cũng sẽ xem xét một số cách để thiết lập và thao tác nội dung của các vùng bộ nhớ tùy ý và xem cách khóa bộ nhớ để nó vẫn nằm trong RAM và chương trình của bạn không có nguy cơ phải chờ kernel phân trang dữ liệu từ không gian hoán đổi.

Không gian địa chỉ quy trình

Linux, giống như bất kỳ hệ điều hành hiện đại nào, đều ảo hóa tài nguyên bộ nhớ vật lý. Các tiến trình không trực tiếp giải quyết bộ nhớ vật lý. Thay vào đó, hạt nhân liên kết mỗi tiến trình với một không gian địa chỉ ảo duy nhất. Không gian địa chỉ này là tuyến tính, với các địa chỉ bắt đầu từ số không và tăng lên đến một giá trị tối đa nào đó.

Trang và Phân trang

Không gian địa chỉ ảo bao gồm các trang. Kiến trúc hệ thống và loại máy xác định kích thước của một trang, kích thước này là cố định; kích thước thông thường bao gồm 4 KB

(đối với hệ thống 32 bit) và 8 KB (đối với hệ thống 64 bit).* Các trang có thể hợp lệ hoặc không hợp lệ. Một trang hợp lệ được liên kết với một trang trong bộ nhớ vật lý hoặc một số bộ nhớ sao lưu thứ cấp, chẳng hạn như phân vùng hoán đổi hoặc tệp trên đĩa. Một trang không hợp lệ không được liên kết với bất kỳ thứ gì và biểu thị một phần không gian địa chỉ chưa sử dụng, chưa được phân bổ. Truy cập vào một trang như vậy gây ra vi phạm phân đoạn. Không gian địa chỉ không nhất thiết phải liền kề. Mặc dù được định địa chỉ tuyến tính, nhưng nó chứa nhiều địa chỉ không thể định địa chỉ khoảng trống.

Một chương trình không thể sử dụng một trang có trong bộ nhớ thứ cấp thay vì trong bộ nhớ vật lý cho đến khi nó được liên kết với một trang trong bộ nhớ vật lý. Khi một tiến trình cố gắng truy cập một địa chỉ trên một trang như vậy, đơn vị quản lý bộ nhớ (MMU) sẽ tạo ra lỗi trang. Sau đó, hạt nhân can thiệp, phân trang trong suốt trang mong muốn từ bộ nhớ thứ cấp sang bộ nhớ vật lý. Vì có nhiều bộ nhớ ảo hơn đáng kể so với bộ nhớ vật lý (thậm chí, trên nhiều hệ thống, trong một không gian địa chỉ ảo duy nhất!), hạt nhân cũng liên tục phân trang bộ nhớ vật lý sang bộ nhớ thứ cấp để tạo chỗ cho nhiều lần phân trang hơn. Hạt nhân cố gắng phân trang dữ liệu ít có khả năng được sử dụng nhất trong tương lai gần, do đó tối ưu hóa hiệu suất.

Chia sẻ và sao chép khi ghi

Nhiều trang bộ nhớ ảo, ngay cả trong các không gian địa chỉ ảo khác nhau do các quy trình khác nhau sở hữu, có thể ánh xạ tới một trang vật lý duy nhất. Điều này cho phép các không gian địa chỉ ảo khác nhau chia sẻ dữ liệu trong bộ nhớ vật lý. Dữ liệu được chia sẻ có thể chỉ đọc hoặc có thể đọc và ghi.

Khi một tiến trình ghi vào một trang có thể ghi được chia sẻ, một trong hai điều sau có thể xảy ra. Đơn giản nhất là hạt nhân cho phép ghi xảy ra, trong trường hợp đó tất cả các tiến trình chia sẻ trang có thể thấy kết quả của hoạt động ghi. Thông thường, cho phép nhiều tiến trình đọc từ hoặc ghi vào một trang chia sẻ đòi hỏi một số mức độ phối hợp và đồng bộ hóa.

Tuy nhiên, theo một cách khác, MMU có thể chặn hoạt động ghi và đưa ra ngoại lệ; phản hồi lại, hạt nhân sẽ tạo một bản sao mới của trang cho quy trình ghi một cách minh bạch và cho phép tiếp tục ghi trên trang mới. Chúng tôi gọi cách tiếp cận này là sao chép khi ghi (COW).† Trên thực tế, các quy trình được phép truy cập đọc vào dữ liệu được chia sẻ, giúp tiết kiệm không gian. Khi một quy trình muốn ghi vào một trang được chia sẻ, nó sẽ nhận được một bản sao duy nhất của trang đó ngay lập tức, do đó cho phép hạt nhân hoạt động như thể quy trình đó luôn có bản sao riêng của nó. Vì sao chép khi ghi diễn ra trên từng trang, với cách tiếp cận này, một tệp lớn có thể được chia sẻ hiệu quả giữa nhiều quy trình và các quy trình riêng lẻ sẽ chỉ nhận được các trang vật lý duy nhất cho những trang mà chính chúng ghi vào.

* Một số hệ thống hỗ trợ một loạt các kích thước trang. Vì lý do này, kích thước trang không phải là một phần của ABI. Các ứng dụng phải lập trình để lấy kích thước trang khi chạy. Chúng tôi đã đề cập đến việc thực hiện điều này trong Chương 4 và sẽ xem xét chủ đề trong

chương này. † Nhớ lại từ Chương 5 rằng fork( ) sử dụng copy-on-write để sao chép và chia sẻ không gian địa chỉ của phần tử cha với đứa trẻ.

Vùng bộ nhớ Nhân sắp xếp

các trang thành các khối chia sẻ một số thuộc tính nhất định, chẳng hạn như quyền truy cập. Các khối này được gọi là vùng bộ nhớ, phân đoạn hoặc ánh xạ. Có thể tìm thấy một số loại vùng bộ nhớ nhất định trong mọi quy trình:

- Đoạn văn bản chứa mã chương trình của quy trình, chuỗi ký tự, biến hằng số và dữ liệu chỉ đọc khác. Trong Linux, đoạn này được đánh dấu là chỉ đọc và được ánh xạ trực tiếp từ tệp đối tượng (chương trình thực thi hoặc thư viện). • Ngăn xếp chứa ngăn xếp thực thi của quy

  trình, ngăn xếp này tăng và giảm động khi độ sâu của ngăn xếp tăng và giảm. Ngăn xếp thực thi chứa các biến cục bộ và dữ liệu trả về của hàm.

- Phân đoạn dữ liệu hoặc heap chứa bộ nhớ động của một tiến trình. Phân đoạn này có thể ghi và có thể tăng hoặc giảm kích thước. Đây là bộ nhớ được trả về bởi malloc() (được thảo luận trong phần tiếp theo).
- Phân đoạn bss* chứa các biến toàn cục chưa được khởi tạo. Các biến này chứa các giá trị đặc biệt (về cơ bản là tất cả đều là số không), theo tiêu chuẩn C.

  Linux tối ưu hóa các biến này theo hai cách. Đầu tiên, vì phân đoạn bss dành riêng cho dữ liệu chưa được khởi tạo, nên trình liên kết (ld) không thực sự lưu trữ các giá trị đặc biệt trong tệp đối tượng. Điều này làm giảm kích thước của tệp nhị phân. Thứ hai, khi phân đoạn này được tải vào bộ nhớ, hạt nhân chỉ cần ánh xạ nó theo cơ sở sao chép khi ghi vào một trang số không, hiệu quả là đặt các biến thành giá trị mặc định của chúng. • Hầu

  hết các không gian địa chỉ đều chứa một số tệp được ánh xạ, chẳng hạn như tệp thực thi chương trình, thư viện C và các thư viện được liên kết khác và tệp dữ liệu. Hãy xem /proc/self/maps hoặc đầu ra từ chương trình pmap để biết ví dụ tuyệt vời về các tệp được ánh xạ trong một quy trình.

Chương này trình bày về các giao diện mà Linux cung cấp để lấy và trả lại bộ nhớ, tạo và hủy các ánh xạ mới cùng mọi thứ ở giữa.

Phân bổ bộ nhớ động

Bộ nhớ cũng có dạng biến tự động và biến tĩnh, nhưng nền tảng của bất kỳ hệ thống quản lý bộ nhớ nào là việc phân bổ, sử dụng và cuối cùng là trả về bộ nhớ động. Bộ nhớ động được phân bổ tại thời gian chạy, không phải thời gian biên dịch, với kích thước có thể không xác định cho đến thời điểm phân bổ. Là một nhà phát triển, bạn cần bộ nhớ động khi lượng bộ nhớ bạn cần hoặc thời gian bạn có thể cần thay đổi và không biết trước khi chương trình chạy. Ví dụ, bạn có thể muốn lưu trữ trong bộ nhớ nội dung của tệp hoặc đầu vào được đọc từ bàn phím. Vì kích thước của tệp không xác định và người dùng có thể nhập bất kỳ số lượng

* Tên này có tính lịch sử; nó bắt nguồn từ khối bắt đầu bằng ký hiệu.

mỗi lần nhấn phím, kích thước của bộ đệm sẽ thay đổi và bạn có thể cần phải tăng kích thước bộ đệm một cách linh hoạt khi bạn đọc ngày càng nhiều dữ liệu.

Không có biến C nào được hỗ trợ bởi bộ nhớ động. Ví dụ, C không cung cấp cơ chế để lấy struct pirate_ship tồn tại trong bộ nhớ động.
Thay vào đó, C cung cấp một cơ chế để phân bổ bộ nhớ động đủ để chứa một cấu trúc pirate_ship . Sau đó, lập trình viên tương tác với bộ nhớ thông qua một con trỏ— trong trường hợp này là một struct pirate_ship *.

Giao diện C cổ điển để lấy bộ nhớ động là malloc( ):

```
#include <stdlib.h>

void * malloc(kích thước_t);
```

Một lệnh gọi thành công đến malloc() sẽ phân bổ size byte bộ nhớ và trả về một con trỏ đến phần đầu của vùng mới được phân bổ. Nội dung của bộ nhớ không được xác định; đừng mong đợi bộ nhớ được đặt về 0. Khi không thành công, malloc() trả về NULL và errno được đặt thành ENOMEM.

Việc sử dụng malloc() có thể khá đơn giản, như trong ví dụ này được sử dụng để phân bổ một số byte cố định:

```
ký tự *p;

/* cho tôi 2 KB! */
p = malloc (2048);
nếu (!
     p) perror ("malloc");
```

hoặc ví dụ này được sử dụng để phân bổ một cấu trúc:

```
cấu trúc treasure_map *map;

/*
 * phân bổ đủ bộ nhớ để chứa cấu trúc treasure_map * và trỏ 'map' vào cấu trúc
 đó */

bản đồ = malloc (sizeof (struct treasure_map));
nếu (!map)
         perror ("malloc");
```

C tự động thăng cấp con trỏ void thành bất kỳ kiểu nào khi gán. Do đó, các ví dụ này không cần phải ép kiểu giá trị trả về của malloc() thành kiểu của lvalue được sử dụng trong các phép gán. Tuy nhiên, ngôn ngữ lập trình C++ không thực hiện việc tự động thăng cấp con trỏ void . Do đó, người dùng C++ cần ép kiểu giá trị trả về của malloc( ) như sau:

```
char *tên;

/* phân bổ 512 byte */ name
= (char *) malloc (512); nếu (!
name)
      perror ("malloc");
```

Một số lập trình viên C thích ép kiểu kết quả của bất kỳ hàm nào trả về con trỏ void, bao gồm malloc() . Tôi phản đối cách làm này vì nó sẽ ẩn lỗi nếu giá trị trả về của hàm thay đổi thành giá trị khác ngoài con trỏ void . Hơn nữa, ép kiểu như vậy cũng ẩn lỗi nếu hàm không được khai báo đúng cách.* Trong khi cách trước không phải là rủi ro với malloc( ), thì cách sau chắc chắn là rủi ro.

Vì malloc() có thể trả về NULL, nên điều cực kỳ quan trọng là các nhà phát triển phải luôn kiểm tra và xử lý các điều kiện lỗi. Nhiều chương trình định nghĩa và sử dụng trình bao bọc malloc() để in thông báo lỗi và kết thúc chương trình nếu malloc() trả về NULL. Theo quy ước, các nhà phát triển gọi trình bao bọc chung này là xmalloc( ):

```
/* giống như malloc( ), nhưng kết thúc khi có lỗi */
void * xmalloc (size_t size) {

        không có *p;

        p = malloc (kích
        thước);
                nếu (!p) { perror
                  ("xmalloc"); thoát (EXIT_FAILURE);
        }

        trả về p;
}
```

Phân bổ mảng Phân bổ

bộ nhớ động cũng có thể khá phức tạp khi kích thước được chỉ định tự nó là động. Một ví dụ như vậy là phân bổ động của mảng, trong đó kích thước của một phần tử mảng có thể cố định, nhưng số lượng phần tử cần phân bổ là động.
Để đơn giản hóa tình huống này, C cung cấp hàm calloc() :

```
#include <stdlib.h>

void * calloc (kích thước_t số, kích thước_t kích thước);
```

Một lệnh gọi calloc( ) thành công sẽ trả về một con trỏ đến một khối bộ nhớ phù hợp để chứa một mảng gồm nr phần tử, mỗi phần tử có kích thước byte. Do đó, lượng bộ nhớ được yêu cầu trong hai lệnh gọi này là giống hệt nhau (cả hai lệnh đều có thể trả về nhiều bộ nhớ hơn mức yêu cầu, nhưng không bao giờ ít hơn):

```
số nguyên *x, *y;

x = malloc (50 * sizeof (int)); nếu
(!x)
     { perror ("malloc"); trả
          về -1;
```

* Các hàm chưa khai báo mặc định trả về một int. Chuyển đổi từ số nguyên sang con trỏ không tự động và tạo ra cảnh báo. Chuyển đổi kiểu sẽ ngăn chặn cảnh báo kết quả.

```
}

y = calloc (50, sizeof (int)); nếu
(!y)
    { perror ("calloc"); trả
        về -1;
}
```

Tuy nhiên, hành vi không giống hệt nhau. Không giống như malloc( ), không đưa ra bất kỳ đảm bảo nào về nội dung của bộ nhớ được phân bổ, calloc() sẽ đưa tất cả các byte về 0 trong khối bộ nhớ được trả về. Do đó, mỗi phần tử trong 50 phần tử của mảng số nguyên y giữ giá trị 0, trong khi nội dung của các phần tử trong x không được xác định. Trừ khi chương trình sẽ ngay lập tức đặt tất cả 50 giá trị, các lập trình viên nên sử dụng calloc() để đảm bảo rằng các phần tử mảng không chứa đầy những ký tự vô nghĩa. Lưu ý rằng số không nhị phân có thể không giống với số không dấu phẩy động!

Người dùng thường muốn "xóa bỏ" bộ nhớ động, ngay cả khi không xử lý mảng.
Sau này trong chương này, chúng ta sẽ xem xét memset( ), cung cấp giao diện để thiết lập mọi byte trong một khối bộ nhớ thành một giá trị nhất định. Tuy nhiên, để calloc() thực hiện việc đưa về 0 sẽ nhanh hơn vì kernel có thể cung cấp bộ nhớ đã được đưa về 0.

Khi lỗi, giống như malloc( ), calloc() trả về NULL và đặt errno thành ENOMEM.

Tại sao các cơ quan tiêu chuẩn không bao giờ định nghĩa một hàm “phân bổ và zero” tách biệt với calloc() vẫn còn là một bí ẩn. Tuy nhiên, các nhà phát triển có thể dễ dàng định nghĩa giao diện của riêng họ:

```
/* hoạt động giống hệt malloc( ), nhưng bộ nhớ được đưa về 0 */
void * malloc0 (size_t size) {

          trả về calloc (1, kích thước);
}
```

Thuận tiện hơn, chúng ta có thể kết hợp malloc0() này với xmalloc( ) trước đó của chúng ta:

```
/* giống như malloc( ), nhưng xóa bộ nhớ và kết thúc khi có lỗi */ void *
xmalloc0 (size_t size) {

          không có *p;

          p = calloc (1, kích
          thước);
                  nếu (!p) { perror
                    ("xmalloc0"); thoát (EXIT_FAILURE);
          }

          trả về p;
}
```

Thay đổi kích thước phân bổ

Ngôn ngữ C cung cấp giao diện để thay đổi kích thước (làm cho lớn hơn hoặc nhỏ hơn) các phân bổ hiện có:

```
#include <stdlib.h>

void * realloc (void *ptr, size_t kích thước);
```

Một lệnh gọi thành công đến realloc( ) sẽ thay đổi kích thước vùng bộ nhớ được trỏ đến bởi ptr thành một kích thước mới là size bytes. Nó trả về một con trỏ đến bộ nhớ có kích thước mới, có thể giống hoặc không giống với ptr—khi mở rộng một vùng bộ nhớ, nếu realloc( ) không thể mở rộng khối bộ nhớ hiện có bằng cách mở rộng khối tại chỗ, hàm có thể phân bổ một vùng bộ nhớ mới có kích thước byte theo chiều dài, sao chép vùng cũ vào vùng mới và giải phóng vùng cũ. Trong bất kỳ hoạt động nào, nội dung của vùng bộ nhớ được bảo toàn ở mức tối thiểu của kích thước cũ và kích thước mới.
Do khả năng sao chép, thao tác realloc() để mở rộng vùng bộ nhớ có thể là một hành động khá tốn kém.

Nếu size là 0, hiệu ứng sẽ giống như khi gọi free() trên ptr.

Nếu ptr là NULL, kết quả của hoạt động sẽ giống như một malloc( ) mới. Nếu ptr không phải NULL, nó phải được trả về thông qua một lệnh gọi trước đó tới malloc( ), calloc( ) hoặc realloc( ).

Khi thất bại, realloc() trả về NULL và đặt errno thành ENOMEM. Trạng thái của bộ nhớ được trỏ tới bởi ptr không thay đổi.

Hãy xem xét một ví dụ về việc thu hẹp vùng bộ nhớ. Đầu tiên, chúng ta sẽ sử dụng calloc() để phân bổ đủ bộ nhớ để chứa một mảng hai phần tử của các cấu trúc bản đồ :

```
cấu trúc bản đồ *p;

/* cấp phát bộ nhớ cho hai cấu trúc bản đồ */ p =
calloc (2, sizeof (struct map)); if (!p)
{ perror
        ("calloc"); return -1;

}

/* sử dụng p[0] và p[1]... */
```

Bây giờ, giả sử chúng ta đã tìm thấy một trong những kho báu và không còn cần bản đồ thứ hai nữa, do đó chúng ta quyết định thay đổi kích thước bộ nhớ và trả lại một nửa khu vực cho hệ thống (nói chung đây không phải là một hoạt động đáng giá, nhưng có thể đáng giá nếu cấu trúc bản đồ rất lớn và chúng ta sẽ giữ bản đồ còn lại trong một thời gian dài):

```
cấu trúc bản đồ *r;

/* bây giờ chúng ta chỉ cần bộ nhớ cho một bản đồ */ r
= realloc (p, sizeof (struct map)); if (!r) {
```

```
            /* lưu ý rằng 'p' vẫn hợp lệ! */ perror
            ("realloc"); return -1;

    }

    /* sử dụng 'r'... */

    miễn phí (r);
```

Trong ví dụ này, p[0] được bảo toàn sau khi gọi realloc( ) . Bất kỳ dữ liệu nào đã có trước đó vẫn còn đó. Nếu lệnh gọi trả về lỗi, hãy lưu ý rằng p không bị ảnh hưởng và do đó vẫn hợp lệ. Chúng ta có thể tiếp tục sử dụng nó và cuối cùng sẽ cần giải phóng nó. Ngược lại, nếu lệnh gọi thành công, chúng ta bỏ qua p và thay vào đó sử dụng r (có khả năng giống như p, vì vùng gần như chắc chắn đã được thay đổi kích thước tại chỗ). Bây giờ chúng ta có trách nhiệm giải phóng r khi hoàn tất.

Giải phóng bộ nhớ động Không

giống như các phân bổ tự động, được tự động thu hoạch khi ngăn xếp được giải phóng, các phân bổ động là các phần cố định của không gian địa chỉ của quy trình cho đến khi chúng được giải phóng thủ công. Do đó, lập trình viên chịu trách nhiệm trả lại bộ nhớ được phân bổ động cho hệ thống. (Tất nhiên, cả phân bổ tĩnh và động đều biến mất khi toàn bộ quy trình thoát.)

Bộ nhớ được phân bổ bằng malloc( ), calloc( ) hoặc realloc() phải được trả về hệ thống khi không còn sử dụng nữa thông qua free( ):

```
#include <stdlib.h>

void giải phóng (void *ptr);
```

Một lệnh gọi đến free() giải phóng bộ nhớ tại ptr. Tham số ptr phải được trả về trước đó bởi malloc( ), calloc( ) hoặc realloc( ). Nghĩa là, bạn không thể sử dụng free() để giải phóng một phần khối bộ nhớ—ví dụ, một nửa khối bộ nhớ—bằng cách truyền một con trỏ vào giữa khối được phân bổ. ptr có thể

là NULL, trong trường hợp đó free() sẽ trả về một cách âm thầm. Do đó, việc thường thấy là kiểm tra ptr để tìm NULL trước khi gọi free() là thừa.

Hãy cùng xem một ví dụ:

```
void print_chars (int n, char c) {

        số nguyên i;

        đối với (i = 0; i < n; i++)
                { char
                *s; int j;

                /*
                 * Phân bổ và đưa mảng i+2 phần tử * về 0. Lưu ý rằng
                 'sizeof (char)'
```

```
                 * luôn luôn là 1.
                 */
                s = calloc (i + 2, 1);
                nếu (!s)
                        { lỗi ("calloc");
                        ngắt;
                }

                đối với (j = 0; j < i + 1; j++)
                        s[j] = c;

                printf ("%s\n", s);

                /* Được rồi, xong rồi. Trả lại bộ nhớ. */ giải
                phóng (s);
        }
}
```

Ví dụ này phân bổ n mảng ký tự chứa số lượng phần tử lớn hơn liên tiếp, từ hai phần tử (2 byte) đến n + 1 phần tử (n + 1 byte).
Sau đó, đối với mỗi mảng, vòng lặp ghi ký tự c vào từng byte ngoại trừ byte cuối cùng (giữ lại số 0 đã có trong byte cuối cùng), in mảng dưới dạng chuỗi, rồi giải phóng bộ nhớ được cấp phát động.

Gọi print_chars() với n bằng 5 và c được đặt thành X, chúng ta nhận được kết quả sau:

```
X
XX
XXX
XXXX
XXXXX
```

Tất nhiên, có nhiều cách hiệu quả hơn nhiều để thực hiện chức năng này.
Tuy nhiên, vấn đề là chúng ta có thể phân bổ và giải phóng bộ nhớ động ngay cả khi kích thước và số lượng phân bổ đó chỉ được biết khi chạy.

Các hệ thống Unix như SunOS và SCO cung cấp một biến thể của free( ) có tên là cfree( ), tùy thuộc vào hệ thống, hoạt động giống như free( ), hoặc nhận ba tham số, phản ánh calloc( ). Trong Linux, free( ) có thể xử lý bộ nhớ thu được từ bất kỳ cơ chế phân bổ nào mà chúng ta đã thảo luận cho đến nay. Không bao giờ nên sử dụng cfree( ) , ngoại trừ khả năng tương thích ngược. Phiên bản Linux giống như free( ).

Lưu ý hậu quả sẽ như thế nào nếu ví dụ này không gọi free( ). Chương trình sẽ không bao giờ trả lại bộ nhớ cho hệ thống và tệ hơn nữa là nó sẽ mất tham chiếu duy nhất đến bộ nhớ—con trỏ s—do đó không bao giờ có thể truy cập vào bộ nhớ. Chúng tôi gọi loại lỗi lập trình này là rò rỉ bộ nhớ.
Rò rỉ bộ nhớ và các lỗi bộ nhớ động tương tự là một trong những sự cố phổ biến nhất và không may là những sự cố có hại nhất trong lập trình C. Bởi vì C

Ngôn ngữ C trao toàn bộ trách nhiệm quản lý bộ nhớ cho lập trình viên, các lập trình viên C phải luôn theo dõi chặt chẽ mọi phân bổ bộ nhớ.

Một lỗi lập trình C phổ biến khác là use-after-free. Lỗi này xảy ra khi một khối bộ nhớ được giải phóng, sau đó được truy cập. Khi free( ) được gọi trên một khối bộ nhớ, một chương trình không bao giờ được truy cập lại nội dung của nó. Các lập trình viên phải đặc biệt cẩn thận để theo dõi các con trỏ lơ lửng hoặc các con trỏ không phải NULL nhưng vẫn trỏ đến các khối bộ nhớ không hợp lệ. Hai công cụ phổ biến để hỗ trợ bạn trong nhiệm vụ này là Electric Fence và valgrind.*

Căn chỉnh Căn

chỉnh dữ liệu đề cập đến mối quan hệ giữa địa chỉ của nó và các khối bộ nhớ được đo bằng phần cứng. Một biến nằm ở địa chỉ bộ nhớ là bội số của kích thước của nó được gọi là căn chỉnh tự nhiên. Ví dụ, một biến 32 bit được căn chỉnh tự nhiên nếu nó nằm trong bộ nhớ tại một địa chỉ là bội số của 4—tức là nếu hai bit thấp nhất của địa chỉ là 0. Do đó, một kiểu có kích thước 2n byte phải có một địa chỉ với n bit ít quan trọng nhất được đặt thành 0.

Các quy tắc liên quan đến căn chỉnh bắt nguồn từ phần cứng. Một số kiến trúc máy có các yêu cầu rất nghiêm ngặt về căn chỉnh dữ liệu. Trên một số hệ thống, một lượng dữ liệu không căn chỉnh sẽ dẫn đến bẫy bộ xử lý. Trên các hệ thống khác, việc truy cập dữ liệu không căn chỉnh là an toàn, nhưng sẽ làm giảm hiệu suất. Khi viết mã di động, phải tránh các vấn đề căn chỉnh và tất cả các loại phải được căn chỉnh tự nhiên.

Phân bổ bộ nhớ được căn

chỉnh Đối với hầu hết các phần, trình biên dịch và thư viện C xử lý minh bạch các mối quan tâm về căn chỉnh. POSIX quy định rằng bộ nhớ được trả về thông qua malloc( ), calloc( ) và realloc() phải được căn chỉnh đúng cách để sử dụng với bất kỳ kiểu C chuẩn nào. Trên Linux, các hàm này luôn trả về bộ nhớ được căn chỉnh theo ranh giới 8 byte trên các hệ thống 32 bit và ranh giới 16 byte trên các hệ thống 64 bit.

Thỉnh thoảng, các lập trình viên yêu cầu bộ nhớ động được căn chỉnh theo một ranh giới lớn hơn, chẳng hạn như một trang. Mặc dù động cơ khác nhau, nhưng phổ biến nhất là nhu cầu căn chỉnh đúng các bộ đệm được sử dụng trong I/O khối trực tiếp hoặc giao tiếp phần mềm-phần cứng khác. Vì mục đích này, POSIX 1003.1d cung cấp một hàm có tên là posix_memalign( ):

```
/* một trong hai -- cái nào cũng được */ #define
_XOPEN_SOURCE 600 #define
_GNU_SOURCE

#include <stdlib.h>

int posix_memalign (void **memptr,
                    size_t căn chỉnh,
                    size_t kích thước);
```

* Xem http://perens.com/FreeSoftware/ElectricFence/ Và http://valgrind.org, tương ứng.

Một lệnh gọi thành công đến posix_memalign( ) phân bổ kích thước byte của bộ nhớ động, đảm bảo nó được căn chỉnh theo một địa chỉ bộ nhớ là bội số của căn chỉnh. Tham số căn chỉnh phải là lũy thừa của 2 và là bội số của kích thước của một con trỏ void . Địa chỉ của bộ nhớ được phân bổ được đặt trong memptr và lệnh gọi trả về 0.

Khi lỗi xảy ra, không có bộ nhớ nào được phân bổ, memptr không được xác định và lệnh gọi trả về một trong các mã lỗi sau:

EINVAL

Căn chỉnh tham số không phải là lũy thừa của 2 hoặc không phải là bội số của kích thước của một con trỏ void .

ENOMEM

Không có đủ bộ nhớ để đáp ứng nhu cầu phân bổ được yêu cầu.

Lưu ý rằng errno không được thiết lập—hàm này sẽ trực tiếp trả về các lỗi này.

Bộ nhớ thu được thông qua posix_memalign() được giải phóng thông qua free( ). Cách sử dụng rất đơn giản:

```
char *buf;
int ret;

/* phân bổ 1 KB dọc theo ranh giới 256 byte */ ret =
posix_memalign (&buf, 256, 1024); if (ret)
{ fprintf
          (stderr, "posix_memalign: %s\n", strerror (ret));
                    return -1;

}

/* sử dụng 'buf'... */

miễn phí (buf);
```

Giao diện cũ hơn. Trước khi POSIX định nghĩa posix_memalign( ) , BSD và SunOS cung cấp các giao diện sau đây:

```
#include <malloc.h>

void * valloc (kích thước size_t);
size_t); void *memalign (kích thước size_t ranh giới, kích thước
```

Hàm valloc() hoạt động giống hệt malloc( ), ngoại trừ bộ nhớ được phân bổ được căn chỉnh theo ranh giới trang. Nhớ lại từ Chương 4 rằng kích thước trang của hệ thống có thể dễ dàng lấy được thông qua getpagesize( ).

Hàm memalign() cũng tương tự, nhưng nó căn chỉnh việc phân bổ theo ranh giới của các byte ranh giới , phải là lũy thừa của 2. Trong ví dụ này, cả hai phép phân bổ này đều trả về một khối bộ nhớ đủ để chứa một cấu trúc tàu , được căn chỉnh theo ranh giới trang:

```
cấu trúc tàu *cướp biển, *hms;

cướp biển = valloc(sizeof(struct ship));
```

```
    nếu (!pirate)
              { perror ("valloc");
              trả về -1;
    }

    hms = memalign(getpagesize(), sizeof(struct ship)); nếu (!hms) { lỗi
    ("memalign");
              miễn phí (cướp biển);
              trả về -1;

    }

    /* sử dụng 'pirate' và 'hms'... */

    miễn phí (hms);
    miễn phí (cướp biển);
```

Trên Linux, bộ nhớ thu được thông qua cả hai hàm này đều có thể giải phóng thông qua free( ). Điều này có thể không đúng trên các hệ thống Unix khác, một số hệ thống không cung cấp cơ chế giải phóng bộ nhớ được phân bổ an toàn bằng các hàm này. Các chương trình liên quan đến tính di động có thể không có lựa chọn nào khác ngoài việc không giải phóng bộ nhớ được phân bổ thông qua các giao diện này!

Các lập trình viên Linux chỉ nên sử dụng hai hàm này cho mục đích di động với các hệ thống cũ hơn; posix_memalign() là tốt hơn. Cả ba giao diện này chỉ cần thiết nếu cần căn chỉnh lớn hơn căn chỉnh do malloc() cung cấp .

Các mối quan tâm căn chỉnh

khác Các mối quan tâm căn chỉnh mở rộng ra ngoài sự căn chỉnh tự nhiên của các kiểu chuẩn và phân bổ bộ nhớ động. Ví dụ, các kiểu không chuẩn và phức tạp có các yêu cầu phức tạp hơn các kiểu chuẩn. Hơn nữa, các mối quan tâm căn chỉnh quan trọng gấp đôi khi gán giá trị giữa các con trỏ có các kiểu khác nhau và sử dụng ép kiểu.

Các kiểu dữ liệu không chuẩn. Các kiểu dữ liệu không chuẩn và phức tạp sở hữu các yêu cầu căn chỉnh vượt ra ngoài yêu cầu căn chỉnh tự nhiên đơn giản. Bốn quy tắc hữu ích sau đây: • Yêu cầu căn chỉnh

của một cấu trúc là yêu cầu của kiểu thành phần lớn nhất của nó.
Ví dụ, nếu kiểu lớn nhất của một cấu trúc là số nguyên 32 bit được căn chỉnh theo ranh giới 4 byte, thì cấu trúc đó cũng phải được căn chỉnh theo ít nhất một ranh giới 4 byte.

• Cấu trúc cũng giới thiệu nhu cầu về phần đệm, được sử dụng để đảm bảo rằng mỗi kiểu thành phần được căn chỉnh đúng theo yêu cầu riêng của kiểu đó. Do đó, nếu một char (có thể căn chỉnh một byte) thấy chính nó theo sau một int (có thể căn chỉnh bốn byte), trình biên dịch sẽ chèn ba byte đệm giữa hai kiểu để đảm bảo rằng int nằm trên ranh giới bốn byte.

Đôi khi, các lập trình viên sắp xếp các thành viên của một cấu trúc—ví dụ, theo kích thước giảm dần—để giảm thiểu không gian “bị lãng phí” do đệm. Tùy chọn GCC -Wpadded có thể hỗ trợ trong nỗ lực này, vì nó tạo ra cảnh báo bất cứ khi nào trình biên dịch chèn đệm ngầm.

- Yêu cầu căn chỉnh của một liên hợp là yêu cầu của loại liên hợp lớn nhất. •
- Yêu cầu căn chỉnh của một mảng là yêu cầu của loại cơ sở. Do đó, các mảng không mang theo yêu cầu nào ngoài một thể hiện duy nhất của loại của chúng. Hành vi này dẫn đến sự căn chỉnh tự nhiên của tất cả các thành viên của một mảng.

Chơi với con trỏ. Vì trình biên dịch xử lý hầu hết các yêu cầu căn chỉnh một cách minh bạch nên cần một chút nỗ lực để phát hiện các vấn đề tiềm ẩn. Tuy nhiên, không phải là chưa từng gặp phải các vấn đề về căn chỉnh khi xử lý con trỏ và ép kiểu.

Truy cập dữ liệu thông qua con trỏ được đúc lại từ khối dữ liệu ít được căn chỉnh hơn sang khối dữ liệu được căn chỉnh lớn hơn có thể dẫn đến việc bộ xử lý tải dữ liệu không được căn chỉnh đúng cho loại lớn hơn. Ví dụ, trong đoạn mã sau, việc gán c cho badnews cố gắng đọc c dưới dạng số nguyên dài không dấu:

```
char greeting[] = "Ahoy Matey";
char *c = greeting[1];
unsigned long badnews = *(unsigned long *) c;
```

Một unsigned long có khả năng được căn chỉnh theo ranh giới bốn hoặc tám byte; c gần như chắc chắn nằm cách ranh giới đó một byte. Do đó, tải của c, khi ép kiểu, gây ra vi phạm căn chỉnh. Tùy thuộc vào kiến trúc, điều này có thể gây ra các kết quả từ nhỏ như ảnh hưởng đến hiệu suất đến lớn như sự cố chương trình. Trên các kiến trúc máy có thể phát hiện nhưng không xử lý đúng cách các vi phạm căn chỉnh, hạt nhân sẽ gửi tín hiệu SIGBUS cho quy trình vi phạm , tín hiệu này sẽ chấm dứt quy trình. Chúng ta sẽ thảo luận về các tín hiệu trong Chương 9.

Những ví dụ như thế này phổ biến hơn người ta nghĩ. Những ví dụ thực tế sẽ không ngớ ngẩn đến vậy về mặt hình thức, nhưng chúng cũng có thể ít rõ ràng hơn.

Quản lý phân đoạn dữ liệu

Các hệ thống Unix trong lịch sử đã cung cấp các giao diện để quản lý trực tiếp phân đoạn dữ liệu. Tuy nhiên, hầu hết các chương trình không sử dụng trực tiếp các giao diện này vì malloc() và các lược đồ phân bổ khác dễ sử dụng hơn và mạnh hơn. Tôi sẽ đề cập đến các giao diện này ở đây để thỏa mãn những người tò mò và cho những độc giả hiếm hoi muốn triển khai cơ chế phân bổ dựa trên heap của riêng mình:

```
#include <unistd.h>

int brk (void *end);
void * sbrk (intptr_t tăng);
```

Các hàm này có tên bắt nguồn từ các hệ thống Unix cũ, trong đó heap và stack nằm trong cùng một phân đoạn. Các phân bổ bộ nhớ động trong heap tăng lên từ dưới cùng của phân đoạn; stack tăng xuống về phía heap từ trên cùng của phân đoạn. Đường phân định ngăn cách hai phân đoạn này được gọi là break hoặc break point. Trên các hệ thống hiện đại, trong đó phân đoạn dữ liệu nằm trong ánh xạ bộ nhớ riêng của nó, chúng ta tiếp tục gắn nhãn địa chỉ cuối của ánh xạ là break point.

Một lệnh gọi đến brk( ) đặt điểm ngắt (điểm kết thúc của phân đoạn dữ liệu) thành địa chỉ được chỉ định bởi end. Khi thành công, nó trả về 0. Khi thất bại, nó trả về -1 và đặt errno thành ENOMEM.

Một lệnh gọi đến sbrk( ) sẽ tăng phần cuối của phân đoạn dữ liệu theo từng phần, có thể là một delta dương hoặc âm. sbrk( ) trả về điểm ngắt đã sửa đổi. Do đó, một phần tăng 0 sẽ in ra điểm ngắt hiện tại:

```
printf ("Điểm dừng hiện tại là %p\n", sbrk (0));
```

Cả POSIX và chuẩn C đều không định nghĩa bất kỳ hàm nào trong số này.
Tuy nhiên, hầu hết các hệ thống Unix đều hỗ trợ một hoặc cả hai. Các chương trình di động phải tuân theo các giao diện dựa trên tiêu chuẩn.

Ánh xạ bộ nhớ ẩn danh Phân bổ bộ nhớ trong glibc sử dụng

phân đoạn dữ liệu và ánh xạ bộ nhớ. Phương pháp cổ điển để triển khai malloc() là chia phân đoạn dữ liệu thành một loạt các phân vùng lũy thừa của 2 và đáp ứng các phân bổ bằng cách trả về phân vùng phù hợp nhất với kích thước được yêu cầu. Giải phóng bộ nhớ đơn giản bằng cách đánh dấu phân vùng là "rảnh". Nếu các phân vùng liền kề rảnh, chúng có thể được hợp nhất thành một phân vùng lớn hơn. Nếu đỉnh của heap hoàn toàn rảnh, hệ thống có thể sử dụng brk() để hạ thấp điểm ngắt, thu nhỏ heap và trả lại bộ nhớ cho kernel.

Thuật toán này được gọi là sơ đồ phân bổ bộ nhớ buddy. Nó có ưu điểm là tốc độ và tính đơn giản, nhưng nhược điểm là đưa ra hai loại phân mảnh. Phân mảnh nội bộ xảy ra khi sử dụng nhiều bộ nhớ hơn yêu cầu để đáp ứng một phân bổ. Điều này dẫn đến việc sử dụng không hiệu quả bộ nhớ khả dụng. Phân mảnh ngoài xảy ra khi có đủ bộ nhớ trống để đáp ứng một yêu cầu, nhưng nó bị chia thành hai hoặc nhiều khối không liền kề. Điều này có thể dẫn đến việc sử dụng bộ nhớ không hiệu quả (vì có thể sử dụng một khối lớn hơn, ít phù hợp hơn) hoặc phân bổ bộ nhớ không thành công (nếu không có khối thay thế nào tồn tại).

Hơn nữa, lược đồ này cho phép một phân bổ bộ nhớ “ghim” một phân bổ khác, ngăn glibc trả lại bộ nhớ đã giải phóng cho kernel. Hãy tưởng tượng rằng hai khối bộ nhớ, khối A và khối B, được phân bổ. Khối A nằm ngay trên điểm ngắt, và khối B nằm ngay bên dưới A. Ngay cả khi chương trình giải phóng B, glibc không thể điều chỉnh điểm ngắt cho đến khi A cũng được giải phóng. Theo cách này, một phân bổ sống lâu có thể ghim tất cả các phân bổ khác trong bộ nhớ.

Điều này không phải lúc nào cũng là mối quan tâm vì glibc không thường xuyên trả lại bộ nhớ cho hệ thống.* Nói chung, heap không bị thu hẹp sau mỗi lần giải phóng. Thay vào đó, glibc giữ lại bộ nhớ được giải phóng xung quanh cho lần phân bổ tiếp theo. Chỉ khi kích thước của

* glibc cũng sử dụng thuật toán phân bổ bộ nhớ tiên tiến hơn đáng kể so với lược đồ phân bổ buddy đơn giản này, được gọi là thuật toán arena.

heap lớn hơn đáng kể so với lượng bộ nhớ được phân bổ, glibc có thu nhỏ phân đoạn dữ liệu không. Tuy nhiên, một phân bổ lớn có thể ngăn chặn sự thu nhỏ này.

Do đó, đối với các phân bổ lớn, glibc không sử dụng heap. Thay vào đó, glibc tạo ra một ánh xạ bộ nhớ ẩn danh để đáp ứng yêu cầu phân bổ. Ánh xạ bộ nhớ ẩn danh tương tự như ánh xạ dựa trên tệp được thảo luận trong Chương 4, ngoại trừ việc chúng không được hỗ trợ bởi bất kỳ tệp nào—do đó có biệt danh "ẩn danh".
Thay vào đó, ánh xạ bộ nhớ ẩn danh chỉ đơn giản là một khối bộ nhớ lớn, không có giá trị nào, sẵn sàng để bạn sử dụng. Hãy nghĩ về nó như một đống hoàn toàn mới, chỉ dành cho một lần phân bổ duy nhất. Vì các ánh xạ này nằm ngoài đống, chúng không góp phần vào sự phân mảnh của phân đoạn dữ liệu.

Việc phân bổ bộ nhớ thông qua ánh xạ ẩn danh có một số lợi ích:

- Không có mối lo ngại về phân mảnh. Khi chương trình không còn cần ánh xạ bộ nhớ ẩn danh, ánh xạ sẽ được hủy ánh xạ và bộ nhớ sẽ được trả về hệ thống ngay lập tức. • Ánh xạ bộ nhớ

  ẩn danh có thể thay đổi kích thước, có quyền điều chỉnh và có thể nhận được lời khuyên giống như ánh xạ thông thường (xem Chương 4).

- Mỗi phân bổ tồn tại trong một ánh xạ bộ nhớ riêng biệt. Không cần phải quản lý heap toàn cục.

Ngoài ra còn có hai nhược điểm khi sử dụng ánh xạ bộ nhớ ẩn danh thay vì heap:

- Mỗi ánh xạ bộ nhớ là bội số nguyên của kích thước trang hệ thống.
  Do đó, các phân bổ không phải là bội số nguyên của các trang về kích thước dẫn đến lãng phí không gian "slack". Không gian slack này là mối quan tâm lớn hơn đối với các phân bổ nhỏ, trong đó không gian lãng phí lớn so với

  kích thước phân bổ. • Việc tạo một ánh xạ bộ nhớ mới phát sinh nhiều chi phí hơn so với việc trả về bộ nhớ từ heap, có thể không liên quan đến bất kỳ tương tác nào của hạt nhân. Phân bổ càng nhỏ thì quan sát này càng hợp lệ.

Tung hứng giữa ưu điểm và nhược điểm, malloc( ) của glibc sử dụng phân đoạn dữ liệu để đáp ứng các phân bổ nhỏ và ánh xạ bộ nhớ ẩn danh để đáp ứng các phân bổ lớn. Ngưỡng có thể định cấu hình được (xem phần “Phân bổ bộ nhớ nâng cao” sau trong chương này) và có thể thay đổi từ bản phát hành glibc này sang bản phát hành khác. Hiện tại, ngưỡng là 128 KB: các phân bổ nhỏ hơn hoặc bằng 128 KB bắt nguồn từ heap, trong khi các phân bổ lớn hơn bắt nguồn từ ánh xạ bộ nhớ ẩn danh.

Tạo bản đồ bộ nhớ ẩn danh

Có lẽ vì bạn muốn buộc sử dụng ánh xạ bộ nhớ trên heap cho một phân bổ cụ thể hoặc có lẽ vì bạn đang viết hệ thống phân bổ bộ nhớ của riêng mình, bạn có thể muốn tự tạo ánh xạ bộ nhớ ẩn danh của riêng mình—hoặc

theo cách này, Linux làm cho nó dễ dàng. Nhớ lại từ Chương 4 rằng lệnh gọi hệ thống mmap( ) tạo ra và hệ thống gọi munmap( ) phá hủy ánh xạ bộ nhớ:

```
#include <sys/mman.h>

trống rỗng * mmap (void *bắt đầu,
                size_t chiều dài,
                int bảo vệ,
                cờ int,
                số nguyên fd,
                bù trừ off_t);

int munmap (void *start, size_t chiều dài);
```

Việc tạo ra một ánh xạ bộ nhớ ẩn danh thực sự dễ hơn việc tạo ra một ánh xạ được hỗ trợ bởi tệp, vì không có tệp nào để mở và quản lý. Sự khác biệt chính
giữa hai loại ánh xạ là sự hiện diện của một lá cờ đặc biệt, biểu thị rằng
việc lập bản đồ là ẩn danh.

Hãy cùng xem một ví dụ:

```
không có *p;

p = mmap (NULL,                                 /* không quan tâm ở đâu */
            512 * 1024,                         /* 512KB */
            ĐỌC PROT | GHI PROT,                /* đọc/ghi */
            MAP_ANONYMOUS | MAP_PRIVATE, /* ẩn danh, riêng tư */
                                                /* fd (bỏ qua) */
            -1, 0); /* bù trừ (bỏ qua) */

nếu (p == MAP_FAILED)
          lỗi ("mmap");
khác
          /* 'p' trỏ tới 512 KB bộ nhớ ẩn danh... */
```

Đối với hầu hết các ánh xạ ẩn danh, các tham số cho mmap() phản ánh ví dụ này, với
ngoại trừ trường hợp truyền vào bất kỳ kích thước nào (tính bằng byte) cho lập trình viên
mong muốn. Các thông số khác thường như sau:

- Tham số đầu tiên, start, được đặt thành NULL, biểu thị rằng ánh xạ ẩn danh
  có thể bắt đầu ở bất kỳ đâu trong bộ nhớ mà hạt nhân mong muốn. Chỉ định một giá trị không phải NULL
  giá trị ở đây là có thể, miễn là nó được căn chỉnh theo trang, nhưng hạn chế khả năng di động. Hiếm khi
  chương trình có quan tâm đến vị trí ánh xạ tồn tại trong bộ nhớ không!
- Tham số prot thường thiết lập cả bit PROT_READ và PROT_WRITE , làm cho
  bản đồ có thể đọc và ghi được. Một bản đồ trống sẽ không có tác dụng gì nếu bạn
  không thể đọc và ghi vào nó. Mặt khác, thực thi mã từ một
  Việc lập bản đồ ẩn danh hiếm khi được mong muốn và việc cho phép điều này sẽ tạo ra một lỗ
  hổng bảo mật tiềm ẩn.
- Tham số cờ đặt bit MAP_ANONYMOUS , làm cho ánh xạ này trở nên ẩn danh và bit MAP_PRIVATE , làm
  cho ánh xạ này trở nên riêng tư.

- Các tham số fd và offset bị bỏ qua khi MAP_ANONYMOUS được đặt. Tuy nhiên, một số hệ thống cũ hơn mong đợi giá trị -1 cho fd, vì vậy, tốt nhất là nên truyền giá trị đó nếu tính di động là mối quan tâm.

Bộ nhớ thu được thông qua ánh xạ ẩn danh trông giống như bộ nhớ thu được thông qua heap. Một lợi ích khi phân bổ từ ánh xạ ẩn danh là các trang đã được điền bằng số không. Điều này xảy ra mà không mất phí, vì hạt nhân ánh xạ các trang ẩn danh của ứng dụng thành một trang đã điền số không thông qua sao chép khi ghi. Do đó, không cần phải memset() bộ nhớ được trả về. Thật vậy, đây là một lợi ích khi sử dụng calloc() trái ngược với malloc() theo sau là memset( ): glibc biết rằng các ánh xạ ẩn danh đã được điền bằng số không và calloc () thỏa mãn từ ánh xạ không yêu cầu điền bằng số không rõ ràng.

Lệnh gọi hệ thống munmap( ) giải phóng một ánh xạ ẩn danh, trả lại bộ nhớ đã phân bổ cho hạt nhân:

```
int ret;

/* hoàn tất với 'p', vì vậy hãy trả lại ánh xạ 512 KB */ ret
= munmap (p, 512 * 1024); if
(ret)
      perror ("munmap");
```

Để biết thêm thông tin về mmap( ), munmap( ) và các ánh xạ nói chung, hãy xem Chương 4.

Mapping /dev/zero Các hệ

thống Unix khác, chẳng hạn như BSD, không có cờ MAP_ANONYMOUS . Thay vào đó, chúng triển khai một giải pháp tương tự bằng cách ánh xạ một tệp thiết bị đặc biệt, /dev/zero. Tệp thiết bị này cung cấp ngữ nghĩa giống hệt với bộ nhớ ẩn danh. Một ánh xạ chứa các trang sao chép khi ghi của tất cả các số không; do đó, hành vi giống như với bộ nhớ ẩn danh.

Linux luôn có một thiết bị /dev/zero và cung cấp khả năng ánh xạ tệp này và lấy bộ nhớ được điền bằng zero. Thật vậy, trước khi giới thiệu MAP_ANONYMOUS, các lập trình viên Linux đã sử dụng cách tiếp cận này. Để cung cấp khả năng tương thích ngược với các phiên bản Linux cũ hơn hoặc khả năng di động sang các hệ thống Unix khác, các nhà phát triển vẫn có thể ánh xạ /dev/zero thay vì tạo ánh xạ ẩn danh. Điều này không khác gì so với ánh xạ bất kỳ tệp nào khác:

```
void *p;
int fd;

/* mở /dev/zero để đọc và ghi */ fd = mở ("/dev/zero",
O_RDWR); nếu (fd < 0) { perror ("mở");
trả về -1;
```

```
}

/* bản đồ [0, kích thước trang) của /dev/zero */
p = mmap (NULL, /* không quan tâm nơi */ getpagesize ( ), /* bản đồ một trang
            */
            PROT_READ | PROT_WRITE, /* đọc/ghi bản đồ */ /*
            MAP_PRIVATE,                  ánh xạ riêng tư */
            fd, /* bản đồ /dev/zero */ 0); /* không có độ lệch */


nếu (p == MAP_FAILED)
          { lỗi ("mmap");
          nếu (đóng (fd))
                    lỗi ("đóng"); trả
          về -1;
}

/* đóng /dev/zero, không cần thiết nữa */
if (close (fd))
          perror ("close");

/* 'p' trỏ tới một trang bộ nhớ, sử dụng nó... */
```

Bộ nhớ được ánh xạ theo cách này tất nhiên sẽ được giải phóng khi sử dụng munmap( ).

Cách tiếp cận này liên quan đến chi phí gọi hệ thống bổ sung khi mở và đóng tệp thiết bị. Do đó, bộ nhớ ẩn danh là giải pháp nhanh hơn.

Phân bổ bộ nhớ nâng cao

Nhiều hoạt động phân bổ được thảo luận trong chương này bị giới hạn và được kiểm soát bởi các tham số hạt nhân mà lập trình viên có thể thay đổi. Để thực hiện như vậy, hãy sử dụng lệnh gọi mallopt() :

```
#include <malloc.h>

int mallopt (int tham số, giá trị int);
```

Một lệnh gọi đến mallopt() đặt tham số liên quan đến quản lý bộ nhớ được chỉ định bởi param thành giá trị được chỉ định bởi value. Khi thành công, lệnh gọi trả về giá trị khác không; khi thất bại, lệnh gọi trả về 0. Lưu ý rằng mallopt() không đặt errno. Nó cũng có xu hướng luôn trả về thành công, vì vậy hãy tránh bất kỳ sự lạc quan nào về việc nhận được thông tin hữu ích từ giá trị trả về.

Linux hiện hỗ trợ sáu giá trị cho param, tất cả đều được định nghĩa trong <malloc.h>:

M_CHECK_ACTION
: Giá trị của biến môi trường MALLOC_CHECK_ (được thảo luận trong phần tiếp theo).

M_MMAP_TỐI ĐA
: Số lượng ánh xạ tối đa mà hệ thống sẽ thực hiện để đáp ứng yêu cầu bộ nhớ động. Khi đạt đến giới hạn này, phân đoạn dữ liệu sẽ được sử dụng cho tất cả các phân bổ, cho đến khi một trong những ánh xạ này được giải phóng. Giá trị 0 vô hiệu hóa tất cả việc sử dụng ánh xạ ẩn danh làm cơ sở cho phân bổ bộ nhớ động.

M_MMAP_NGƯỜI
: Ngưỡng kích thước (được đo bằng byte) mà yêu cầu phân bổ sẽ được thực hiện được thỏa mãn thông qua một ánh xạ ẩn danh thay vì phân đoạn dữ liệu. Lưu ý rằng phân bổ nhỏ hơn ngưỡng này cũng có thể được đáp ứng thông qua lệnh map-ping ẩn danh theo quyết định của hệ thống. Giá trị 0 cho phép sử dụng lệnh map-ping ẩn danh ánh xạ cho tất cả các phân bổ, vô hiệu hóa hiệu quả việc sử dụng phân đoạn dữ liệu cho phân bổ bộ nhớ động.

M_MXNHANH
: Kích thước tối đa (tính bằng byte) của một thùng chứa nhanh. Thùng chứa nhanh là các khối bộ nhớ đặc biệt trong heap không bao giờ được hợp nhất với các khối liền kề và không bao giờ được trả lại cho hệ thống, cho phép phân bổ rất nhanh với chi phí tăng phân mảnh. Giá trị 0 sẽ vô hiệu hóa mọi việc sử dụng thùng chứa nhanh.

M_TOP_BÀN_BÀN
: Lượng đệm (tính bằng byte) được sử dụng khi điều chỉnh kích thước của phân đoạn dữ liệu. Bất cứ khi nào glibc sử dụng brk( ) để tăng kích thước của phân đoạn dữ liệu, nó có thể yêu cầu nhiều bộ nhớ hơn mức cần thiết, với hy vọng làm giảm nhu cầu về một cuộc gọi brk() bổ sung trong tương lai gần. Tương tự như vậy, bất cứ khi nào glibc thu nhỏ kích thước của phân đoạn dữ liệu, nó có thể giữ thêm bộ nhớ, trả lại ít hơn một chút so với nó nếu không thì. Những byte bổ sung này là phần đệm. Giá trị 0 vô hiệu hóa mọi việc sử dụng của đệm.

M_TRIM_NGƯỠNG
: Lượng bộ nhớ trống tối thiểu (tính bằng byte) được phép ở đầu dữ liệu phân đoạn. Nếu số lượng giảm xuống dưới ngưỡng này, glibc sẽ gọi brk() để cung cấp trả lại bộ nhớ cho hạt nhân.

Tiêu chuẩn XPG, định nghĩa một cách lỏng lẻo mallopt( ), chỉ định ba tham số khác: M_GRAIN, M_KEEP và M_NLBLKS. Linux định nghĩa các tham số này, nhưng thiết lập chúng giá trị không có hiệu lực. Xem Bảng 8-1 để biết danh sách đầy đủ tất cả các tham số hợp lệ, mặc định của chúng giá trị và phạm vi giá trị được chấp nhận của chúng.

Bảng 8-1. tham số mallopt()

Tham số	Nguồn gốc	Giá trị mặc định	Giá trị hợp lệ	Giá trị đặc biệt
HÀNH ĐỘNG KIỂM TRA	Dành riêng cho Linux	0	0 – 2	
M_HỎNG	Tiêu chuẩn XPG	Không được hỗ trợ trên Linux	>= 0	
GIỮ LẠI	Tiêu chuẩn XPG	Không được hỗ trợ trên Linux	>= 0	

Bảng 8-1. tham số mallopt() (tiếp theo)

Tham số	Nguồn gốc	Giá trị mặc định	Giá trị hợp lệ	Giá trị đặc biệt
M_MMAP_TỐI ĐA	Dành riêng cho Linux	64 * 1024	>= 0	0 vô hiệu hóa việc sử dụng mmap()
M_MMAP_NGƯỜI	Dành riêng cho Linux	128 * 1024	>= 0	0 vô hiệu hóa việc sử dụng heap
M_MXNHANH	Tiêu chuẩn XPG	64	0 - 80	0 vô hiệu hóa thùng rác nhanh
M_NLBLKS	Tiêu chuẩn XPG	Không được hỗ trợ trên Linux >= 0		
M_TOP_BÀN_BÀN	Dành riêng cho Linux	0	>= 0	0 vô hiệu hóa phần đệm

Các chương trình phải thực hiện bất kỳ lệnh gọi mallopt() nào trước khi gọi malloc() lần đầu tiên hoặc bất kỳ giao diện phân bổ bộ nhớ nào khác. Cách sử dụng rất đơn giản:

```
int ret;

/* sử dụng mmap( ) cho tất cả các phân bổ trên 64 KB */
ret = mallopt (M_MMAP_THRESHOLD, 64 * 1024);
nếu (!ret)
          fprintf (stderr, "mallopt không thành công!\n");
```

Tinh chỉnh với malloc_usable_size() và malloc_trim()

Linux cung cấp một số chức năng cung cấp khả năng kiểm soát cấp thấp đối với bộ nhớ glibc hệ thống phân bổ. Chức năng đầu tiên như vậy cho phép một chương trình hỏi có bao nhiêu byte mà một phân bổ bộ nhớ nhất định chứa:

```
#include <malloc.h>

kích thước_t malloc_usable_size (void *ptr);
```

Một cuộc gọi thành công đến malloc_usable_size() trả về kích thước phân bổ thực tế của khối bộ nhớ được trỏ tới bởi ptr. Bởi vì glibc có thể làm tròn các phân bổ để phù hợp trong một khối hiện có hoặc ánh xạ ẩn danh, không gian có thể sử dụng trong một phân bổ có thể lớn hơn yêu cầu. Tất nhiên, phân bổ sẽ không bao giờ nhỏ hơn được yêu cầu. Sau đây là ví dụ về cách sử dụng hàm:

```
kích thước_t len = 21;
kích thước_t kích thước;
ký tự *buf;

buf = malloc (chiều dài);
nếu (!buf) {
          lỗi ("malloc");
          trả về -1;
}

kích thước = malloc_usable_size (buf);

/* chúng ta thực sự có thể sử dụng 'size' byte của 'buf'... */
```

Chức năng thứ hai trong hai chức năng cho phép chương trình buộc glibc trả lại toàn bộ bộ nhớ có thể giải phóng ngay lập tức cho nhân:

```
#include <malloc.h>

int malloc_trim (kích thước đệm_t);
```

Một lệnh gọi thành công đến malloc_trim() sẽ thu nhỏ phân đoạn dữ liệu nhiều nhất có thể, trừ đi các byte đệm , được dành riêng. Sau đó, nó trả về 1. Khi lệnh gọi không thành công, nó trả về 0. Thông thường, glibc sẽ tự động thu nhỏ như vậy, bất cứ khi nào bộ nhớ có thể giải phóng đạt đến M_TRIM_THRESHOLD byte. Nó sử dụng một phần đệm của M_TOP_PAD.

Bạn hầu như không bao giờ muốn sử dụng hai hàm này cho bất kỳ mục đích nào khác ngoài mục đích gỡ lỗi hoặc giáo dục. Chúng không thể di chuyển và để lộ các chi tiết cấp thấp của hệ thống phân bổ bộ nhớ glibc cho chương trình của bạn.

Gỡ lỗi phân bổ bộ nhớ

Các chương trình có thể thiết lập biến môi trường MALLOC_CHECK_ để cho phép gỡ lỗi nâng cao trong hệ thống con bộ nhớ. Các kiểm tra gỡ lỗi bổ sung sẽ làm giảm hiệu quả phân bổ bộ nhớ, nhưng chi phí này thường đáng giá trong giai đoạn gỡ lỗi của quá trình phát triển ứng dụng.

Vì biến môi trường kiểm soát việc gỡ lỗi, nên không cần phải biên dịch lại chương trình của bạn. Ví dụ, bạn có thể chỉ cần đưa ra lệnh như sau:

```
$ MALLOC_CHECK_=1 ./bánh lái
```

Nếu MALLOC_CHECK_ được đặt thành 0, hệ thống bộ nhớ sẽ bỏ qua mọi lỗi một cách âm thầm. Nếu được đặt thành 1, một thông báo có thông tin sẽ được in ra stderr. Nếu được đặt thành 2, chương trình sẽ ngay lập tức bị chấm dứt thông qua abort( ). Vì MALLOC_CHECK_ thay đổi hành vi của chương trình đang chạy, nên các chương trình setuid sẽ bỏ qua biến này.

Thu thập số liệu thống kê

Linux cung cấp hàm mallinfo() để lấy số liệu thống kê liên quan đến hệ thống phân bổ bộ nhớ:

```
#include <malloc.h>

cấu trúc mallinfo mallinfo (void);
```

Một lệnh gọi đến mallinfo() trả về số liệu thống kê trong một cấu trúc mallinfo . Cấu trúc được trả về theo giá trị, không phải thông qua một con trỏ. Nội dung của nó cũng được định nghĩa trong <malloc.h>:

```
/* tất cả các kích thước tính
bằng byte */ struct
          mallinfo          /* kích thước của phân đoạn dữ liệu được malloc sử dụng */
          { int arena; int ordblks; /* số lượng các khối trống */
```

```
        int smblks; /* số lượng thùng chứa nhanh */
        int hblks; /* số lượng ánh xạ ẩn danh */ int hblkhd; /*
        kích thước của các ánh xạ ẩn danh */ int usmblks; /*
        tổng kích thước được phân bổ tối đa */ int fsmblks; /*
        kích thước của các thùng chứa nhanh khả dụng */ int
        uordblks; /* kích thước của tổng không gian được phân bổ
        */ int fordblks; /* kích thước của các khối khả dụng
        */ int keepcost; /* kích thước của không gian có thể cắt bớt */
};
```

Cách sử dụng rất đơn giản:

```
cấu trúc mallinfo m;

m = mallinfo ( );

printf ("các khối miễn phí: %d\n", m.ordblks);
```

Linux cũng cung cấp hàm malloc_stats() , hàm này in các số liệu thống kê liên quan đến bộ nhớ vào stderr:

```
#include <malloc.h>

void malloc_stats (void);
```

Gọi malloc_stats() trong một chương trình sử dụng nhiều bộ nhớ sẽ tạo ra một số con số lớn:

```
Đầu trường 0:
byte hệ thống          = 865939456
đang sử dụng           = 851988200
byte Tổng số (bao gồm
mmap): byte hệ thống = 3216519168 byte
đang sử dụng = 3202567912 vùng mmap tối
đa = 65536 byte mmap tối đa = 2350579712
```

Phân bổ dựa trên ngăn xếp

Cho đến nay, tất cả các cơ chế phân bổ bộ nhớ động mà chúng ta đã nghiên cứu đều sử dụng heap hoặc ánh xạ bộ nhớ để có được bộ nhớ động. Chúng ta nên mong đợi điều này vì heap và ánh xạ bộ nhớ có bản chất động rõ ràng. Cấu trúc phổ biến khác trong không gian địa chỉ của chương trình, ngăn xếp, là nơi chứa các biến tự động của chương trình .

Tuy nhiên, không có lý do gì mà một lập trình viên không thể sử dụng ngăn xếp để phân bổ bộ nhớ động. Miễn là việc phân bổ không tràn ngăn xếp, thì cách tiếp cận như vậy sẽ dễ dàng và sẽ hoạt động khá tốt. Để thực hiện phân bổ bộ nhớ động từ ngăn xếp, hãy sử dụng lệnh gọi hệ thống alloca() :

```
#include <alloca.h>

void * alloca(kích thước_t);
```

Khi thành công, lệnh gọi alloca( ) trả về một con trỏ tới kích thước byte bộ nhớ. Bộ nhớ này nằm trên ngăn xếp và được tự động giải phóng khi hàm gọi trả về. Một số triển khai trả về NULL khi lỗi, nhưng hầu hết các triển khai alloca( ) không thể lỗi hoặc không thể báo cáo lỗi. Lỗi được biểu hiện dưới dạng tràn ngăn xếp.

Cách sử dụng giống hệt malloc( ), nhưng bạn không cần (thực ra là không được) giải phóng bộ nhớ được phân bổ. Sau đây là ví dụ về một hàm mở một tệp nhất định trong thư mục cấu hình của hệ thống, có thể là /etc, nhưng được xác định tại thời điểm biên dịch. Hàm phải phân bổ một bộ đệm mới, sao chép thư mục cấu hình hệ thống vào bộ đệm, sau đó nối bộ đệm này với tên tệp được cung cấp:

```
int open_sysconf (const char *file, int cờ, int chế độ) {

        const char *etc; = SYSCONF_DIR; /* "/etc/" */
        char *name;

        tên = alloca (strlen (v.v.) + strlen (tệp) + 1);
        strcpy (tên, v.v.);
        strcat (tên, tệp);

        trả về mở (tên, cờ, chế độ);
}
```

Khi trả về, bộ nhớ được phân bổ với alloca() sẽ tự động được giải phóng khi ngăn xếp được giải phóng trở lại hàm gọi. Điều này có nghĩa là bạn không thể sử dụng bộ nhớ này sau khi hàm gọi alloca() trả về! Tuy nhiên, vì bạn không phải thực hiện bất kỳ dọn dẹp nào bằng cách gọi free( ), nên mã kết quả sẽ sạch hơn một chút. Sau đây là cùng một hàm được triển khai bằng malloc( ):

```
int open_sysconf (const char *file, int cờ, int chế độ) {

        const char *etc = SYSCONF_DIR; /* "/etc/" */
        char *name;
        int fd;

        tên = malloc (strlen (v.v.) + strlen (tệp) + 1); nếu (!
        tên) { lỗi
                ("malloc"); trả về
                -1;
        }

        strcpy (tên, v.v.);
        strcat (tên, tệp);
        fd = mở (tên, cờ, chế độ); miễn
        phí (tên);

        trả về fd;
}
```

Lưu ý rằng bạn không nên sử dụng bộ nhớ được phân bổ alloca( ) trong các tham số cho lệnh gọi hàm, vì bộ nhớ được phân bổ sau đó sẽ tồn tại ở giữa không gian ngăn xếp dành riêng cho các tham số hàm. Ví dụ, những điều sau đây là không được phép:

```
/* KHÔNG LÀM ĐIỀU NÀY! */ ret
= foo (x, alloca (10));
```

Giao diện alloca( ) có lịch sử không ổn định. Trên nhiều hệ thống, giao diện này hoạt động kém hoặc dẫn đến hành vi không xác định. Trên các hệ thống có ngăn xếp nhỏ và có kích thước cố định, sử dụng alloca( ) là cách dễ dàng để tràn ngăn xếp và giết chết chương trình của bạn. Trên các hệ thống khác, alloca( ) thậm chí còn không tồn tại. Theo thời gian, các triển khai lỗi và không nhất quán đã khiến alloca( ) có tiếng xấu.

Vì vậy, nếu chương trình của bạn phải duy trì tính di động, bạn nên tránh alloca( ). Tuy nhiên, trên Linux, alloca() là một công cụ hữu ích tuyệt vời và ít được sử dụng. Nó hoạt động cực kỳ tốt—trên nhiều kiến trúc, việc phân bổ thông qua alloca() chỉ làm tăng con trỏ ngăn xếp—và hiệu suất cao hơn malloc( ). Đối với các phân bổ nhỏ trong mã dành riêng cho Linux, alloca() có thể mang lại hiệu suất tăng tuyệt vời.

Nhân đôi chuỗi trên ngăn xếp Một cách sử

dụng rất phổ biến của alloca( ) là nhân đôi tạm thời một chuỗi. Ví dụ:

```
/* chúng ta muốn sao chép 'bài hát' */
char *dup;

dup = alloca (strlen (bài hát) + 1);
strcpy (dup, bài hát);

/* thao tác 'dup'... */

return; /* 'dup' được tự động giải phóng */
```

Do tần suất cần thiết này và lợi ích về tốc độ mà alloca( ) mang lại,
Hệ thống Linux cung cấp các biến thể của strdup() để sao chép chuỗi đã cho vào ngăn xếp:

```
#define _GNU_SOURCE
#include <string.h>

ký tự    strdupa (const char *s);
* ký tự *strndupa (const char *s, size_t n);
```

Một lệnh gọi đến strdupa( ) trả về một bản sao của s. Một lệnh gọi đến strndupa( ) sao chép tối đa n ký tự của s. Nếu s dài hơn n, quá trình sao chép dừng lại ở n và hàm sẽ thêm một byte null. Các hàm này cung cấp cùng lợi ích như alloca( ). Chuỗi được sao chép sẽ tự động được giải phóng khi hàm gọi trả về.

POSIX không định nghĩa các hàm alloca( ), strdupa( ), hoặc strndupa( ) , và hồ sơ của chúng trên các hệ điều hành khác không đầy đủ. Nếu tính di động là mối quan tâm, việc sử dụng các hàm này không được khuyến khích. Tuy nhiên, trên Linux, alloca( ) và các hàm tương tự thực hiện

khá tốt và có thể cung cấp hiệu suất tăng cường tuyệt vời, thay thế quá trình phân bổ bộ nhớ động phức tạp chỉ bằng một điều chỉnh đơn giản của con trỏ khung ngăn xếp.

Mảng có độ dài thay đổi C99

giới thiệu mảng có độ dài thay đổi (VLA), là mảng có hình học được thiết lập tại thời điểm chạy, chứ không phải tại thời điểm biên dịch. GNUC đã hỗ trợ mảng có độ dài thay đổi trong một thời gian, nhưng hiện tại C99 đã chuẩn hóa chúng, nên có nhiều động lực hơn để sử dụng chúng. VLA tránh được chi phí cấp phát bộ nhớ động theo cách tương tự như alloca( ).

Việc sử dụng chúng chính xác như những gì bạn mong đợi:

```
đối với (i = 0; i < n; ++i)
        { char foo[i + 1];

        /* sử dụng 'foo'... */
}
```

Trong đoạn mã này, foo là một mảng các ký tự có kích thước biến i + 1. Ở mỗi lần lặp của vòng lặp, foo được tạo động và tự động dọn dẹp khi nó nằm ngoài phạm vi. Nếu chúng ta sử dụng alloca( ) thay vì VLA, bộ nhớ sẽ không được giải phóng cho đến khi hàm trả về. Sử dụng VLA đảm bảo rằng bộ nhớ được giải phóng ở mỗi lần lặp của vòng lặp. Do đó, sử dụng VLA tiêu thụ ít nhất là n byte, trong khi alloca( ) sẽ tiêu thụ n*(n+1)/2 byte.

Sử dụng mảng có độ dài thay đổi, chúng ta có thể viết lại hàm open_sysconf() như sau:

```
int open_sysconf (const char *file, int cờ, int chế độ) {

        const char *etc; = SYSCONF_DIR; /* "/etc/" */ char
        name[strlen (etc) + strlen (tệp) + 1];

        strcpy (tên, v.v.);
        strcat (tên, tệp);

        trả về mở (tên, cờ, chế độ);
}
```

Sự khác biệt chính giữa alloca() và mảng có độ dài thay đổi là bộ nhớ thu được thông qua mảng trước tồn tại trong suốt thời gian của hàm, trong khi bộ nhớ thu được thông qua mảng sau tồn tại cho đến khi biến giữ nằm ngoài phạm vi, có thể là trước khi hàm hiện tại trả về. Điều này có thể được hoan nghênh hoặc không được hoan nghênh. Trong vòng lặp for mà chúng ta vừa xem xét, việc thu hồi bộ nhớ trong mỗi lần lặp lại vòng lặp sẽ làm giảm mức tiêu thụ bộ nhớ ròng mà không có bất kỳ tác dụng phụ nào (chúng ta không cần bộ nhớ bổ sung treo xung quanh). Tuy nhiên, nếu vì lý do nào đó, chúng ta muốn bộ nhớ tồn tại lâu hơn một lần lặp lại vòng lặp duy nhất, thì việc sử dụng alloca() sẽ hợp lý hơn.

Việc kết hợp alloca() và các mảng có độ dài thay đổi trong một hàm duy nhất có thể mời gọi hành vi kỳ lạ. Chơi an toàn và sử dụng một trong hai trong một chức năng.

Chọn cơ chế phân bổ bộ nhớ

Vô số tùy chọn phân bổ bộ nhớ được thảo luận trong chương này có thể để lại các lập trình viên tự hỏi chính xác giải pháp nào là tốt nhất cho một công việc nhất định. Trong phần lớn tình huống, malloc( ) là lựa chọn tốt nhất của bạn. Tuy nhiên, đôi khi, một cách tiếp cận khác cung cấp một công cụ tốt hơn. Bảng 8-2 tóm tắt các hướng dẫn để lựa chọn phân bổ cơ chế.

Bảng 8-2. Các phương pháp phân bổ bộ nhớ trong Linux

Phương pháp phân bổ	Ưu điểm	Nhược điểm
malloc()	Dễ dàng, đơn giản, phổ biến.	Bộ nhớ trả về không nhất thiết phải được đặt về 0.
calloc()	Làm cho việc phân bổ mảng trở nên đơn giản, số không bộ nhớ được trả về.	Giao diện phức tạp nếu không phân bổ mảng.
phân bổ lại( )	Thay đổi kích thước phân bổ hiện có.	Chỉ hữu ích cho việc thay đổi kích thước hiện có phân bổ.
brk( ) và sbrk( )	Cho phép kiểm soát chặt chẽ đống dữ liệu.	Quá thấp so với hầu hết người dùng.
Ánh xạ bộ nhớ ẩn danh	Dễ dàng làm việc, có thể chia sẻ, cho phép nhà phát triển điều chỉnh mức độ bảo vệ và cung cấp lời khuyên; tối ưu cho các bản đồ lớn.	Không tối ưu cho các phân bổ nhỏ; malloc() tự động sử dụng ánh xạ bộ nhớ ẩn danh khi ở trạng thái tối ưu.
căn chỉnh posix_memalign ( )	Phân bổ bộ nhớ phù hợp với bất kỳ ranh giới hợp lý nào.	Tương đối mới và do đó tính di động là có thể nghi ngờ; quá mức trừ khi căn chỉnh mối quan tâm đang cấp bách.
memalign( ) và valloc( )	Phổ biến hơn trên các hệ thống Unix khác hơn posix_memalign( ).	Không phải là tiêu chuẩn POSIX, cung cấp ít sự liên kết hơn kiểm soát hơn posix_memalign( ).
phân bổ( )	Phân bổ rất nhanh, không cần phải rõ ràng bộ nhớ trống; tuyệt vời cho việc phân bổ nhỏ.	Không thể trả về lỗi, không tốt cho lớn phân bổ, bị hỏng trên một số hệ thống Unix.
Mảng có độ dài thay đổi	Giống như alloca( ), nhưng giải phóng bộ nhớ khi mảng nằm ngoài phạm vi, không phải khi hàm trả về.	Chỉ hữu ích cho các mảng; hành vi giải phóng alloca( ) có thể được ưa thích hơn trong một số tình huống; ít phổ biến hơn trên Unix khác hệ thống hơn alloca( ).

Cuối cùng, chúng ta đừng quên phương án thay thế cho tất cả các tùy chọn này: tự động và tĩnh phân bổ bộ nhớ. Phân bổ các biến tự động trên ngăn xếp hoặc các biến toàn cục trên heap, thường dễ hơn và không yêu cầu lập trình viên quản lý các con trỏ và lo lắng về việc giải phóng bộ nhớ.

Thao tác bộ nhớ

Ngôn ngữ C cung cấp một họ các hàm để thao tác các byte thô của bộ nhớ. Các hàm này hoạt động theo nhiều cách tương tự như các giao diện thao tác chuỗi như strcmp() và strcpy( ), nhưng chúng dựa vào kích thước bộ đệm do người dùng cung cấp thay vì giả định rằng các chuỗi được kết thúc bằng null. Lưu ý rằng không có hàm nào trong số các hàm này có thể trả về lỗi. Việc ngăn ngừa lỗi tùy thuộc vào lập trình viên—truyền vào vùng bộ nhớ sai và không có giải pháp thay thế nào khác, ngoại trừ vi phạm phân đoạn kết quả!

Thiết lập Byte

Trong số các hàm xử lý bộ nhớ, hàm phổ biến nhất là memset( ):

```
#include <chuỗi.h>

void * memset (void *s, int c, size_t n);
```

Một lệnh gọi đến memset() sẽ đặt n byte bắt đầu từ s thành byte c và trả về s. Một cách sử dụng thường xuyên là đưa một khối bộ nhớ về 0:

```
/* đưa về 0 [s,s+256) */
memset (s, '\0', 256);
```

bzero() là một giao diện cũ hơn, đã lỗi thời được BSD giới thiệu để thực hiện cùng một tác vụ. Mã mới nên sử dụng memset( ), nhưng Linux cung cấp bzero() để tương thích ngược và khả năng di động với các hệ thống khác:

```
#include <chuỗi.h>

void bzero (void *s, size_t n);
```

Lời gọi sau đây giống hệt với ví dụ memset() trước đó :

```
bzero (s, 256);
```

Lưu ý rằng bzero( )—cùng với các giao diện b khác —yêu cầu tiêu đề <strings.h> chứ không phải <string.h>.

Không sử dụng memset() nếu bạn có thể sử dụng calloc( )! Tránh phân bổ bộ nhớ bằng malloc( ), sau đó ngay lập tức xóa bộ nhớ bằng memset( ). Mặc dù kết quả có thể giống nhau, nhưng việc loại bỏ hai hàm để dùng một hàm calloc( ) duy nhất, trả về bộ nhớ bằng 0, sẽ tốt hơn nhiều. Bạn không chỉ thực hiện ít hơn một lệnh gọi hàm mà calloc() còn có thể lấy được bộ nhớ đã bằng 0 từ kernel. Trong trường hợp đó, bạn tránh được việc đặt thủ công từng byte thành 0 và cải thiện hiệu suất.

So sánh các byte

Tương tự như strcmp( ), memcmp() so sánh hai khối bộ nhớ để xác định tính tương đương:

```
#include <chuỗi.h>

int memcmp (const void *s1, const void *s2, size_t n);
```

Lệnh gọi sẽ so sánh n byte đầu tiên của s1 với s2 và trả về 0 nếu các khối bộ nhớ tương đương, trả về giá trị nhỏ hơn 0 nếu s1 nhỏ hơn s2 và trả về giá trị lớn hơn 0 nếu s1 lớn hơn s2.

BSD một lần nữa cung cấp một giao diện hiện đã lỗi thời thực hiện phần lớn nhiệm vụ tương tự:

```
#include <chuỗi.h>

int bcmp (const void *s1, const void *s2, size_t n);
```

Lệnh bcmp( ) sẽ so sánh n byte đầu tiên của s1 với s2, trả về 0 nếu các khối bộ nhớ tương đương và trả về giá trị khác không nếu chúng khác nhau.

Do cấu trúc đệm (xem “Các mối quan tâm về căn chỉnh khác” ở đầu chương này), việc so sánh hai cấu trúc để tìm sự tương đương thông qua memcmp() hoặc bcmp() là không đáng tin cậy. Có thể có rác chưa được khởi tạo trong phần đệm khác nhau giữa hai trường hợp giống hệt nhau của một cấu trúc. Do đó, mã như sau không an toàn:

```
/* hai chiếc xuồng có giống hệt nhau không? (BỊ
HỎNG) */ int compare_dinghies (struct dinghy *a, struct dinghy *b)
{
         trả về memcmp (a, b, sizeof (struct dinghy));
}
```

Thay vào đó, các lập trình viên muốn so sánh các cấu trúc nên so sánh từng phần tử của các cấu trúc, từng phần tử một. Cách tiếp cận này cho phép một số tối ưu hóa, nhưng chắc chắn là nhiều việc hơn so với cách tiếp cận không an toàn memcmp() . Sau đây là mã tương đương:

```
/* hai chiếc xuồng có giống hệt nhau
không? */ int compare_dinghies (struct dinghy *a, struct dinghy
*b) {
         int ret;

         nếu (a->nr_oars < b->nr_oars)
                  trả về -1;
         nếu (a->nr_oars > b->nr_oars)
                  trả về 1;

         ret = strcmp (a->tên_thuyền, b->tên_thuyền); nếu
         (ret)
               trả về ret;

         /* và cứ thế cho mỗi thành viên... */
}
```

Di chuyển byte

memmove() sao chép n byte đầu tiên của src tới dst, trả về dst:

```
#include <chuỗi.h>

trống rỗng * memmove (void *dst, const void *src, size_t n);
```

Một lần nữa, BSD cung cấp một giao diện không còn được sử dụng nữa để thực hiện cùng một tác vụ:

```
#include <chuỗi.h>

void bcopy (const void *src, void *dst, size_t n);
```

Lưu ý rằng mặc dù cả hai hàm đều sử dụng cùng một tham số, nhưng thứ tự của hai tham số đầu tiên bị đảo ngược trong bcopy( ).

Cả bcopy() và memmove() đều có thể xử lý an toàn các vùng bộ nhớ chồng lấn (ví dụ, nếu một phần của dst nằm bên trong src). Ví dụ, điều này cho phép các byte bộ nhớ dịch chuyển lên hoặc xuống trong một vùng nhất định. Vì tình huống này hiếm khi xảy ra và một lập trình viên sẽ biết nếu trường hợp đó xảy ra, nên chuẩn C định nghĩa một biến thể của memmove() không hỗ trợ các vùng bộ nhớ chồng lấn. Biến thể này có khả năng nhanh hơn:

```
#include <chuỗi.h>

trống rỗng * memcpy (void *dst, const void *src, size_t n); Hàm này
```

hoạt động giống hệt như memmove( ), ngoại trừ dst và src không được chồng lấn nhau.
Nếu có thì kết quả vẫn chưa được xác định.

Một hàm sao chép an toàn khác là memccpy( ):

```
#include <chuỗi.h>

trống rỗng * memccpy (void *dst, const void *src, int c, size_t n);
```

Hàm memccpy() hoạt động giống như hàm memcpy( ), ngoại trừ việc nó dừng sao chép nếu hàm tìm thấy byte c trong n byte đầu tiên của src. Lệnh gọi trả về một con trỏ đến byte tiếp theo trong dst sau c hoặc NULL nếu không tìm thấy c .

Cuối cùng, bạn có thể sử dụng mempcpy() để duyệt bộ nhớ:

```
#define _GNU_SOURCE
#include <string.h>

trống rỗng * mempcpy (void *dst, const void *src, size_t n);
```

Hàm mempcpy() thực hiện giống như hàm memcpy( ), ngoại trừ việc nó trả về một con trỏ tới byte tiếp theo sau byte cuối cùng được sao chép. Điều này hữu ích nếu một tập dữ liệu được sao chép tới các vị trí bộ nhớ liên tiếp —nhưng nó không phải là cải tiến đáng kể vì giá trị trả về chỉ là dst + n. Hàm này dành riêng cho GNU.

Tìm kiếm byte

Các hàm memchr() và memrchr() xác định vị trí một byte nhất định trong một khối bộ nhớ:

```
#include <chuỗi.h>

void * memchr (const void *s, int c, size_t n);
```

Hàm memchr () quét n byte bộ nhớ được trỏ tới bởi s để tìm ký tự c, được hiểu là một ký tự không dấu:

```
#define _GNU_SOURCE
#include <string.h>

void * memrchr (const void *s, int c, size_t n); Lệnh
```

gọi trả về một con trỏ tới byte đầu tiên để khớp với c hoặc NULL nếu không tìm thấy c .

Hàm memrchr() giống như hàm memchr() , ngoại trừ việc nó tìm kiếm ngược từ cuối n byte được trỏ tới bởi s thay vì tìm kiếm tiến từ phía trước. Không giống như memchr( ), memrchr() là một phần mở rộng GNU và không phải là một phần của ngôn ngữ C.

Đối với các nhiệm vụ tìm kiếm phức tạp hơn, hàm có tên rất hay là memmem() sẽ tìm kiếm một khối bộ nhớ để tìm một mảng byte tùy ý:

```
#define _GNU_SOURCE
#include <string.h>

trống rỗng * memmem (const void *haystack,
                     size_t haystacklen,
                     const void *needle,
                     size_t needlelen);
```

Hàm memmem () trả về một con trỏ đến lần xuất hiện đầu tiên của khối con needle, có độ dài needlelen byte, trong khối bộ nhớ haystack, có độ dài haystacklen byte. Nếu hàm không tìm thấy needle trong haystack, nó trả về NULL.
Chức năng này cũng là một phần mở rộng của GNU.

Bytes Frobnicating

Thư viện C của Linux cung cấp một giao diện để xử lý các byte dữ liệu một cách đơn giản:

```
#define _GNU_SOURCE
#include <string.h>

void * memfrob (void *s, size_t n);
```

Một lệnh gọi đến memfrob( ) sẽ che khuất n byte đầu tiên của bộ nhớ bắt đầu từ s bằng cách thực hiện phép toán OR loại trừ (XOR) mỗi byte với số 42. Lệnh gọi trả về s.

Hiệu ứng của lệnh gọi đến memfrob() có thể được đảo ngược bằng cách gọi lại memfrob() trên cùng một vùng bộ nhớ. Do đó, đoạn mã sau đây là no-op đối với secret:

```
memfrob (memfrob (bí mật, len), len);
```

Chức năng này không phải là sự thay thế thích hợp (hoặc thậm chí là kém) cho mã hóa; việc sử dụng nó chỉ giới hạn ở việc làm tối nghĩa các chuỗi. Nó dành riêng cho GNU.

Khóa bộ nhớ Linux triển khai

phân trang theo nhu cầu, nghĩa là các trang được hoán đổi từ đĩa khi cần và hoán đổi ra đĩa khi không còn cần nữa. Điều này cho phép không gian địa chỉ ảo của các tiến trình trên hệ thống không có mối quan hệ trực tiếp với tổng lượng bộ nhớ vật lý, vì không gian hoán đổi trên đĩa có thể tạo ra ảo giác về nguồn cung cấp bộ nhớ vật lý gần như vô hạn.

Việc hoán đổi này diễn ra một cách minh bạch và các ứng dụng thường không cần quan tâm đến (hoặc thậm chí không cần biết về) hành vi phân trang của hạt nhân Linux. Tuy nhiên, có hai tình huống mà các ứng dụng có thể muốn tác động đến hành vi phân trang của hệ thống:

Chủ nghĩa quyết định

Các ứng dụng có ràng buộc về thời gian yêu cầu hành vi xác định. Nếu một số lần truy cập bộ nhớ dẫn đến lỗi trang—gây ra các hoạt động I/O đĩa tốn kém—các ứng dụng có thể vượt quá nhu cầu về thời gian của chúng. Bằng cách đảm bảo rằng các trang cần thiết luôn nằm trong bộ nhớ vật lý và không bao giờ được phân trang vào đĩa, ứng dụng có thể đảm bảo rằng các lần truy cập bộ nhớ sẽ không dẫn đến lỗi trang, mang lại tính nhất quán, tính xác định và hiệu suất được cải thiện.

Bảo mật

Nếu bí mật riêng tư được lưu trong bộ nhớ, bí mật có thể được phân trang và lưu trữ không được mã hóa trên đĩa. Ví dụ, nếu khóa riêng tư của người dùng thường được lưu trữ được mã hóa trên đĩa, một bản sao không được mã hóa của khóa trong bộ nhớ có thể nằm trong tệp hoán đổi. Trong môi trường bảo mật cao, hành vi này có thể không được chấp nhận. Các ứng dụng có thể gặp vấn đề này có thể yêu cầu bộ nhớ chứa khóa luôn nằm trong bộ nhớ vật lý.

Tất nhiên, việc thay đổi hành vi của hạt nhân có thể dẫn đến tác động tiêu cực đến hiệu suất hệ thống tổng thể. Tính xác định hoặc bảo mật của một ứng dụng có thể được cải thiện, nhưng trong khi các trang của nó bị khóa vào bộ nhớ, các trang của ứng dụng khác sẽ được phân trang thay thế. Hạt nhân, nếu chúng ta tin tưởng vào thiết kế của nó, luôn chọn trang tối ưu để hoán đổi—tức là trang ít có khả năng được sử dụng nhất trong tương lai—vì vậy khi bạn thay đổi hành vi của nó, nó phải hoán đổi một trang không tối ưu.

Khóa một phần của không gian địa chỉ POSIX

1003.1b-1993 định nghĩa hai giao diện để "khóa" một hoặc nhiều trang vào bộ nhớ vật lý, đảm bảo rằng chúng không bao giờ được phân trang ra đĩa. Giao diện đầu tiên khóa một khoảng thời gian địa chỉ nhất định:

```
#include <sys/mman.h>

int mlock (const void *addr, size_t len);
```

Một lệnh gọi đến mlock() sẽ khóa bộ nhớ ảo bắt đầu từ addr và mở rộng thành len byte trong bộ nhớ vật lý. Khi thành công, lệnh gọi trả về 0; khi thất bại, lệnh gọi trả về -1 và đặt errno thành giá trị phù hợp.

Cuộc gọi thành công sẽ khóa tất cả các trang vật lý có chứa [addr,addr+len) trong bộ nhớ. Ví dụ, nếu lệnh gọi chỉ định một byte đơn, toàn bộ trang mà byte đó nằm trong đó sẽ bị khóa vào bộ nhớ. Chuẩn POSIX quy định rằng addr phải được căn chỉnh theo ranh giới trang. Linux không thực thi yêu cầu này, âm thầm làm tròn addr xuống trang gần nhất nếu cần. Tuy nhiên, các chương trình yêu cầu khả năng di chuyển sang các hệ thống khác phải đảm bảo rằng addr nằm trên ranh giới trang.

Các mã lỗi hợp lệ bao gồm:

EINVAL
: Tham số len là số âm.

ENOMEM
: Người gọi đã cố gắng khóa nhiều trang hơn giới hạn tài nguyên RLIMIT_MEMLOCK cho phép (xem phần sau "Giới hạn khóa").

EPERM
: Giới hạn tài nguyên RLIMIT_MEMLOCK là 0, nhưng quy trình không sở hữu khả năng CAP_IPC_LOCK (xem lại "Giới hạn khóa").

Một tiến trình con không kế thừa bộ nhớ bị khóa qua một fork( ). Tuy nhiên, do hành vi sao chép khi ghi của không gian địa chỉ trong Linux, các trang của tiến trình con thực sự bị khóa trong bộ nhớ cho đến khi tiến trình con ghi vào chúng.

Ví dụ, giả sử một chương trình lưu trữ một chuỗi đã giải mã trong bộ nhớ. Một tiến trình có thể khóa trang chứa chuỗi đó bằng mã như sau:

```
int ret;

/* khóa 'bí mật' trong bộ nhớ
*/ ret = mlock (bí mật, strlen (bí mật));
nếu
    (ret) perror ("mlock");
```

Khóa toàn bộ không gian địa chỉ Nếu một tiến

trình muốn khóa toàn bộ không gian địa chỉ của nó vào bộ nhớ vật lý, mlock() là một giao diện cồng kềnh. Với mục đích như vậy—phổ biến đối với các ứng dụng thời gian thực—POSIX định nghĩa một lệnh gọi hệ thống khóa toàn bộ không gian địa chỉ:

```
#include <sys/mman.h>

int mlockall (int flags);
```

Một lệnh gọi đến mlockall() sẽ khóa tất cả các trang trong không gian địa chỉ của tiến trình hiện tại vào bộ nhớ vật lý. Tham số flags , là phép OR bitwise của hai giá trị sau, điều khiển hành vi:

MCL_CURRENT
: Nếu được đặt, giá trị này sẽ hướng dẫn mlockall() khóa tất cả các trang hiện đang được ánh xạ—ngăn xếp, phân đoạn dữ liệu, tệp được ánh xạ, v.v.—vào không gian địa chỉ của quy trình.

MCL_FUTURE
: Nếu được đặt, giá trị này sẽ hướng dẫn mlockall() đảm bảo rằng tất cả các trang được ánh xạ vào không gian địa chỉ trong tương lai cũng được khóa vào bộ nhớ.

Hầu hết các ứng dụng đều chỉ định phép toán OR từng bit của cả hai giá trị.

Nếu thành công, lệnh gọi trả về 0; nếu thất bại, lệnh gọi trả về -1 và đặt errno thành một trong các mã lỗi sau:

EINVAL
: Tham số flags là số âm.

ENOMEM
: Người gọi đã cố gắng khóa nhiều trang hơn giới hạn tài nguyên RLIMIT_MEMLOCK cho phép (xem phần sau “Giới hạn khóa”).

EPERM
: Giới hạn tài nguyên RLIMIT_MEMLOCK là 0, nhưng quy trình không sở hữu khả năng CAP_IPC_LOCK (xem lại “Giới hạn khóa”).

Mở khóa bộ nhớ

Để mở khóa các trang khỏi bộ nhớ vật lý, một lần nữa cho phép nhân hoán đổi các trang ra đĩa khi cần, POSIX chuẩn hóa thêm hai giao diện:

```
#include <sys/mman.h>

int munlock (const void *addr, size_t len);
int munlockall (void);
```

Hệ thống gọi munlock( ) mở khóa các trang bắt đầu từ addr và mở rộng cho len byte. Nó hoàn tác các hiệu ứng của mlock( ). Hệ thống gọi munlockall( ) hoàn tác

tác dụng của mlockall( ). Cả hai lệnh gọi đều trả về 0 nếu thành công, và trả về -1 nếu có lỗi, và đặt errno thành một trong những giá trị sau:

EINVAL

Tham số len không hợp lệ (chỉ áp dụng cho munlock()).

ENOMEM

Một số trang được chỉ định không hợp lệ.

EPERM

Giới hạn tài nguyên RLIMIT_MEMLOCK là 0, nhưng quy trình không sở hữu khả năng CAP_IPC_LOCK (xem phần tiếp theo, "Giới hạn khóa").

Khóa bộ nhớ không lồng nhau. Do đó, một mlock() hoặc munlock() duy nhất sẽ mở khóa một trang bị khóa, bất kể trang đó đã bị khóa bao nhiêu lần thông qua mlock() hoặc mlockall( ).

Giới hạn khóa Vì việc

khóa bộ nhớ có thể ảnh hưởng đến hiệu suất chung của hệ thống—thực tế, nếu quá nhiều trang bị khóa, việc phân bổ bộ nhớ có thể không thành công—Linux đặt ra giới hạn về số lượng trang mà một tiến trình có thể khóa.

Các tiến trình sở hữu khả năng CAP_IPC_LOCK có thể khóa bất kỳ số trang nào vào bộ nhớ. Các tiến trình không có khả năng này chỉ có thể khóa các byte RLIMIT_MEMLOCK . Theo mặc định, giới hạn tài nguyên này là 32 KB—đủ lớn để khóa một hoặc hai bí mật trong bộ nhớ, nhưng không đủ lớn để ảnh hưởng xấu đến hiệu suất hệ thống. (Chương 6 thảo luận về giới hạn tài nguyên và cách truy xuất và thay đổi giá trị này.)

Một trang có nằm trong bộ nhớ vật lý không?

Với mục đích gỡ lỗi và chẩn đoán, Linux cung cấp hàm mincore() , có thể được sử dụng để xác định xem một phạm vi bộ nhớ nhất định có nằm trong bộ nhớ vật lý hay được hoán đổi sang đĩa hay không:

```
#include <unistd.h>
#include <sys/mman.h>

int mincore (void *start,
             size_t length,
             unsigned char *vec);
```

Một lệnh gọi đến mincore() cung cấp một vectơ mô tả các trang nào của ánh xạ nằm trong bộ nhớ vật lý tại thời điểm lệnh gọi hệ thống. Lệnh gọi trả về vectơ thông qua vec và mô tả các trang bắt đầu từ start (phải được căn chỉnh theo trang) và mở rộng theo byte độ dài (không cần phải căn chỉnh theo trang). Mỗi byte trong vec tương ứng với một trang trong phạm vi được cung cấp, bắt đầu bằng byte đầu tiên mô tả trang đầu tiên và di chuyển tuyến tính về phía trước. Do đó, vec phải đủ lớn để chứa (length-1+ page size) / page size byte. Bit có thứ tự thấp nhất trong mỗi

byte là 1 nếu trang nằm trong bộ nhớ vật lý và là 0 nếu không. Các bit khác hiện chưa được xác định và được dành riêng cho mục đích sử dụng trong tương lai.

Nếu thành công, lệnh gọi trả về 0. Nếu thất bại, lệnh gọi trả về -1 và đặt errno thành một trong những giá trị sau:

LẠI LẦN NỮA

Không có đủ tài nguyên hạt nhân để thực hiện yêu cầu.

EFAULT

Tham số vec trỏ đến một địa chỉ không hợp lệ.

EINVAL

Tham số bắt đầu không được căn chỉnh theo ranh giới trang.

ENOMEM

[địa chỉ, địa chỉ+1) chứa bộ nhớ không phải là một phần của ánh xạ dựa trên tệp.

Hiện tại, lệnh gọi hệ thống này chỉ hoạt động bình thường đối với các ánh xạ dựa trên tệp được tạo bằng MAP_SHARED. Điều này hạn chế đáng kể việc sử dụng lệnh gọi.

Phân bổ cơ hội

Linux sử dụng chiến lược phân bổ cơ hội . Khi một tiến trình yêu cầu bộ nhớ bổ sung từ nhân—ví dụ, bằng cách mở rộng phân đoạn dữ liệu của nó hoặc bằng cách tạo một ánh xạ bộ nhớ mới—nhân sẽ cam kết với bộ nhớ mà không thực sự cung cấp bất kỳ bộ nhớ vật lý nào. Chỉ khi tiến trình ghi vào bộ nhớ mới được phân bổ thì nhân mới đáp ứng cam kết bằng cách chuyển đổi cam kết cho bộ nhớ thành một phân bổ bộ nhớ vật lý. Nhân thực hiện điều này trên cơ sở từng trang, thực hiện phân trang theo yêu cầu và sao chép khi ghi khi cần.

Hành vi này có một số lợi thế. Đầu tiên, việc phân bổ bộ nhớ một cách lười biếng cho phép ker-nel hoãn hầu hết công việc cho đến thời điểm cuối cùng có thể—nếu thực sự nó phải đáp ứng các phân bổ. Thứ hai, vì các yêu cầu được đáp ứng từng trang và theo yêu cầu, nên chỉ có bộ nhớ vật lý trong quá trình sử dụng thực tế mới cần sử dụng bộ nhớ vật lý.
Cuối cùng, lượng bộ nhớ được cam kết có thể vượt xa lượng bộ nhớ vật lý và thậm chí là không gian hoán đổi khả dụng. Tính năng cuối cùng này được gọi là cam kết quá mức.

Overcommiting và OOM Overcommiting cho

phép hệ thống chạy nhiều ứng dụng hơn và lớn hơn nhiều so với khi mọi trang bộ nhớ được yêu cầu phải được sao lưu bằng bộ nhớ vật lý tại điểm phân bổ thay vì điểm sử dụng. Nếu không có overcommiting, việc ánh xạ một tệp 2 GB sao chép khi ghi sẽ yêu cầu nhân phải dành riêng 2 GB dung lượng lưu trữ. Với overcommitment, việc ánh xạ một tệp 2 GB chỉ yêu cầu lưu trữ cho mỗi trang dữ liệu mà quy trình thực sự ghi vào. Tương tự như vậy, nếu không có overcommiting, mọi fork() sẽ yêu cầu đủ dung lượng lưu trữ trống để sao chép không gian địa chỉ, mặc dù phần lớn các trang không bao giờ trải qua copy-on-write.

Tuy nhiên, nếu các tiến trình cố gắng đáp ứng nhiều cam kết nổi bật hơn bộ nhớ vật lý và không gian hoán đổi của hệ thống thì sao? Trong trường hợp đó, một hoặc nhiều sự thỏa mãn phải thất bại. Vì hạt nhân đã cam kết với bộ nhớ—lệnh gọi hệ thống yêu cầu cam kết trả về thành công—và một tiến trình đang cố gắng sử dụng bộ nhớ đã cam kết đó, thì cách duy nhất của hạt nhân là hủy một tiến trình, giải phóng bộ nhớ khả dụng.

Khi overcommitment dẫn đến bộ nhớ không đủ để đáp ứng yêu cầu đã cam kết, chúng ta nói rằng tình trạng out of memory (OOM) đã xảy ra. Để đáp ứng tình trạng OOM, kernel sử dụng OOM killer để chọn một tiến trình "xứng đáng" chấm dứt. Với mục đích này, kernel cố gắng tìm tiến trình ít quan trọng nhất đang tiêu thụ nhiều bộ nhớ nhất.

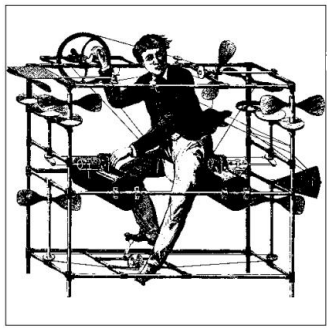
Điều kiện OOM rất hiếm—do đó tiện ích to lớn khi cho phép cam kết quá mức ngay từ đầu. Tuy nhiên, chắc chắn là những điều kiện này không được chào đón và việc chấm dứt không xác định một quy trình bởi trình diệt OOM thường không được chấp nhận.

Đối với các hệ thống có trường hợp này, hạt nhân cho phép vô hiệu hóa việc cam kết quá mức thông qua tệp /proc/sys/vm/overcommit_memory và tham số sysctl tương tự vm.overcommit_memory.

Giá trị mặc định cho tham số này, 0, hướng dẫn hạt nhân thực hiện chiến lược cam kết quá mức theo phương pháp heuristic, cam kết quá mức bộ nhớ trong phạm vi hợp lý, nhưng không cho phép cam kết quá mức nghiêm trọng. Giá trị 1 cho phép tất cả các cam kết thành công, bỏ qua sự thận trọng. Một số ứng dụng sử dụng nhiều bộ nhớ, chẳng hạn như trong lĩnh vực khoa học, có xu hướng yêu cầu nhiều bộ nhớ hơn mức chúng cần để thỏa mãn rằng tùy chọn như vậy là hợp lý.

Giá trị 2 vô hiệu hóa hoàn toàn các cam kết quá mức và cho phép tính toán chặt chẽ. Trong chế độ này, các cam kết bộ nhớ bị giới hạn ở kích thước của vùng hoán đổi, cộng với một tỷ lệ phần trăm có thể cấu hình của bộ nhớ vật lý. Tỷ lệ phần trăm cấu hình được đặt thông qua tệp /proc/sys/vm/overcommit_ratio hoặc tham số sysctl tương tự , là vm.overcommit_ratio. Mặc định là 50, giới hạn các cam kết bộ nhớ ở kích thước của vùng hoán đổi cộng với một nửa bộ nhớ vật lý. Vì bộ nhớ vật lý chứa hạt nhân, bảng trang, trang được hệ thống dành riêng, trang bị khóa, v.v., nên chỉ một phần của nó thực sự có thể hoán đổi và được đảm bảo có thể đáp ứng các cam kết.

Hãy cẩn thận với kế toán chặt chẽ! Nhiều nhà thiết kế hệ thống, phản đối khái niệm về kẻ giết chết OOM, nghĩ rằng kế toán chặt chẽ là một phương thuốc chữa bách bệnh. Tuy nhiên, các ứng dụng thường thực hiện nhiều phân bổ không cần thiết, đi sâu vào lãnh thổ quá mức cam kết và cho phép hành vi này là một trong những động lực chính đằng sau bộ nhớ ảo.



CHƯƠNG 9

Tín hiệu

Tín hiệu là các ngắt phần mềm cung cấp cơ chế xử lý không đồng bộ
sự kiện. Những sự kiện này có thể bắt nguồn từ bên ngoài hệ thống—chẳng hạn như khi người dùng
tạo ra ký tự ngắt (thường thông qua Ctrl-C)—hoặc từ các hoạt động trong
chương trình hoặc hạt nhân, chẳng hạn như khi tiến trình thực thi mã chia cho số không. Là một
dạng nguyên thủy của giao tiếp giữa các tiến trình (IPC), một tiến trình cũng có thể gửi tín hiệu đến một
tiến trình khác.

Điểm mấu chốt không chỉ là các sự kiện xảy ra không đồng bộ—ví dụ, người dùng có thể nhấn Ctrl-C tại bất
kỳ thời điểm nào trong quá trình thực thi chương trình—mà còn là
chương trình xử lý các tín hiệu không đồng bộ. Các hàm xử lý tín hiệu được đăng ký với hạt nhân, gọi các
hàm không đồng bộ từ phần còn lại của
chương trình khi tín hiệu được truyền đi.

Signals đã là một phần của Unix từ những ngày đầu. Tuy nhiên, theo thời gian, chúng đã
đã phát triển—đáng chú ý nhất là về độ tin cậy, vì tín hiệu có thể bị mất một lần, và trong
về mặt chức năng, vì các tín hiệu hiện có thể mang tải trọng do người dùng xác định. Lúc đầu, các hệ
thống Unix khác nhau đã thực hiện các thay đổi không tương thích đối với các tín hiệu. Rất may, POSIX đã ra đời
để giải cứu và xử lý tín hiệu chuẩn hóa. Tiêu chuẩn này là những gì Linux cung cấp và là những gì chúng
ta sẽ thảo luận ở đây.

Trong chương này, chúng ta sẽ bắt đầu với phần tổng quan về các tín hiệu và thảo luận về cách sử dụng chúng.
và sử dụng sai. Sau đó chúng ta sẽ tìm hiểu về các giao diện Linux khác nhau quản lý và thao tác tín hiệu.

Hầu hết các ứng dụng không tầm thường đều tương tác với tín hiệu. Ngay cả khi bạn cố tình thiết kế
ứng dụng của bạn không dựa vào tín hiệu cho nhu cầu giao tiếp của nó—thường là một điều tốt
ý tưởng!—bạn vẫn sẽ buộc phải làm việc với các tín hiệu trong một số trường hợp nhất định, chẳng hạn như
khi xử lý việc kết thúc chương trình.

Khái niệm tín hiệu

Tín hiệu có vòng đời rất chính xác. Đầu tiên, tín hiệu được đưa ra (đôi khi chúng ta cũng nói rằng nó được gửi hoặc được tạo ra). Sau đó, hạt nhân lưu trữ tín hiệu cho đến khi có thể phân phối. Cuối cùng, khi đã rảnh để làm như vậy, nhân xử lý tín hiệu theo cách phù hợp. Nhân có thể thực hiện một trong ba hành động, tùy thuộc vào những gì quy trình yêu cầu nó thực hiện: Bỏ qua

tín hiệu Không có

hành động nào được thực hiện. Có hai tín hiệu không thể bỏ qua: SIGKILL và SIGSTOP. Lý do cho điều này là quản trị viên hệ thống cần có khả năng giết hoặc dừng các quy trình và sẽ là hành vi lách luật quyền đó nếu một quy trình có thể chọn bỏ qua SIGKILL (khiến nó không thể giết được) hoặc SIGSTOP (khiến nó không thể dừng được).

Bắt và xử lý tín hiệu

Kernel sẽ tạm dừng thực thi đường dẫn mã hiện tại của tiến trình và nhảy đến một hàm đã đăng ký trước đó. Sau đó, tiến trình sẽ thực thi hàm này.
Khi tiến trình trả về từ hàm này, nó sẽ quay lại vị trí ban đầu khi bắt được tín hiệu.

SIGINT và SIGTERM là hai tín hiệu thường bị bắt. Các tiến trình bắt SIGINT để xử lý người dùng tạo ký tự ngắt—ví dụ, một thiết bị đầu cuối có thể bắt tín hiệu này và quay lại dấu nhắc chính. Các tiến trình bắt SIGTERM để thực hiện dọn dẹp cần thiết, chẳng hạn như ngắt kết nối khỏi mạng hoặc xóa các tệp tạm thời, trước khi kết thúc. Không thể bắt SIGKILL và SIGSTOP .

Thực hiện hành động mặc định

Hành động này phụ thuộc vào tín hiệu được gửi. Hành động mặc định thường là chấm dứt quy trình. Ví dụ , đây là trường hợp của SIGKILL . Tuy nhiên, nhiều tín hiệu được cung cấp cho các mục đích cụ thể liên quan đến lập trình viên trong các tình huống cụ thể và các tín hiệu này bị bỏ qua theo mặc định vì nhiều chương trình không quan tâm đến chúng. Chúng ta sẽ xem xét các tín hiệu khác nhau và các hành động mặc định của chúng ngay sau đây.

Theo truyền thống, khi một tín hiệu được truyền đi, hàm xử lý tín hiệu không có thông tin về những gì đã xảy ra ngoại trừ thực tế là một tín hiệu cụ thể đã xảy ra. Ngày nay, hạt nhân có thể cung cấp rất nhiều ngữ cảnh cho các lập trình viên muốn nhận nó và các tín hiệu thậm chí có thể truyền dữ liệu do người dùng xác định, giống như các cơ chế IPC tiên tiến hơn và sau này.

Signal Identifiers Mỗi

tín hiệu đều có tên tượng trưng bắt đầu bằng tiền tố SIG. Ví dụ, SIGINT là tín hiệu được gửi khi người dùng nhấn Ctrl-C, SIGABRT là tín hiệu được gửi khi tiến trình gọi hàm abort() và SIGKILL là tín hiệu được gửi khi tiến trình bị chấm dứt cưỡng bức.

Tất cả các tín hiệu này đều được định nghĩa trong một tệp tiêu đề được bao gồm từ <signal.h>. Các tín hiệu là chỉ đơn giản là các định nghĩa tiền xử lý biểu diễn các số nguyên dương—tức là mọi tín hiệu cũng được liên kết với một định danh số nguyên. Việc ánh xạ tên thành số nguyên cho các tín hiệu phụ thuộc vào việc triển khai và khác nhau giữa các hệ thống Unix, mặc dù đầu tiên khoảng một tá tín hiệu thường được ánh xạ theo cùng một cách (SIGKILL khét tiếng là tín hiệu 9, Ví dụ). Một lập trình viên giỏi sẽ luôn sử dụng tên tín hiệu dễ đọc đối với con người, và không bao giờ là giá trị nguyên của nó.

Các số tín hiệu bắt đầu từ 1 (thường là SIGHUP) và tiến triển theo đường thẳng hướng lên trên. có tổng cộng khoảng 31 tín hiệu, nhưng hầu hết các chương trình chỉ xử lý thường xuyên một số ít chúng. Không có tín hiệu nào có giá trị 0, đây là một giá trị đặc biệt được gọi là null tín hiệu. Thực sự không có gì quan trọng về tín hiệu null—nó không xứng đáng tên đặc biệt—nhưng một số lệnh gọi hệ thống (chẳng hạn như kill( )) sử dụng giá trị 0 làm tên đặc biệt trường hợp.

Bạn có thể tạo danh sách các tín hiệu được hỗ trợ trên hệ thống của mình bằng lệnh kill -l.

Tín hiệu được hỗ trợ bởi Linux

Bảng 9-1 liệt kê các tín hiệu mà Linux hỗ trợ.

Bảng 9-1. Tín hiệu

Tín hiệu	Sự miêu tả	Hành động mặc định
SIGABRT	Được gửi bởi abort( )	Kết thúc bằng core dump
SIGALRM	Được gửi bởi báo động ( )	chấm dứt
SIGBUS	Lỗi phần cứng hoặc căn chỉnh	Kết thúc bằng core dump
SIGCHLD	Trẻ em đã chấm dứt	Bỏ qua
SIGCONT	Quá trình vẫn tiếp tục sau khi đã dừng	Bỏ qua
SIGFPE	Ngoại lệ số học	Kết thúc bằng core dump
CÚT	Thiết bị đầu cuối điều khiển của quy trình đã bị đóng (thường là người dùng đã đăng nhập) ngoài)	chấm dứt
DẤU HIỆU	Tiến trình đã cố gắng thực hiện một lệnh bất hợp pháp	Kết thúc bằng core dump
TÍN HIỆU	Người dùng tạo ra ký tự ngắt (Ctrl-C)	chấm dứt
SIGIO	Sự kiện I/O không đồng bộ	Chấm dứt
GIẾT NGƯỜI	Kết thúc tiến trình không thể bắt được	chấm dứt
SIGPIP	Tiến trình được ghi vào đường ống nhưng không có trình đọc nào	chấm dứt
SIGPROF	Bộ đếm thời gian định hình đã hết hạn	chấm dứt
SIGPWR	Mất điện	chấm dứt
TỪ BỎ	Người dùng tạo ký tự thoát (Ctrl-\)	Kết thúc bằng core dump
SIGSEGV	Vi phạm truy cập bộ nhớ	Kết thúc bằng core dump
SIGSTKFLT	Lỗi ngăn xếp đồng xử lý	Chấm dứtb
SIGSTOP	Tạm dừng thực hiện tiến trình	Dừng lại

Bảng 9-1. Tín hiệu (tiếp theo)

Tín hiệu	Sự miêu tả	Hành động mặc định
SIGSYS	Quá trình đã cố gắng thực hiện một lệnh gọi hệ thống không hợp lệ	Kết thúc bằng core dump
SIGTERM	Kết thúc quy trình có thể bắt được	chấm dứt
SIGTRAP	Điểm dừng gặp phải	Kết thúc bằng core dump
SIGTSTP	Người dùng tạo ký tự tạm dừng (Ctrl-Z)	Dừng lại
SIGTTIN	Tiến trình nền được đọc từ thiết bị đầu cuối điều khiển	Dừng lại
SIGTTOU	Tiến trình nền được ghi vào thiết bị đầu cuối điều khiển	Dừng lại
SIGURG	Đang chờ xử lý I/O khẩn cấp	Bỏ qua
SIGUSR1	Tín hiệu được xác định theo quy trình	chấm dứt
SIGUSR2	Tín hiệu được xác định theo quy trình	chấm dứt
SIGVTALRM	Được tạo bởi setitimer() khi được gọi bằng cờ ITIMER_VIRTUAL	chấm dứt
SIGWINCH	Kích thước của cửa sổ điều khiển thiết bị đầu cuối đã thay đổi	Bỏ qua
SIGXCPU	Đã vượt quá giới hạn tài nguyên bộ xử lý	Kết thúc bằng core dump
SIGXFSZ	Đã vượt quá giới hạn tài nguyên tệp	Kết thúc bằng core dump

Hành vi trên các hệ thống Unix khác, chẳng hạn như BSD, là bỏ qua tín hiệu này.

b Nhân Linux không còn tạo ra tín hiệu này nữa; nó chỉ được giữ lại để tương thích ngược.

Có một số giá trị tín hiệu khác tồn tại, nhưng Linux định nghĩa chúng tương đương với các giá trị khác giá trị: SIGINFO được định nghĩa là SIGPWR,* SIGIOT được định nghĩa là SIGABRT và SIGPOLL và SIGLOST được định nghĩa là SIGIO.

Bây giờ chúng ta đã có một bảng để tham khảo nhanh, hãy cùng xem xét từng tín hiệu trong chi tiết:

SIGABRT

Hàm abort() gửi tín hiệu này đến tiến trình gọi nó. Tiến trình sau đó kết thúc và tạo ra một tệp lõi. Trong Linux, các khẳng định như assert() gọi abort() khi điều kiện không thành công.

SIGALRM

Các hàm alarm() và setitimer() (với cờ ITIMER_REAL) gửi thông tin này tín hiệu cho quy trình đã gọi chúng khi báo động hết hạn. Chương 10 thảo luận về những chức năng này và các chức năng liên quan.

SIGBUS

Hạt nhân đưa ra tín hiệu này khi quá trình gặp lỗi phần cứng khác ngoài bảo vệ bộ nhớ, tạo ra một SIGSEGV. Trên các hệ thống Unix truyền thống, điều này tín hiệu biểu thị nhiều lỗi không thể phục hồi, chẳng hạn như truy cập bộ nhớ không được căn chỉnh. Tuy nhiên, hạt nhân Linux tự động sửa hầu hết các lỗi này mà không tạo ra tín hiệu. Hạt nhân sẽ đưa ra tín hiệu này khi một quy trình không đúng cách

* Chỉ có kiến trúc Alpha xác định tín hiệu này. Trên tất cả các kiến trúc máy khác, tín hiệu này không tồn tại.

truy cập vào vùng bộ nhớ được tạo qua mmap( ) (xem Chương 8 để thảo luận về ánh xạ bộ nhớ). Trừ khi tín hiệu này được bắt, hạt nhân sẽ chấm dứt quy trình và tạo ra một bản dump lõi.

SIGCHLD

Bất cứ khi nào một tiến trình kết thúc hoặc dừng, hạt nhân sẽ gửi tín hiệu này đến tiến trình cha. Vì SIGCHLD bị bỏ qua theo mặc định, các tiến trình phải bắt và xử lý nó một cách rõ ràng nếu chúng quan tâm đến mạng sống của các tiến trình con. Một trình xử lý cho tín hiệu này thường gọi wait( ), được thảo luận trong Chương 5, để xác định pid và mã thoát của tiến trình con.

SIGCONT

Kernel gửi tín hiệu này đến một tiến trình khi tiến trình được tiếp tục sau khi bị dừng. Theo mặc định, tín hiệu này bị bỏ qua, nhưng các tiến trình có thể bắt được tín hiệu này nếu chúng muốn thực hiện một hành động sau khi được tiếp tục. Tín hiệu này thường được sử dụng bởi các thiết bị đầu cuối hoặc trình soạn thảo, những thiết bị muốn làm mới màn hình.

SIGFPE

Mặc dù có tên như vậy, tín hiệu này biểu thị bất kỳ ngoại lệ số học nào, và không chỉ những ngoại lệ liên quan đến các phép toán dấu phẩy động. Các ngoại lệ bao gồm tràn, thiếu và chia cho số không. Hành động mặc định là chấm dứt quy trình và tạo tệp lõi, nhưng các quy trình có thể bắt và xử lý tín hiệu này nếu chúng muốn.

Lưu ý rằng hành vi của một quy trình và kết quả của hoạt động vi phạm là không xác định nếu quy trình đó quyết định tiếp tục chạy.

CÚT

Kernel gửi tín hiệu này đến session leader bất cứ khi nào terminal của session ngắt kết nối. Kernel cũng gửi tín hiệu này đến từng tiến trình trong nhóm tiến trình tiền cảnh khi session leader kết thúc. Hành động mặc định là kết thúc, điều này có lý—tín hiệu cho biết người dùng đã đăng xuất.

Tiến trình daemon “quá tải” tín hiệu này bằng một cơ chế hướng dẫn chúng tải lại các tệp cấu hình của chúng. Ví dụ, gửi SIGHUP đến Apache sẽ hướng dẫn nó đọc lại httpd.conf. Sử dụng SIGHUP cho mục đích này là một quy ước chung, nhưng không bắt buộc. Thực hành này an toàn vì daemon không có thiết bị đầu cuối điều khiển, và do đó thông thường không bao giờ nhận được tín hiệu này.

DẤU HIỆU

Hạt nhân gửi tín hiệu này khi một tiến trình cố gắng thực hiện một lệnh máy bất hợp pháp. Hành động mặc định là chấm dứt tiến trình và tạo ra một bản sao lưu lõi. Các tiến trình có thể chọn bắt và xử lý SIGILL, nhưng hành vi của chúng không được xác định sau khi xảy ra.

SIGINT

Tín hiệu này được gửi đến tất cả các quy trình trong nhóm quy trình tiền cảnh khi người dùng nhập ký tự ngắt (thường là Ctrl-C). Hành vi mặc định là chấm dứt; tuy nhiên, các quy trình có thể chọn bắt và xử lý tín hiệu này và thường làm như vậy để dọn dẹp trước khi chấm dứt.

SIGIO

Tín hiệu này được gửi khi sự kiện I/O bất đồng bộ kiểu BSD được tạo ra. Kiểu I/O này hiếm khi được sử dụng trên Linux. (Xem Chương 4 để thảo luận về các kỹ thuật I/O nâng cao phổ biến trong Linux.)

SIGKILL

Tín hiệu này được gửi từ lệnh gọi hệ thống kill() ; nó tồn tại để cung cấp cho người quản trị hệ thống một cách chắc chắn để vô điều kiện giết một tiến trình. Tín hiệu này không thể bị bắt hoặc bỏ qua, và kết quả của nó luôn là chấm dứt tiến trình.

SIGPIPE

Nếu một tiến trình ghi vào một đường ống, nhưng trình đọc đã kết thúc, thì hạt nhân sẽ đưa ra tín hiệu này. Hành động mặc định là kết thúc tiến trình, nhưng tín hiệu này có thể được bắt và xử lý.

SIGPROF

Hàm setitimer () , khi được sử dụng với cờ ITIMER_PROF , sẽ tạo ra tín hiệu này khi bộ đếm thời gian lập hồ sơ hết hạn. Hành động mặc định là chấm dứt quy trình.

SIGPWR

Tín hiệu này phụ thuộc vào hệ thống. Trên Linux, nó biểu thị tình trạng pin yếu (như trong bộ nguồn điện liên tục hoặc UPS). Một dae-mon giám sát UPS gửi tín hiệu này đến init, sau đó init phản hồi bằng cách dọn dẹp và tắt hệ thống—hy vọng là trước khi mất điện!

SIGQUIT

Hạt nhân sẽ đưa ra tín hiệu này cho tất cả các tiến trình trong nhóm tiến trình tiền cảnh khi người dùng cung cấp ký tự thoát thiết bị đầu cuối (thường là Ctrl-\). Hành động mặc định là chấm dứt các tiến trình và tạo một bản sao lưu lõi.

SIGSEGV

Tín hiệu này, có tên bắt nguồn từ vi phạm phân đoạn, được gửi đến một tiến trình khi nó cố gắng truy cập bộ nhớ không hợp lệ. Điều này bao gồm truy cập bộ nhớ chưa được ánh xạ, đọc từ bộ nhớ không được bật chế độ đọc, thực thi mã trong bộ nhớ không được bật chế độ thực thi hoặc ghi vào bộ nhớ không được bật chế độ ghi. Các tiến trình có thể bắt và xử lý tín hiệu này, nhưng hành động mặc định là chấm dứt tiến trình và tạo bản sao lưu lõi.

SIGSTOP

Tín hiệu này chỉ được gửi bởi kill( ). Nó dừng một tiến trình vô điều kiện và không thể bị bắt hoặc bỏ qua.

SIGSYS

Hạt nhân gửi tín hiệu này đến một tiến trình khi nó cố gắng gọi một lệnh gọi hệ thống không hợp lệ. Điều này có thể xảy ra nếu một tệp nhị phân được xây dựng trên phiên bản mới hơn của hệ điều hành (với các phiên bản mới hơn của lệnh gọi hệ thống), nhưng sau đó chạy trên phiên bản cũ hơn. Các tệp nhị phân được xây dựng đúng cách thực hiện lệnh gọi hệ thống của chúng thông qua glibc sẽ không bao giờ nhận được tín hiệu này. Thay vào đó, các lệnh gọi hệ thống không hợp lệ sẽ trả về -1 và đặt errno thành ENOSYS.

SIGTERM

Tín hiệu này chỉ được gửi bởi kill( ); nó cho phép người dùng chấm dứt tiến trình một cách nhẹ nhàng (hành động mặc định). Các tiến trình có thể chọn bắt tín hiệu này và dọn dẹp trước khi chấm dứt, nhưng sẽ bị coi là thô lỗ nếu bắt tín hiệu này và không chấm dứt ngay lập tức.

SIGTRAP

Nhân gửi tín hiệu này đến một tiến trình khi nó vượt qua điểm dừng. Nói chung, trình gỡ lỗi sẽ bắt được tín hiệu này và các tiến trình khác sẽ bỏ qua nó.

SIGTSTP

Hạt nhân gửi tín hiệu này đến tất cả các tiến trình trong nhóm tiến trình nền trước khi người dùng nhập ký tự tạm dừng (thường là Ctrl-Z).

SIGTTIN

Tín hiệu này được gửi đến một tiến trình đang ở chế độ nền khi nó cố gắng đọc từ thiết bị đầu cuối điều khiển của nó. Hành động mặc định là dừng tiến trình.

SIGTTOU

Tín hiệu này được gửi đến một tiến trình đang ở chế độ nền khi nó cố gắng ghi vào thiết bị đầu cuối điều khiển của nó. Hành động mặc định là dừng tiến trình.

SIGURG

Hạt nhân gửi tín hiệu này đến một tiến trình khi dữ liệu ngoài băng tần (OOB) đã đến ổ cắm. Dữ liệu ngoài băng tần nằm ngoài phạm vi của cuốn sách này.

SIGUSR1 và SIGUSR2 Các

tín hiệu này có sẵn cho mục đích do người dùng xác định; hạt nhân không bao giờ kích hoạt chúng. Các tiến trình có thể sử dụng SIGUSR1 và SIGUSR2 cho bất kỳ mục đích nào chúng thích. Một cách sử dụng phổ biến là hướng dẫn một tiến trình daemon hoạt động khác đi. Hành động mặc định là chấm dứt tiến trình.

SIGVTALRM

Hàm setitimer() gửi tín hiệu này khi bộ đếm thời gian được tạo bằng cờ ITIMER_VIRTUAL hết hạn. Chương 10 thảo luận về bộ đếm thời gian.

SIGWINCH

Kernel sẽ đưa ra tín hiệu này cho tất cả các tiến trình trong nhóm tiến trình tiền cảnh khi kích thước cửa sổ terminal của chúng thay đổi. Theo mặc định, các tiến trình sẽ bỏ qua tín hiệu này, nhưng chúng có thể chọn bắt và xử lý tín hiệu này nếu chúng biết kích thước cửa sổ terminal của chúng. Một ví dụ hay về chương trình bắt được tín hiệu này là top— hãy thử thay đổi kích thước cửa sổ của nó trong khi nó đang chạy và xem nó phản hồi như thế nào.

SIGXCPU

Kernel sẽ tăng tín hiệu này khi một tiến trình vượt quá giới hạn bộ xử lý mềm của nó. Kernel sẽ tiếp tục tăng tín hiệu này một lần mỗi giây cho đến khi tiến trình thoát hoặc vượt quá giới hạn bộ xử lý cứng của nó. Khi giới hạn cứng bị vượt quá, kernel sẽ gửi cho tiến trình một SIGKILL.

SIGXFSZ

Kernel sẽ đưa ra tín hiệu này khi một tiến trình vượt quá giới hạn kích thước tệp của nó. Hành động mặc định là chấm dứt tiến trình, nhưng nếu tín hiệu này bị bắt hoặc bỏ qua, lệnh gọi hệ thống dẫn đến vượt quá giới hạn kích thước tệp sẽ trả về -1 và đặt errno thành EFBIG.

Quản lý tín hiệu cơ bản

Giao diện đơn giản và lâu đời nhất để quản lý tín hiệu là hàm signal() .
Được định nghĩa theo tiêu chuẩn ISO C89, tiêu chuẩn này chỉ chuẩn hóa mẫu số chung thấp nhất của hỗ trợ tín hiệu, lệnh gọi hệ thống này rất cơ bản. Linux cung cấp khả năng kiểm soát tín hiệu nhiều hơn đáng kể thông qua các giao diện khác, chúng ta sẽ đề cập sau trong chương này. Vì signal() là cơ bản nhất và, nhờ có sự hiện diện của nó trong ISO C, khá phổ biến, chúng ta sẽ đề cập đến nó trước:

```
#include <tín hiệu.h>

kiểu dữ liệu void (*sighandler_t)(int);

sighandler_t signal (int signo, sighandler_t handler); Một
```

lệnh gọi thành công đến signal( ) sẽ xóa hành động hiện tại được thực hiện khi nhận được sign-nal signo và thay vào đó xử lý tín hiệu bằng trình xử lý tín hiệu được chỉ định bởi handler. signo là một trong những tên tín hiệu được thảo luận trong phần trước, chẳng hạn như SIGINT hoặc SIGUSR1. Hãy nhớ rằng một quy trình không thể bắt được cả SIGKILL và SIGSTOP, do đó việc thiết lập trình xử lý cho bất kỳ tín hiệu nào trong hai tín hiệu này đều không có ý nghĩa.

Hàm xử lý phải trả về giá trị void, điều này hợp lý vì (không giống như các hàm bình thường) không có vị trí chuẩn nào trong chương trình để hàm này trả về.
Hàm này lấy một đối số, một số nguyên, là mã định danh tín hiệu (ví dụ: SIGUSR2) của tín hiệu đang được xử lý. Điều này cho phép một hàm duy nhất xử lý nhiều tín hiệu. Một nguyên mẫu có dạng: void my_handler (int signo);

Linux sử dụng typedef, sighandler_t, để định nghĩa nguyên mẫu này. Các hệ thống Unix khác sử dụng trực tiếp các con trỏ hàm; một số hệ thống có các kiểu riêng, có thể không được đặt tên là sighandler_t. Các chương trình tìm kiếm khả năng di động không nên tham chiếu trực tiếp đến kiểu đó.

Khi nó đưa ra tín hiệu cho một tiến trình đã đăng ký trình xử lý tín hiệu, hạt nhân sẽ tạm dừng thực thi luồng lệnh thông thường của chương trình và gọi trình xử lý tín hiệu. Trình xử lý được truyền giá trị của tín hiệu, là signo ban đầu được cung cấp cho signal( ).

Bạn cũng có thể sử dụng signal() để hướng dẫn kernel bỏ qua tín hiệu đã cho cho tiến trình hiện tại hoặc đặt lại tín hiệu về hành vi mặc định. Điều này được thực hiện bằng cách sử dụng các giá trị đặc biệt cho tham số trình xử lý :

SIG_DFL

Đặt hành vi của tín hiệu được signo đưa ra thành mặc định. Ví dụ, trong trường hợp SIGPIPE, quy trình sẽ kết thúc.

SIG_IGN

Bỏ qua tín hiệu được đưa ra bởi signo.

Hàm signal () trả về hành vi trước đó của tín hiệu, có thể là con trỏ đến trình xử lý tín hiệu, SIG_DFL hoặc SIG_IGN. Khi có lỗi, hàm trả về SIG_ERR.
Nó không thiết lập lỗi.

Chờ đợi một tín hiệu, bất kỳ tín hiệu nào

Hữu ích cho việc gỡ lỗi và viết các đoạn mã minh họa, lệnh gọi hệ thống pause() do POSIX định nghĩa sẽ đưa một tiến trình vào trạng thái ngủ cho đến khi nhận được tín hiệu xử lý hoặc chấm dứt tiến trình:

```
#include <unistd.h>

int tạm dừng (void);
```

pause() chỉ trả về nếu nhận được tín hiệu bắt được, trong trường hợp đó tín hiệu được xử lý, và pause() trả về -1, và đặt errno thành EINTR. Nếu kernel đưa ra tín hiệu bị bỏ qua, tiến trình sẽ không thức dậy.

Trong nhân Linux, pause() là một trong những lệnh gọi hệ thống đơn giản nhất. Nó chỉ thực hiện hai hành động. Đầu tiên, nó đặt tiến trình vào trạng thái ngủ có thể ngắt. Tiếp theo, nó gọi schedule() để gọi trình lập lịch tiến trình Linux để tìm một tiến trình khác để chạy. Vì tiến trình không thực sự chờ đợi bất cứ điều gì, nên nhân sẽ không đánh thức nó trừ khi nó nhận được tín hiệu. Toàn bộ thử thách này chỉ tiêu tốn hai dòng mã C.*

Ví dụ Hãy

xem một vài ví dụ đơn giản. Ví dụ đầu tiên này đăng ký một trình xử lý tín hiệu cho SIGINT chỉ in một thông báo và sau đó kết thúc chương trình (như SIGINT vẫn làm):

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

/* trình xử lý cho SIGINT */
static void sigint_handler (int signo) {
```

* Do đó, pause() chỉ là lệnh gọi hệ thống đơn giản thứ hai. Những người chiến thắng chung cuộc là getpid() và gettid( ), mỗi lệnh chỉ có một dòng.

```
        /*
         * Về mặt kỹ thuật, bạn không nên sử dụng printf( ) trong trình
         xử lý tín hiệu, nhưng đó không phải là tận thế. Tôi sẽ thảo
         luận lý do tại sao trong phần "Reentrancy." */


        printf ("Đã phát hiện SIGINT!\n");
        thoát (EXIT_SUCCESS);
}

int main (void) {

        /*
         *
           Đăng ký sigint_handler làm trình xử lý tín hiệu * cho
         SIGINT. */


        nếu (tín hiệu (SIGINT, sigint_handler) == SIG_ERR) {
                fprintf (stderr, "Không thể xử lý SIGINT!\n"); thoát
                (EXIT_FAILURE);
        }

        đối với (;;)
                tạm dừng ( );

        trả về 0;
}
```

Trong ví dụ sau, chúng tôi đăng ký cùng một trình xử lý cho SIGTERM và SIGINT. Chúng tôi cũng đặt lại hành vi cho SIGPROF về mặc định (là chấm dứt quy trình) và bỏ qua SIGHUP (nếu không sẽ chấm dứt quy trình):

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

/* trình xử lý cho SIGINT */
void tĩnh xử lý tín hiệu (int signo) {

        nếu (signo == SIGINT)
                printf ("Đã bắt được SIGINT!\n");
        nếu không thì nếu (signo ==
                SIGTERM) printf ("Đã bắt được SIGTERM!\n");
        khác {
                /* điều này không bao giờ được phép
                xảy ra */ fprintf (stderr, "Tín hiệu không mong muốn!
                \n"); exit (EXIT_FAILURE);

        } thoát (EXIT_SUCCESS);
}

int main (void) {
```

```
        /*
         *
           Đăng ký signal_handler làm trình xử lý tín hiệu * cho
         SIGINT. */

        if (tín hiệu (SIGINT, signal_handler) == SIG_ERR) {
                  fprintf (stderr, "Không thể xử lý SIGINT!\n"); thoát
                  (EXIT_FAILURE);
        }

        /*
         *
           Đăng ký signal_handler làm trình xử lý tín hiệu * cho
         SIGTERM. */

        if (tín hiệu (SIGTERM, signal_handler) == SIG_ERR) {
                  fprintf (stderr, "Không thể xử lý SIGTERM!\n"); thoát
                  (EXIT_FAILURE);
        }

        /* Đặt lại hành vi của SIGPROF về mặc định. */ if (signal
        (SIGPROF, SIG_DFL) == SIG_ERR) {
                  fprintf (stderr, "Không thể thiết lập lại SIGPROF!
                  \n"); thoát (EXIT_FAILURE);
        }

        /* Bỏ qua SIGHUP. */ if
        (signal (SIGHUP, SIG_IGN) == SIG_ERR) { fprintf
                  (stderr, "Không thể bỏ qua SIGHUP!\n"); exit
                  (EXIT_FAILURE);
        }

        đối với (;;)
                  tạm dừng ( );

        trả về 0;
}
```

Thực hiện và kế thừa

Khi một tiến trình được thực thi lần đầu, tất cả các tín hiệu được đặt thành hành động mặc định của chúng, trừ khi tiến trình cha (tiến trình thực thi tiến trình mới) đang bỏ qua chúng; trong trường hợp này, tiến trình mới được tạo cũng sẽ bỏ qua các tín hiệu đó. Nói cách khác, bất kỳ tín hiệu nào được bắt bởi tiến trình cha sẽ được đặt lại thành hành động mặc định trong tiến trình mới và tất cả các tín hiệu khác vẫn giữ nguyên. Điều này có lý vì một tiến trình mới thực thi không chia sẻ không gian địa chỉ của tiến trình cha, do đó bất kỳ trình xử lý tín hiệu nào đã đăng ký có thể không tồn tại.

Hành vi này khi thực thi quy trình có một công dụng đáng chú ý: khi shell thực thi một quy trình "ở chế độ nền" (hoặc khi một quy trình nền khác thực thi một quy trình khác), quy trình mới được thực thi sẽ bỏ qua các ký tự ngắt và thoát.

Do đó, trước khi shell thực hiện một tiến trình nền, nó phải đặt SIGINT và SIGQUIT thành SIG_IGN. Do đó, các chương trình xử lý các tín hiệu này thường kiểm tra trước để đảm bảo chúng không bị bỏ qua. Ví dụ:

```
/* xử lý SIGINT, nhưng chỉ khi nó không bị bỏ qua */
if (signal (SIGINT, SIG_IGN) != SIG_IGN) { if
        (signal (SIGINT, sigint_handler) == SIG_ERR)
                fprintf (stderr, "Không xử lý được SIGINT!\n");
}

/* xử lý SIGQUIT, nhưng chỉ khi nó không bị bỏ qua */
if (signal (SIGQUIT, SIG_IGN) != SIG_IGN) { if
        (signal (SIGQUIT, sigquit_handler) == SIG_ERR)
                fprintf (stderr, "Không xử lý được SIGQUIT!\n");
}
```

Nhu cầu thiết lập hành vi tín hiệu để kiểm tra hành vi tín hiệu làm nổi bật một thiếu sót trong giao diện signal() . Sau đó, chúng ta sẽ nghiên cứu một hàm không có lỗi này.

Hành vi với fork() , như bạn có thể mong đợi, là khác nhau. Khi một tiến trình gọi fork( ), tiến trình con kế thừa chính xác cùng một ngữ nghĩa tín hiệu như tiến trình cha. Điều này cũng hợp lý, vì tiến trình con và tiến trình cha chia sẻ một không gian địa chỉ, và do đó trình xử lý tín hiệu của tiến trình cha tồn tại trong tiến trình con.

Ánh xạ số tín hiệu thành chuỗi Trong các ví dụ của

chúng tôi cho đến nay, chúng tôi đã mã hóa cứng tên của các tín hiệu. Nhưng đôi khi thuận tiện hơn (hoặc thậm chí là yêu cầu) là bạn có thể chuyển đổi số tín hiệu thành chuỗi biểu diễn tên của nó. Có một số cách để thực hiện việc này. Một là truy xuất chuỗi từ danh sách được xác định tĩnh:

```
extern const char * const sys_siglist[];
```

sys_siglist là một mảng chuỗi chứa tên của các tín hiệu được hệ thống hỗ trợ, được lập chỉ mục theo số tín hiệu.

Một giải pháp thay thế là giao diện psignal() do BSD định nghĩa , khá phổ biến đến mức Linux cũng hỗ trợ:

```
#include <tín hiệu.h>

void psignal (int signo, const char *msg);
```

Một lệnh gọi đến psignal( ) sẽ in ra stderr chuỗi bạn cung cấp làm đối số msg , theo sau là dấu hai chấm, một khoảng trắng và tên của tín hiệu do signo cung cấp. Nếu signo không hợp lệ, thông báo được in sẽ cho biết như vậy.

Một giao diện tốt hơn là strsignal( ). Giao diện này không được chuẩn hóa, nhưng Linux và nhiều hệ thống không phải Linux hỗ trợ giao diện này:

```
#define _GNU_SOURCE
#include <string.h>

char *strsignal (int signo);
```

Một lệnh gọi đến strsignal() trả về một con trỏ đến mô tả tín hiệu được signo đưa ra. Nếu signo không hợp lệ, mô tả được trả về thường sẽ nói như vậy (một số hệ thống Unix hỗ trợ hàm này trả về NULL thay thế). Chuỗi được trả về chỉ hợp lệ cho đến lần gọi tiếp theo của strsignal( ), do đó hàm này không an toàn cho luồng.

Sử dụng sys_siglist thường là lựa chọn tốt nhất của bạn. Sử dụng cách tiếp cận này, chúng ta có thể viết lại trình xử lý tín hiệu trước đó như sau:

```
void tĩnh xử lý tín hiệu (int signo) {

        printf ("Đã bắt được %s\n", sys_siglist[signo]);
}
```

Gửi tín hiệu

Lệnh gọi hệ thống kill() , cơ sở của tiện ích kill phổ biến , gửi tín hiệu từ một tiến trình tới một tiến trình khác:

```
#include <sys/types.h>
#include <signal.h>

int giết (pid_t pid, int signo);
```

Trong cách sử dụng thông thường (tức là nếu pid lớn hơn 0), kill() sẽ gửi tín hiệu signo đến tiến trình được xác định bởi pid.

Nếu pid là 0, signo sẽ được gửi đến mọi tiến trình trong nhóm tiến trình đang gọi.

Nếu pid là -1, signo được gửi đến mọi tiến trình mà tiến trình gọi có quyền gửi tín hiệu, ngoại trừ chính nó và init. Chúng ta sẽ thảo luận về các quyền điều chỉnh việc phân phối tín hiệu trong tiểu mục tiếp theo.

Nếu pid nhỏ hơn -1, signo sẽ được gửi đến nhóm quy trình -pid.

Khi thành công, kill() trả về 0. Cuộc gọi được coi là thành công miễn là một tín hiệu duy nhất được gửi. Khi thất bại (không có tín hiệu nào được gửi), cuộc gọi trả về -1 và đặt errno thành một trong những giá trị sau:

EINVAL

Tín hiệu được chỉ định bởi signo không hợp lệ.

EPERM

Tiến trình gọi không có đủ quyền để gửi tín hiệu đến bất kỳ tiến trình nào được yêu cầu.

CÂU CHUYỆN

Tiến trình hoặc nhóm tiến trình được biểu thị bằng pid không tồn tại hoặc trong trường hợp của một tiến trình, nó là một tiến trình zombie.

Quyền

Để gửi tín hiệu đến một tiến trình khác, tiến trình gửi cần có quyền thích hợp. Một tiến trình có khả năng CAP_KILL (thường là do root sở hữu) có thể gửi tín hiệu đến bất kỳ tiến trình nào. Nếu không có khả năng này, ID người dùng thực tế hoặc hiệu lực của tiến trình gửi phải bằng ID người dùng thực tế hoặc đã lưu của tiến trình nhận. Nói một cách đơn giản hơn, người dùng chỉ có thể gửi tín hiệu đến một tiến trình mà họ sở hữu.

Hệ thống Unix định nghĩa một ngoại lệ cho SIGCONT: một tiến trình có thể gửi tín hiệu này đến bất kỳ tiến trình nào khác trong cùng một phiên. ID người dùng không cần phải khớp.

Nếu signo là 0—tín hiệu null đã đề cập ở trên—lệnh gọi không gửi tín hiệu, nhưng vẫn thực hiện kiểm tra lỗi. Điều này hữu ích để kiểm tra xem một quy trình có quyền phù hợp để gửi quy trình đã cung cấp hay xử lý tín hiệu hay không.

Ví dụ

Sau đây là cách gửi SIGHUP tới tiến trình có ID tiến trình 1722:

```
int ret;

ret = kill (1722, SIGHUP); if
(ret)
      perror ("giết");
```

Đoạn mã này thực chất giống hệt như lệnh gọi tiện ích kill sau đây :

```
$ giết -HUP 1722
```

Để kiểm tra xem chúng ta có được phép gửi tín hiệu đến 1722 mà không thực sự gửi bất kỳ tín hiệu nào hay không, chúng ta có thể thực hiện như sau:

```
int ret;

ret = kill (1722, 0);
nếu
    (ret); /* chúng ta không có quyền */
khác
          ; /* chúng tôi có quyền */
```

Gửi tín hiệu cho chính bạn

Hàm raise() là một cách đơn giản để một tiến trình gửi tín hiệu cho chính nó:

```
#include <tín hiệu.h>

int raise (int signo);
```

Lời kêu gọi này:

```
nâng lên (signo);
```

tương đương với lệnh gọi sau:

```
kill (getpid (), signo);
```

Cuộc gọi trả về 0 khi thành công và giá trị khác không khi thất bại. Nó không đặt errno.

Gửi tín hiệu đến toàn bộ nhóm quy trình

Một chức năng tiện lợi khác giúp dễ dàng gửi tín hiệu đến tất cả các quy trình trong một nhóm quy trình nhất định, trong trường hợp việc phủ định ID nhóm quy trình và sử dụng kill() được coi là quá phức tạp:

```
#include <tín hiệu.h>

int killpg (int pgrp, int signo);
```

Lời kêu gọi này:

```
killpg (pgrp, signo);
```

tương đương với lệnh gọi sau:

```
giết (-pgrp, signo);
```

Điều này vẫn đúng ngay cả khi pgrp là 0, trong trường hợp đó killpg( ) sẽ gửi tín hiệu signo đến mọi tiến trình trong nhóm tiến trình đang gọi.

Nếu thành công, killpg() trả về 0. Nếu thất bại, nó trả về -1 và đặt errno thành một trong các giá trị sau:

EINVAL

Tín hiệu do signo chỉ định không hợp lệ.

EPERM

Tiến trình gọi không có đủ quyền để gửi tín hiệu đến bất kỳ tiến trình nào được yêu cầu.

ESRCH

Nhóm quy trình được biểu thị bằng pgrp không tồn tại.

Sự tái nhập

Khi hạt nhân đưa ra tín hiệu, một tiến trình có thể thực thi mã ở bất kỳ đâu. Ví dụ, nó có thể đang ở giữa một hoạt động quan trọng, nếu bị ngắt quãng, sẽ khiến tiến trình ở trạng thái không nhất quán—ví dụ, với cấu trúc dữ liệu chỉ được cập nhật một nửa hoặc phép tính chỉ được thực hiện một phần. Tiến trình thậm chí có thể đang xử lý một tín hiệu khác.

Trình xử lý tín hiệu không thể cho biết quy trình đang thực thi mã nào khi tín hiệu chạm vào; trình xử lý có thể chạy ở giữa bất kỳ thứ gì. Do đó, điều rất quan trọng là bất kỳ trình xử lý tín hiệu nào mà quy trình của bạn cài đặt đều phải rất cẩn thận về các hành động mà nó thực hiện và dữ liệu mà nó chạm vào. Trình xử lý tín hiệu phải cẩn thận không đưa ra giả định về những gì

quá trình đang thực hiện khi nó bị gián đoạn. Đặc biệt, họ phải thực hành thận trọng khi sửa đổi dữ liệu toàn cầu (tức là dữ liệu được chia sẻ). Nhìn chung, đó là một ý tưởng tốt cho trình xử lý tín hiệu không bao giờ chạm vào dữ liệu toàn cục; tuy nhiên, trong phần sắp tới, chúng tôi sẽ hãy xem xét một cách để tạm thời chặn việc cung cấp tín hiệu, như một cách để cho phép an toàn việc xử lý dữ liệu được chia sẻ bởi bộ xử lý tín hiệu và phần còn lại của quy trình.

Còn các cuộc gọi hệ thống và các chức năng thư viện khác thì sao? Nếu quy trình của bạn đang ở trong giữa việc ghi vào một tệp hoặc phân bổ bộ nhớ và trình xử lý tín hiệu ghi vào cùng một tệp hoặc cũng gọi malloc( )? Hoặc nếu một tiến trình đang ở giữa một cuộc gọi đến hàm sử dụng bộ đệm tĩnh, chẳng hạn như strsignal( ), khi tín hiệu được truyền đi?

Một số chức năng rõ ràng là không thể nhập lại. Nếu một chương trình đang trong quá trình thực hiện một hàm không tái nhập, một tín hiệu xảy ra và trình xử lý tín hiệu sau đó gọi tín hiệu đó hàm không tái nhập, hỗn loạn có thể xảy ra. Hàm tái nhập là hàm an toàn để gọi từ bên trong chính nó (hoặc đồng thời, từ một luồng khác trong cùng một tiến trình). Để đủ điều kiện là có thể nhập lại, một hàm không được thao tác dữ liệu tĩnh, phải chỉ thao tác dữ liệu được phân bổ trong ngăn xếp hoặc dữ liệu được cung cấp cho nó bởi người gọi và phải không gọi bất kỳ hàm không thể nhập lại nào.

Các hàm có thể nhập lại được đảm bảo

Khi viết trình xử lý tín hiệu, bạn phải giả định rằng quá trình bị ngắt có thể ở giữa một hàm không tái nhập (hoặc bất kỳ hàm nào khác). Do đó, trình xử lý tín hiệu chỉ được sử dụng các hàm có khả năng nhập lại.

Nhiều tiêu chuẩn khác nhau đã ban hành danh sách các hàm an toàn với tín hiệu—tức là có thể nhập lại và do đó an toàn khi sử dụng từ bên trong trình xử lý tín hiệu. Đáng chú ý nhất là POSIX.1-2003 và Đặc tả UNIX Đơn chỉ định một danh sách các chức năng được đảm bảo là có khả năng tái nhập và an toàn tín hiệu trên tất cả các nền tảng tuân thủ. Bảng 9-2 liệt kê các chức năng.

Bảng 9-2. Các hàm được đảm bảo có thể nhập lại an toàn để sử dụng trong tín hiệu

hủy bỏ( )	chấp nhận( )	truy cập( )
lỗi_aio( ) báo	aio_return( ) liên	aio_suspend ( )
động( )	kết( )	cfgetispeed( )
cfgetospeed( )	cfsetispeed( )	cfsetospeed ( )
chdir( )	chmod( )	chown( )
lấy thời gian_đồng	đóng( )	kết nối( )
hồ( )	dup( )	nhân đôi2( )
tạo( ) thực	execve( )	Thoát( )
hiện( )	fchmod( )	fchown ( )
_exit( )	fdatasync( )	cái nĩa( )
fcntl( )	fstat( )	đồng bộ hóa
fpathconf( )	getegid( )	lấy id( )
ftruncate( ) getgid( )	getgroups( )	lấy tên người dùng( )

Bảng 9-2. Các hàm được đảm bảo có thể nhập lại an toàn để sử dụng trong tín hiệu (tiếp theo)

getpgrp( )	getpid( )	lấy ppid( )
getsockname( )	getsockopt( )	lấy được ( )
kill( )	liên	Nghe( )
lseek( )	kết( )	mkdir ( )
mkfifo( )	lstat( )	đường dẫn( )
pause( )	mở( )	thăm dò ( )
posix_trace_event( ) read( )	ống( )	nâng lên( )
	pselect( )	nhận( )
recvfrom( )	đọcliên	đổi tên( )
rmdir( )	kết( )	sem_post ( )
send( )	recvmsg( )	gửi tới( )
setgid( )	chọn( )	tập hợp ( )
setsockopt( )	gửimsg( )	tắt máy( )
sigaction( )	setpgid( )	dấu hiệu( )
sigemptyset( )	setuid( )	thành viên sigis( )
signal( )	sigaddset( )	đang chờ đợi( )
sigprocmask( )	sigfillset( )	chữ ký( )
sigsuspend( )	sigpause( )	ổ cắm( )
socketpair( )		liên kết tượng trưng()
sysconf( )	sigqueue( )	dòng chảy tc
tcflush( )	ngủ( ) stat( )	tcgetpgrp()
tcsendbreak( )	tcdrain( )	tcsetpgrp()
time( )	tcgetattr( ) tcsetattr( )	hẹn giờ lấy thời gian ( )
timer_settime( )		umask( )
uname( )		thời gian( )
wait( )	timer_getoverrun( ) lần( ) hủy liên kết( ) waitpid( )	viết( )

Nhiều chức năng khác cũng an toàn, nhưng Linux và các hệ thống tuân thủ POSIX khác chỉ đảm bảo tính có thể nhập lại của những chức năng này.

Bộ tín hiệu

Một số hàm mà chúng ta sẽ xem xét sau trong chương này cần phải thao tác các tập hợp của các tín hiệu, chẳng hạn như tập hợp các tín hiệu bị chặn bởi một quy trình hoặc tập hợp các tín hiệu đang chờ xử lý đối với một quy trình. Các hoạt động tập hợp tín hiệu quản lý các tập hợp tín hiệu này:

```
#include <tín hiệu.h>

int sigemptyset (sigset_t *set);

int sigfillset (sigset_t *set);
```

```
int sigaddset (sigset_t *set, int signo);

int sigdelset (sigset_t *set, int signo);

int sigismember (const sigset_t *set, int signo);
```

sigemptyset() khởi tạo tập tín hiệu do set cung cấp, đánh dấu là rỗng (tất cả các tín hiệu bị loại trừ khỏi tập). sigfillset() khởi tạo tập tín hiệu do set cung cấp, đánh dấu là đầy (tất cả các tín hiệu có trong tập). Cả hai hàm đều trả về 0. Bạn nên gọi một trong hai hàm này trên một tập tín hiệu trước khi sử dụng tập tiếp theo. sigaddset() thêm signo vào tập tín hiệu

do set cung cấp, trong khi sigdelset() xóa signo khỏi tập tín hiệu do set cung cấp. Cả hai đều trả về 0 nếu thành công hoặc -1 nếu có lỗi, trong trường hợp đó errno được đặt thành mã lỗi EINVAL, biểu thị rằng signo là một định danh tín hiệu không hợp lệ.

sigismember( ) trả về 1 nếu signo nằm trong tập tín hiệu do set đưa ra, 0 nếu không và -1 nếu có lỗi. Trong trường hợp sau, errno lại được đặt thành EINVAL, biểu thị rằng signo không hợp lệ.

Nhiều hàm Signal Set Các hàm trước đó

đều được chuẩn hóa bởi POSIX và có trên bất kỳ hệ thống Unix hiện đại nào. Linux cũng cung cấp một số hàm không chuẩn:

```
#define _GNU_SOURCE
#define <signal.h>

int sigisemptyset (sigset_t *set);

int sigorset (sigset_t *đích, sigset_t *trái, sigset_t *phải);

int sigandset (sigset_t *đích, sigset_t *trái, sigset_t *phải);
```

sigisemptyset() trả về 1 nếu tập tín hiệu được set đưa ra là rỗng, và trả về 0 nếu không.

sigorset() đặt hợp (OR nhị phân) của các tập tín hiệu left và right trong dest. sigandset() đặt giao điểm (AND nhị phân) của các tập tín hiệu left và right trong dest. Cả hai đều trả về 0 nếu thành công và -1 nếu lỗi, đặt errno thành EINVAL.

Các chức năng này rất hữu ích, nhưng các chương trình muốn tuân thủ đầy đủ POSIX nên tránh sử dụng chúng.

Tín hiệu chặn

Trước đó, chúng ta đã thảo luận về tính tái nhập và các vấn đề phát sinh do trình xử lý tín hiệu chạy không đồng bộ, bất kỳ lúc nào. Chúng ta đã thảo luận về các hàm không được gọi từ bên trong trình xử lý tín hiệu vì bản thân chúng không tái nhập.

Nhưng nếu chương trình của bạn cần chia sẻ dữ liệu giữa trình xử lý tín hiệu và các nơi khác trong chương trình thì sao? Nếu có những phần thực thi chương trình của bạn trong đó

bạn không muốn bất kỳ sự gián đoạn nào, bao gồm cả từ trình xử lý tín hiệu? Chúng tôi gọi những phần như vậy của một chương trình là vùng quan trọng và chúng tôi bảo vệ chúng bằng cách tạm thời dừng việc phân phối tín hiệu. Chúng tôi nói rằng các tín hiệu như vậy bị chặn. Bất kỳ tín hiệu nào được đưa ra trong khi bị chặn sẽ không được xử lý cho đến khi chúng được bỏ chặn. Một quy trình có thể chặn bất kỳ số lượng tín hiệu nào; tập hợp các tín hiệu bị một quy trình chặn được gọi là mặt nạ tín hiệu của quy trình đó.

POSIX định nghĩa và Linux triển khai một hàm để quản lý mặt nạ tín hiệu của quy trình:

```
#include <tín hiệu.h>

int sigprocmask (int how,
                 const sigset_t *set,
                 sigset_t *oldset);
```

Hành vi của sigprocmask() phụ thuộc vào giá trị của how, đây là một trong các cờ sau:

SIG_SETMASK

Mặt nạ tín hiệu cho quy trình gọi được thay đổi thành thiết lập.

SIG_BLOCK

Các tín hiệu trong tập hợp được thêm vào mặt nạ tín hiệu của quy trình gọi. Nói cách khác, mặt nạ tín hiệu được thay đổi thành hợp (OR nhị phân) của mặt nạ hiện tại và tập hợp.

SIG_BỎ CHẶN

Các tín hiệu trong tập hợp được loại bỏ khỏi mặt nạ tín hiệu của quy trình gọi. Nói cách khác, tín hiệu được thay đổi thành giao điểm (AND nhị phân) của mặt nạ hiện tại và phủ định (NOT nhị phân) của tập hợp. Việc bỏ chặn tín hiệu không bị chặn là bất hợp pháp.

Nếu oldset không phải là NULL, hàm sẽ đặt tập tín hiệu trước đó vào oldset.

Nếu set là NULL, hàm sẽ bỏ qua how và không thay đổi mặt nạ tín hiệu, nhưng nó sẽ đặt mặt nạ tín hiệu vào oldset. Nói cách khác, truyền giá trị null dưới dạng set là cách để lấy mặt nạ tín hiệu hiện tại.

Nếu thành công, lệnh gọi trả về 0. Nếu thất bại, lệnh gọi trả về -1 và đặt errno thành EINVAL, biểu thị rằng how không hợp lệ hoặc EFAULT, biểu thị rằng set hoặc oldset là con trỏ không hợp lệ.

Không được phép chặn SIGKILL hoặc SIGSTOP . sigprocmask( ) sẽ âm thầm bỏ qua mọi nỗ lực thêm tín hiệu vào mặt nạ tín hiệu.

Truy xuất tín hiệu đang chờ Khi kernel

đưa ra tín hiệu bị chặn, nó không được chuyển giao. Chúng tôi gọi những tín hiệu như vậy là đang chờ. Khi tín hiệu đang chờ được bỏ chặn, kernel sau đó chuyển nó cho tiến trình để xử lý.

POSIX định nghĩa một hàm để lấy tập hợp các tín hiệu đang chờ xử lý:

```
#include <tín hiệu.h>

int đang chờ (sigset_t *set);
```

Một lệnh gọi thành công đến sigpending() sẽ đặt tập hợp các tín hiệu đang chờ xử lý vào set và trả về 0. Khi lệnh gọi thất bại, lệnh gọi sẽ trả về -1 và đặt errno thành EFAULT, biểu thị rằng set là một con trỏ không hợp lệ.

Chờ một bộ tín hiệu

Hàm thứ ba được định nghĩa theo POSIX cho phép một tiến trình tạm thời thay đổi mặt nạ tín hiệu của nó, sau đó đợi cho đến khi có tín hiệu được đưa ra để kết thúc hoặc được tiến trình xử lý:

```
#include <tín hiệu.h>

int sigsuspend (const sigset_t *set); Nếu
```

một tín hiệu chấm dứt tiến trình, sigsuspend() không trả về. Nếu một tín hiệu được đưa ra và xử lý, sigsuspend() trả về -1 sau khi trình xử lý tín hiệu trả về, đặt errno thành EINTR. Nếu set là một con trỏ không hợp lệ, errno được đặt thành EFAULT.

Một kịch bản sử dụng sigsuspend() phổ biến là lấy các tín hiệu có thể đã đến và bị chặn trong một vùng quan trọng của quá trình thực thi chương trình. Trước tiên, quy trình sử dụng sigprocmask() để chặn một tập hợp các tín hiệu, lưu mặt nạ cũ trong oldset. Sau khi thoát khỏi vùng quan trọng, quy trình sau đó gọi sigsuspend( ), cung cấp oldset cho set.

Quản lý tín hiệu nâng cao

Hàm signal () mà chúng ta đã học ở đầu chương này rất cơ bản.
Vì nó là một phần của thư viện C chuẩn, và do đó phải phản ánh các giả định tối thiểu về khả năng của hệ điều hành mà nó chạy, nên nó chỉ có thể cung cấp một mẫu số chung thấp nhất cho quản lý tín hiệu. Một giải pháp thay thế, POSIX chuẩn hóa lệnh gọi hệ thống sigaction() , cung cấp khả năng quản lý tín hiệu lớn hơn nhiều. Trong số những thứ khác, bạn có thể sử dụng nó để chặn việc tiếp nhận các tín hiệu được chỉ định trong khi trình xử lý của bạn chạy và để truy xuất nhiều loại dữ liệu về trạng thái hệ thống và quy trình tại thời điểm tín hiệu được đưa ra:

```
#include <tín hiệu.h>

int sigaction (int signo,
               const struct sigaction *act,
               struct sigaction *oldact);
```

Một lệnh gọi đến sigaction() sẽ thay đổi hành vi của tín hiệu được xác định bởi signo, có thể là bất kỳ giá trị nào ngoại trừ những giá trị được liên kết với SIGKILL và SIGSTOP. Nếu act không phải là NULL, lệnh gọi hệ thống sẽ thay đổi hành vi hiện tại của tín hiệu theo chỉ định của act. Nếu

oldact không phải là NULL, lệnh gọi sẽ lưu trữ hành vi trước đó (hoặc hiện tại, nếu act là NULL) của tín hiệu đã cho tại đó.

Cấu trúc sigaction cho phép kiểm soát chi tiết các tín hiệu. Tiêu đề <sys/signal.h>, được bao gồm từ <signal.h>, định nghĩa cấu trúc như sau:

```
struct sigaction
        { void (*sa_handler)(int); /* trình xử lý tín hiệu hoặc hành
        động */ void (*sa_sigaction)(int, siginfo_t *, void
        *); sigset_t sa_mask; /* tín hiệu để chặn */ int
        sa_flags; /* cờ */ void (*sa_restorer)
        (void); /* lỗi thời và không phải POSIX */
}
```

Trường sa_handler chỉ thị hành động cần thực hiện khi nhận được tín hiệu. Giống như signal( ), trường này có thể là SIG_DFL, biểu thị hành động mặc định, SIG_IGN, hướng dẫn kernel bỏ qua tín hiệu cho tiến trình hoặc là con trỏ đến hàm xử lý tín hiệu. Hàm này có cùng nguyên mẫu với trình xử lý tín hiệu được cài đặt bởi signal( ):

```
void my_handler (int signo);
```

Nếu SA_SIGINFO được đặt trong sa_flags, sa_sigaction, chứ không phải sa_handler, sẽ chỉ định hàm xử lý tín hiệu. Nguyên mẫu của hàm này hơi khác một chút:

```
void my_handler (int signo, signinfo_t *si, void *ucontext);
```

Hàm này nhận số tín hiệu làm tham số đầu tiên, cấu trúc siginfo_t làm tham số thứ hai và cấu trúc ucontext_t (ép kiểu thành con trỏ void) làm tham số thứ ba. Nó không có giá trị trả về. Cấu trúc siginfo_t cung cấp rất nhiều thông tin cho trình xử lý tín hiệu; chúng ta sẽ xem xét nó ngay sau đây.

Lưu ý rằng trên một số kiến trúc máy (và có thể là các hệ thống Unix khác), sa_handler và sa_sigaction nằm trong một liên hợp và bạn không nên gán giá trị cho cả hai trường.

Trường sa_mask cung cấp một tập hợp các tín hiệu mà hệ thống phải chặn trong suốt thời gian thực thi trình xử lý tín hiệu. Điều này cho phép các lập trình viên thực thi bảo vệ thích hợp khỏi việc nhập lại giữa nhiều trình xử lý tín hiệu. Tín hiệu hiện đang được xử lý cũng bị chặn, trừ khi cờ SA_NODEFER được đặt trong sa_flags.
Bạn không thể chặn SIGKILL hoặc SIGSTOP; lệnh gọi sẽ tự động bỏ qua cả hai lệnh này trong sa_mask.

Trường sa_flags là một bitmask gồm không, một hoặc nhiều cờ thay đổi cách xử lý tín hiệu do signo đưa ra. Chúng ta đã xem xét các cờ SA_SIGINFO và SA_NODEFER ; các giá trị khác cho sa_flags bao gồm:

SA_NOCLDSTOP

Nếu signo là SIGCHLD, cờ này sẽ hướng dẫn hệ thống không cung cấp thông báo khi một tiến trình con dừng hoặc tiếp tục.

SA_NOCLDWAIT

Nếu signo là SIGCHLD, cờ này cho phép tự động thu hoạch con: con không được chuyển đổi thành zombie khi kết thúc và cha mẹ không cần (và không thể) gọi wait() trên chúng. Xem Chương 5 để thảo luận sôi nổi về con, zombie và wait( ).

SA_NOMASK

Cờ này là một cờ không phải POSIX tương đương với SA_NODEFER đã lỗi thời (đã thảo luận trước đó trong phần này). Sử dụng SA_NODEFER thay cho cờ này, nhưng hãy chuẩn bị để thấy giá trị này xuất hiện trong mã cũ hơn.

SA_ONESHOT Cờ

này là một cờ không phải POSIX tương đương với SA_RESETHAND đã lỗi thời (sẽ được thảo luận sau trong danh sách này). Sử dụng SA_RESETHAND thay cho cờ này, nhưng hãy chuẩn bị để thấy giá trị này xuất hiện trong mã cũ hơn.

SA_ONSTACK

Cờ này hướng dẫn hệ thống gọi trình xử lý tín hiệu đã cho trên một ngăn xếp tín hiệu thay thế, như được cung cấp bởi sigaltstack( ). Nếu bạn không cung cấp ngăn xếp thay thế, ngăn xếp mặc định sẽ được sử dụng—tức là hệ thống sẽ hoạt động như thể bạn không cung cấp cờ này. Các ngăn xếp tín hiệu thay thế rất hiếm, mặc dù chúng hữu ích trong một số ứng dụng pthreads có các ngăn xếp luồng nhỏ hơn có thể bị tràn ngập bởi một số cách sử dụng trình xử lý tín hiệu. Chúng tôi không thảo luận thêm về sigaltstack() trong cuốn sách này.

SA_RESTART Cờ

này cho phép khởi động lại theo kiểu BSD các lệnh gọi hệ thống bị gián đoạn bởi tín hiệu.

SA_RESETHAND Cờ

này kích hoạt chế độ “one-shot”. Hành vi của tín hiệu đã cho được đặt lại về mặc định khi trình xử lý tín hiệu trả về.

Trường sa_restorer đã lỗi thời và không còn được sử dụng trong Linux nữa. Dù sao thì nó cũng không phải là một phần của POSIX. Hãy giả vờ như nó không có ở đó và

đừng chạm vào nó. sigaction() trả về 0 khi thành công. Khi thất bại, lệnh gọi trả về -1 và đặt errno thành một trong các mã lỗi sau:

EFAULT

act hoặc oldact là một con trỏ không hợp lệ.

Dấu hiệu

EINVAL là tín hiệu không hợp lệ, SIGKILL hoặc SIGSTOP.

Cấu trúc siginfo_t Cấu trúc

siginfo_t cũng được định nghĩa trong <sys/signal.h> như sau:

```
typedef struct siginfo_t { int
        si_signo; /* số tín hiệu */ int si_errno; /* giá trị
        errno */ int si_code; /* mã tín hiệu */ pid_t
        si_pid; /* PID của tiến trình gửi */ uid_t
        si_uid; /* UID thực của tiến trình gửi */ int si_status; /* giá
        trị thoát hoặc tín hiệu */ clock_t si_utime; /* thời gian người dùng
        sử dụng */ clock_t si_stime; /* thời gian hệ thống sử dụng */
        sigval_t si_value; /* giá trị tải trọng tín hiệu */
```

```
            int si_int;          /* Tín hiệu POSIX.1b */ /*
            void *si_ptr;        Tín hiệu POSIX.1b */ /* vị
            void *si_addr;       trí bộ nhớ gây ra lỗi */ /* sự kiện băng tần */ /* mô
            int si_band;         tả tệp */
            int si_fd;
    };
```

Cấu trúc này chứa đầy thông tin được truyền đến trình xử lý tín hiệu (nếu bạn đang sử dụng sa_sigaction thay cho sa_sighandler). Với máy tính hiện đại, nhiều người coi mô hình tín hiệu Unix là một phương pháp tệ hại để thực hiện IPC. Có lẽ vấn đề là những người này bị mắc kẹt khi sử dụng signal() khi họ nên sử dụng sigaction() với SA_SIGINFO. Cấu trúc sigaction_t mở ra cánh cửa để khai thác nhiều chức năng hơn từ các tín hiệu.

Có rất nhiều dữ liệu thú vị trong cấu trúc này, bao gồm thông tin về quá trình gửi tín hiệu và nguyên nhân của tín hiệu. Sau đây là mô tả chi tiết về từng trường:

si_signo
: Số tín hiệu của tín hiệu đang được đề cập. Trong trình xử lý tín hiệu của bạn, đối số đầu tiên cũng cung cấp thông tin này (và tránh tham chiếu con trỏ). si_errno

Nếu khác
: không, mã lỗi được liên kết với tín hiệu này. Trường này hợp lệ với tất cả các tín hiệu.

si_code
: Giải thích lý do và nơi mà quy trình nhận được tín hiệu (ví dụ: từ kill( )). Chúng ta sẽ xem xét các giá trị có thể có trong phần sau. Trường này hợp lệ với tất cả các tín hiệu. si_pid Đối với SIGCHLD,

PID
: của quy trình đã kết thúc. si_uid Đối với SIGCHLD,

UID sở
: hữu của quy trình đã kết thúc. si_status Đối với SIGCHLD,

trạng
: thái thoát của quy trình đã kết thúc. si_utime Đối với

SIGCHLD,
: thời gian người dùng sử dụng bởi quy trình đã kết thúc. si_stime Đối

với
: SIGCHLD, thời gian hệ thống sử dụng bởi quy trình đã kết thúc. si_value

Hợp nhất
: của si_int và si_ptr. si_int

Đối
: với các tín hiệu được gửi qua sigqueue() (xem “Gửi tín hiệu có tải trọng” ở phần sau của chương này), tải trọng được cung cấp được nhập dưới dạng số nguyên.

si_ptr
    Đối với các tín hiệu được gửi qua sigqueue() (xem “Gửi tín hiệu với Payload” sau trong chương này), payload được cung cấp được nhập là một con trỏ

void .
    si_addr Đối với SIGBUS, SIGFPE, SIGILL, SIGSEGV và SIGTRAP, con trỏ void này chứa địa chỉ của lỗi vi phạm. Ví dụ, trong trường hợp của SIGSEGV, trường này chứa địa chỉ của vi phạm truy cập bộ nhớ (và do đó thường là NULL!). si_band Đối với SIGPOLL,

thông
    tin ngoài băng tần và ưu tiên cho mô tả tệp được liệt kê trong si_fd.

si_fd
    Đối với SIGPOLL, mô tả tệp cho tệp có thao tác đã hoàn tất.

si_value, si_int và si_ptr là những chủ đề đặc biệt phức tạp vì một tiến trình có thể sử dụng chúng để truyền dữ liệu tùy ý cho một tiến trình khác. Do đó, bạn có thể sử dụng chúng để gửi một số nguyên đơn giản hoặc một con trỏ đến một cấu trúc dữ liệu (lưu ý rằng con trỏ không giúp ích nhiều nếu các tiến trình không chia sẻ không gian địa chỉ). Các trường này được thảo luận trong phần sắp tới “Gửi tín hiệu với tải trọng”.

POSIX đảm bảo rằng chỉ có ba trường đầu tiên là hợp lệ cho tất cả các tín hiệu. Các trường khác chỉ nên được truy cập khi xử lý tín hiệu áp dụng. Ví dụ, bạn chỉ nên truy cập trường si_fd nếu tín hiệu là SIGPOLL.

Thế giới kỳ diệu của si_code Trường si_code chỉ

ra nguyên nhân của tín hiệu. Đối với tín hiệu do người dùng gửi, trường này chỉ ra cách tín hiệu được gửi. Đối với tín hiệu do kernel gửi, trường này chỉ ra lý do tại sao tín hiệu được gửi.

Các giá trị si_code sau đây có giá trị đối với bất kỳ tín hiệu nào. Chúng chỉ ra cách thức/lý do tín hiệu được gửi:

SI_ASYNCIO
    Tín hiệu được gửi do hoàn tất I/O không đồng bộ (xem Chương 5).

SI_KERNEL
    Tín hiệu được đưa ra bởi hạt nhân.

SI_MESGQ
    Tín hiệu được gửi do trạng thái của hàng đợi tin nhắn POSIX thay đổi (không được đề cập trong sách này).

SI_QUEUE
    Tín hiệu được gửi bởi sigqueue() (xem phần tiếp theo).

SI_TIMER
    Tín hiệu được gửi do bộ đếm thời gian POSIX hết hạn (xem Chương 10).

SI_TKILL

Tín hiệu được gửi bởi tkill() hoặc tgkill( ). Các lệnh gọi hệ thống này được sử dụng bởi các thư viện luồng và không được đề cập trong cuốn sách này.

SI_SIGIO

Tín hiệu được gửi do SIGIO đang xếp hàng.

SI_USER

Tín hiệu được gửi bởi kill() hoặc raise( ).

Các giá trị si_code sau đây chỉ hợp lệ cho SIGBUS . Chúng chỉ ra loại lỗi phần cứng đã xảy ra:

BUS_ADRALN

Quá trình này phát sinh lỗi căn chỉnh (xem Chương 8 để thảo luận về căn chỉnh).

BUS_ADRERR

Quá trình đã truy cập vào một địa chỉ vật lý không hợp lệ.

BUS_OBJERR

Quá trình này gây ra một số lỗi phần cứng khác.

Đối với SIGCHLD, các giá trị sau đây xác định những gì con đã làm để tạo ra tín hiệu được gửi đến cha mẹ của nó:

CLD_CONTINUED

Đứa trẻ đã dừng lại nhưng đã tiếp tục.

CLD_DUMPED

Chương trình con bị chấm dứt bất thường.

CLD_EXITED

Chương trình con được kết thúc bình thường thông qua exit( ).

CLD_KILLED

Đứa trẻ đã bị giết.

CLD_STOPPED

Đứa trẻ dừng lại.

CLD_TRAPPED

Đứa trẻ đã vướng phải bẫy.

Các giá trị sau đây chỉ hợp lệ đối với SIGFPE . Chúng giải thích loại lỗi số học đã xảy ra:

FPE_FLTDIV

Quá trình thực hiện một phép toán dấu phẩy động dẫn đến phép chia cho không.

FPE_FLTOVF

Quá trình này thực hiện một phép toán số thực dẫn đến tràn số.

FPE_FLTINV

Quá trình này đã thực hiện một phép toán dấu phẩy động không hợp lệ.

FPE_FLTRES

Quá trình này thực hiện một phép toán số thực mang lại kết quả không chính xác hoặc không hợp lệ.

FPE_FLTSUB

Quá trình này thực hiện một phép toán dấu phẩy động dẫn đến chỉ số nằm ngoài phạm vi.

FPE_FLTUND

Quá trình này thực hiện một phép toán dấu phẩy động dẫn đến tràn số.

FPE_INTDIV

Quá trình này thực hiện phép toán số nguyên dẫn đến phép chia cho số không.

FPE_INTOVF

Quá trình này thực hiện một phép toán số nguyên dẫn đến tràn số.

Các giá trị si_code sau đây chỉ hợp lệ đối với SIGILL . Chúng giải thích bản chất của việc thực thi lệnh bất hợp pháp:

ILL_ILLADR

Quá trình này đã cố gắng chuyển sang chế độ địa chỉ bất hợp pháp.

ILL_ILLOPC

Quá trình này đã cố gắng thực thi một mã lệnh bất hợp pháp.

ILL_ILLOPN

Tiến trình đã cố gắng thực thi một toán hạng không hợp lệ.

ILL_PRVOPC

Quá trình này đã cố gắng thực thi một mã lệnh đặc quyền.

ILL_PRVREG

Tiến trình đã cố gắng thực thi trên một thanh ghi đặc quyền.

ILL_ILLTRP

Quá trình này đã cố gắng xâm nhập vào một bẫy bất hợp pháp.

Đối với tất cả các giá trị này, si_addr trỏ đến địa chỉ của vi phạm.

Đối với SIGPOLL, các giá trị sau đây xác định sự kiện I/O tạo ra tín hiệu:

POLL_ERR

Đã xảy ra lỗi I/O.

POLL_HUP

Thiết bị bị treo hoặc ổ cắm bị ngắt kết nối.

POLL_IN

Tệp có dữ liệu có thể đọc.

POLL_MSG

Có tin nhắn.